Sandip Ray

Scalable Techniques for Formal Verification

Springer

Scalable Techniques for Formal Verification

Sandip Ray

Scalable Techniques for Formal Verification

Dr. Sandip Ray
Department of Computer Sciences
University of Texas, Austin
University Station 1
78712-0233 Austin Texas
MS C0500
USA
sandip@cs.utexas.edu

ISBN 978-1-4419-5997-3 e-ISBN 978-1-4419-5998-0
DOI 10.1007/978-1-4419-5998-0
Springer New York Dordrecht Heidelberg London

Library of Congress Control Number: 2010927798

Printed on acid-free paper

Springer is part of Springer Science+Business Media (www.springer.com)

To Anindita
Who showed me that sometimes dreams can come true

Preface

This book is about *formal verification*, that is, the use of mathematical reasoning to ensure correct execution of computing systems. With the increasing use of computing systems in safety-critical and security-critical applications, it is becoming increasingly important for our well-being to ensure that those systems execute correctly. Over the last decade, formal verification has made significant headway in the analysis of industrial systems, particularly in the realm of verification of hardware. A key advantage of formal verification is that it provides a mathematical guarantee of their correctness (up to the accuracy of formal models and correctness of reasoning tools). In the process, the analysis can expose subtle design errors. Formal verification is particularly effective in finding corner-case bugs that are difficult to detect through traditional simulation and testing. Nevertheless, and in spite of its promise, the application of formal verification has so far been limited in an industrial design validation tool flow. The difficulties in its large-scale adoption include the following (1) deductive verification using theorem provers often involves excessive and prohibitive manual effort and (2) automated decision procedures (*e.g.*, model checking) can quickly hit the bounds of available time and memory.

This book presents recent advances in formal verification techniques and discusses the applicability of the techniques in ensuring the reliability of large-scale systems. We deal with the verification of a range of computing systems, from sequential programs to concurrent protocols and pipelined machines. Each problem is different, bringing in unique challenges for verification. We show how to ameliorate verification complexity by applying formal analysis judiciously within a hierarchical, disciplined reasoning framework.

The research documented here originally appeared as the author's Ph.D. dissertation. The research owes its existence more to skepticism about the possibility of mathematical analysis at this scale than to anything else. Coming from a background in theory of computation and armed with traditional results in computability and complexity theory, I approached the area with extreme suspicion: how can techniques that are so severely handicapped with undecidability, incompleteness, and (in the better scenarios) lower bounds of computational complexity be successful in practice? The answer to the paradox that I found in my own research is that formal reasoning should not be viewed as a stand-alone tool but rather an integral part of the system design process. In particular, formal reasoning thrives when

it serves to make explicit the intuitions already available to the system designers, helping them effectively articulate the informal justifications behind optimizations and design tweaks. A well-designed reasoning framework permits specification of a problem at a level that matches the intuition of the designer and this turns the analysis into a vehicle for making the intuitive justifications rigorous. Intuitive justification and mathematical rigor must go hand-in-hand, each clearing the way of the other.

Most treatises and research articles on formal verification do not focus on the role of analysis as part of the system design process. Indeed, verification research today is essentially single tracked: the mantra is to develop algorithms and heuristics for providing more automation in the analysis of larger and larger systems. Automation is certainly critical to large-scale system verification. However, with design complexity is increasing faster than automation heuristics can catch up, improvement in raw automation alone is insufficient: one must additionally have disciplined, hierarchical, and customized reasoning infrastructures for different domains. Such infrastructures must permit (1) decomposition of a complex problem into tractable pieces, (2) effective management and control of the overall verification and decomposition, and (3) the ability of different, customized, automated tools to bear upon each individual piece. Furthermore, the act of developing a reusable framework forces thinking about a verification problem at an appropriate level of abstraction. Note that an "abstraction" *does not* mean a "toy." Rather, the abstraction must capture the essence of the class of systems being analyzed (often by generalization), while eliminating irrelevant complexities. Admittedly, developing a generic, compositional framework for a specific target domain is not easy. Thinking about a problem in a generic manner requires creativity, experience, and practice. However, the cost of doing so is ameliorated by the fact that the same framework can be used over and over on different problems. I have found the act of developing careful, generic frameworks for different problem domains to be both instructive and fruitful, as it exposes the underlying structure of the problems, helps spot connections between various subproblems, and thereby leads to the identification of the verification strategy most suited for the problem at hand. The research presented in this book shows numerous examples of generic verification strategies for different problem domains. The infrastructures described here are intended to serve as a guide to formal verification researchers and practitioners, who intend to pursue mathematical reasoning for large-scale systems. The book is also intended to show system designers how mathematical reasoning can serve as a helpful practical tool for ensuring reliable execution of the systems they design.

This research owes its existence to a number of people, including my teachers, collaborators, friends, and family. Perhaps the biggest debt goes to my Ph.D. supervisor J. Strother Moore for giving me the right balance of freedom, encouragement, and direction to guide the course of this research. Some other key players are (in alphabetical order) Jeff Golden, Warren A. Hunt, Jr., Matt Kaufmann, John Matthews, Erik Reeber, Fei Xie, and Thomas Wahl.

The research presented in this monograph has been supported over the years in part by funding from the National Science Foundation under Grants ISS-041741,

CCF-0916772, and CNS-0910913, by the Defense Advanced Research Projects Agency and National Science Foundation under Grant CNS-0429591, and by the Semiconductor Research Corporation under Grants 02-TJ-1032 and 08-TJ-1849. I am grateful to these agencies for the support. However, any opinions, findings, conclusions, or recommendations expressed herein are mine and do not necessarily reflect the views of any of these agencies.

Finally, I thank my wife Anindita for bearing with me through all the late hours that went behind the book. This book would not have seen the light of the day without her understanding and support.

Austin, TX Sandip Ray
November 2009

Contents

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, therefore, crucial to ensure that computing systems behave correctly and reliably. The 1999 PITAC[1] report to the President of the United States of America succinctly explains how the well-being of our society is tightly intertwined with reliability of computing systems [206]

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

The correct and reliable behavior of a system depends on the correct behavior of the hardware and software used to implement the system. Ensuring correct behavior of a modern computing system implementation, 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. Today, a modern microcontroller laid out in a small silicon die has, and needs to have, computing power several times that of a large supercomputer of 30 years back. Implementations of such systems involve megabytes of program code, and even a description of their desired properties, if written down, goes to hundreds of pages. Given such complexity, it is not surprising that modern computing systems 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 can lead to a serious error. The currently practiced methods for ensuring reliable executions of most system implementations 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

[1] President's Information Technology Advisory Committee

S. Ray, *Scalable Techniques for Formal Verification*, DOI 10.1007/978-1-4419-5998-0_1,

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 a mathematical logic and formally proves that it satisfies its desired specifications. Formal verification in practice makes use of a mechanical reasoning tool, *i.e.*, 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 recent successes in proving the correctness of industrial-scale hardware and software systems. Formal verification is now part of the tool flow in microprocessor companies like AMD, IBM, and Intel. For example, many floating-point operations of the AMD K5™ [179], Athlon™ [223] and Opteron™, and the Intel Pentium® Pro [191] microprocessors have been formally proven to be IEEE compliant. However, in spite of these 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 book, we identify some of the difficulties in 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 (1) deductive verification or theorem proving [88, 100, 122, 189, 194] and (2) algorithmic decision procedures such as model checking [49, 207], equivalence checking [171], symbolic simulation [116], and symbolic trajectory evaluation [44]. The key technical difference between the deductive and algorithmic methods is in the expressiveness of the formal logic employed. Theorem provers use an expressive logic that 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, etc. However, theorem proving is not automatic in general, and its successful use for proving nontrivial theorems about complicated systems depends on significant interaction between the theorem prover and 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. On the other hand, in the algorithmic approach, 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 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 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 book, we present tools and techniques to facilitate the conditions given above. Since we wish to preserve the expressiveness, flexibility, and diversity of proof techniques afforded by deductive reasoning, theorem proving provides the foundations of the approach. However, we 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.

Carrying out the above program turns out to be more subtle than it may at first appear. In addition to the engineering challenges involved in effectively integrating different procedures, combination of different formal tools brings 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? The question is not merely academic, and we will see in Chap. 12 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 Chap. 5, 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.

The approach requires that we work inside a practical general-purpose theorem prover. We use ACL2 [121, 122] 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 computing systems that have ever undergone formal verification [32, 179, 223]. 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 [242]. 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 book. However, this book 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 the integration techniques presented here can be used or adopted for increasing the capacity of verification in other general-purpose theorem provers such as HOL [88] or PVS [194]. Our presentation 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 book consists of the following six parts:

- Preliminaries
- Sequential Program Verification
- Verification of Reactive Systems
- Invariant Proving
- Formal Integration of Decision Procedures
- Conclusion

Part I provides the foundational groundwork for the book. In Chap. 2, we review selected verification techniques, namely theorem proving, model checking, and verification condition generation. In Chap. 3, we describe the logic of the ACL2 theorem prover. These two chapters are intended to provide the necessary background to keep the book 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–V contain the chief technical contributions of the monograph. 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, nonterminating 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 nondeterminism 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* [90] of the system. Predicate abstractions are computed by *term rewriting* [9], 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 [28] and SMV [165].

In Part V, 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 obstacles that need to be overcome in order to make the integration effective. This research has led provided impetus in the development of a generic interface for connecting ACL2 with external deduction tools; we outline the architecture of this interface along with implementation complexities involved in the design of such an interface with a mature, industrial-strength theorem prover.

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 coauthors and the published version of the work wherever applicable.

The book contains several case studies. However, the focus here is on verification techniques, 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.

Part I
Preliminaries

Chapter 2
Overview of Formal Verification

Formal verification of a computing system entails a mathematical proof showing that the system satisfies its desired property or specification. To do this, we must use some mathematical structure to model the system of interest and derive its desired properties 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 will survey some of the key formal verification techniques and understand 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 directly relevant to the techniques discussed in the subsequent chapters. 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 is one of the key approaches to formal verification. A theorem prover is a computer program for constructing and checking derivations in a formal logic. A formal logic comprises the following three components.

1. A *formal language* to express formulas
2. A collection of formulas called *axioms*
3. A collection of *inference rules* for deriving new formulas from existing ones

To use formal logic for reasoning about a mathematical artifact, one considers a logic such that the formulas representing the axioms are *valid*, *i.e.*, can be interpreted as self-evident truths about the artifact, and the inference rules are validity preserving. Consequently, 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*). A sequence σ of formulas such that each member

S. Ray, *Scalable Techniques for Formal Verification*, DOI 10.1007/978-1-4419-5998-0_2,

of σ is either an axiom or obtained from a collection of previous members of σ by applying an inference rule is called a *derivation* or *deduction*. In the context of theorem proving, "verifying a formula" means showing that there is a deduction of the formula in the logic of the theorem prover.

There are many theorem provers in active use today. The popular ones include ACL2 [122], Coq [70], HOL [88], Isabelle [196], NuPrl [59], and PVS [194]. The underlying logics of theorem provers vary considerably. There are theorem provers for set theory, constructive type theory, first-order logic, higher order logic, etc. There is also substantial diversity in the amount of automation provided by the different theorem provers; some are essentially 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, ACL2. In spite of diversity of features, however, a common aspect of all theorem provers is that they support logics that 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 [220, 234], and Ramsey's Theorem and Cantor's Theorem have been proven in Isabelle [197, 198].

However, the 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 [81, 250]. That means that there is no automatic procedure (or algorithm) that, given a formula, can always determine if there exists a derivation of the formula in the logic. Thus, the successful use of theorem proving for deriving nontrivial theorems typically involves interaction with a trained user. Nevertheless, theorem provers are still invaluable tools for formal verification. Some of the things that a theorem prover can do 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 [187, 237], and ACL2 has procedures for deciding linear inequalities over rationals [24, 111].

In spite of success in approaches designed to automate the search of proofs, it must be admitted that undecidability poses an insurmountable barrier in developing automation. In practice, the construction of a nontrivial derivation via theorem proving is a creative process requiring substantial interaction between the theorem prover and a trained user. The user 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 monograph 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 nontrivial amount of discipline and attention to detail. Kaufmann and Moore [125] 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, *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 for proving such properties. Even when the properties can be expressed in a decidable logic, *e.g.*, 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 $\mathscr{S}_3$ of three 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 $\mathscr{S}_4$ with four processes, the verification of $\mathscr{S}_3$ does not give us any assurance in the correctness of this new system, and we must reverify it. In general, we would probably like to verify the implementation for a system $\mathscr{S}_n$ of n processes. However, the mutual exclusion property for such a parameterized system $\mathscr{S}_n$ 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 [71]. Also, even when decision procedures are applicable in principle (*e.g.*, for the system $\mathscr{S}_{100}$ 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 provide a substantial degree of control in the process of derivation of complex theorems.[1] This can be exploited by the user in different forms, *e.g.*, 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 [29, 31] 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 [200] 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 monograph, 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, is a key strength. A substantial amount of research in formal verification is aimed at designing decidable

[1] By theorem provers in this monograph, we normally mean the so-called *general-purpose* theorem provers. There are other more automated theorem provers, for instance Otter [162], 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.

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.

In addition to providing automation, the use of a decidable formalisms has one other significant benefit. When the system has a bug, that is, its desired properties are not theorems, 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 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 nonterminating 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 Chap. 13, 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 verification 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 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 [53] 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 Chap. 13 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 are 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 $\mathscr{A}\mathscr{P}$ called the set of *atomic propositions*, standard Boolean operators "$\wedge$," "$\vee$," and "$\neg$," and four *temporal operators* "X," "G," "F," and "U," as specified by the following definition.

Definition 2.1 (LTL Syntax). The syntax of an LTL formula is recursively defined as follows.

- If $\psi \in \mathscr{A}\mathscr{P}$ 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, specified below.

Definition 2.2 (Kripke Structure). A Kripke Structure κ is a tuple $\langle S, R, L, s_0 \rangle$, 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*, $L : S \to 2^{\mathscr{A}\mathscr{P}}$ is called the *state labeling function*, and $s_0 \in S$ is called the *initial state*.[2]

It is easy to model computing systems in terms of Kripke Structures. A system usually comprises 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 nondeterministic 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

[2] The initial state s_0 is dropped in some expositions on model checking.

sequence π, we will refer to the ith 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.

Definition 2.3 (LTL Semantics). The notion of a path π satisfying an LTL formula ψ is defined recursively as follows:

1. If $\psi \in \mathscr{AP}$ 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 interesting properties of reactive systems using LTL. Consider the mutual exclusion property for the three-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

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

Remark 2.1 (Conventions). Notice the use of "$\doteq$" above. When we write $A \doteq B$, we mean that A is a shorthand for writing B, and we use this notation throughout the monograph. In other treatises, one might often write this as $A = B$. But since much of the work in this monograph is based on a fixed formal logic, namely ACL2, we restrict the use of the symbol "=" to formulas in ACL2.

Clearly, 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; we need stronger atomic propositions that can specify quantification over 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*.[3] There are several interesting model checking algorithms and a complete treatment of such is beyond the scope of this monograph. We merely sketch one algorithm that is based on the construction of a Büchi automaton, since we will use properties of this algorithm in Chap. 13.

Definition 2.4 (Büchi Automaton). A Büchi automaton $\mathscr{A}$ 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*.[4]

Definition 2.5 (Language of Büchi Automaton). An infinite sequence of symbols from Σ constitutes a *word*. We say that $\mathscr{A}$ *accepts* a word σ if there exists an infinite sequence ρ of states of $\mathscr{A}$ with the following properties:

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

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

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 $\mathscr{A}_\kappa$ for a Kripke Structure $\kappa \doteq \langle S, R, L, s_0 \rangle$, such that every word σ accepted by $\mathscr{A}_\kappa$ corresponds to paths in κ. That is, for every word σ accepted by $\mathscr{S}$, $\sigma^i \subseteq \mathscr{AP}$ and there is a path π in κ such that σ^i is equal to $L(\pi^i)$.

- The set of states of $\mathscr{A}$ is the set S. Each state is an accepting state.
- $\Sigma \doteq 2^{\mathscr{AP}}$ where $\mathscr{AP}$ 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 $\mathscr{A}_\psi$ such that every word accepted by this automaton satisfies ψ. This construction, often referred to as *tableau construction*, is complicated but well known [51, 53]. Checking if κ satisfies ψ now reduces to checking $\mathscr{L}(\mathscr{A}_\kappa) \subseteq \mathscr{L}(\mathscr{A}_\psi)$. 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 $\mathscr{A}$, there is an algorithm to check if $\mathscr{L}(\mathscr{A})$ is empty. Thus, we can check the language containment question above as follows. Create a Büchi automaton $\mathscr{A}_{\kappa,\psi}$ such that $\mathscr{L}(\mathscr{A}_{\kappa,\psi}) \doteq \mathscr{L}(\mathscr{A}_\kappa) \cap \overline{\mathscr{L}(\mathscr{A}_\psi)}$. Then check if $\mathscr{L}(\mathscr{A}_{\kappa,\psi})$ is empty.

[3] 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 monograph.

[4] 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, *e.g.*, the executions 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.

No practical model checker actually constructs the above automaton and checks emptiness explicitly. However, the construction suggests a key computational bottleneck of model checking. 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 key approach is to efficiently represent sets of states using BDDs [35]: this 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 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 Chap. 13. Modern model checkers perform a host of reductions to make the model checking problem tractable. They include identification of *symmetry* [50], reductions based on partial order [119], assume-guarantee reasoning [203], and many others. In addition, one popular approach is based on iterative refinement of abstractions based on counterexamples [52], often called "Counter Example Guided Abstraction Refinement" (CEGAR for short). Since model checking is a decision procedure, if a Kripke Structure does not satisfy a formula, then the procedure can return a counterexample (*i.e.*, a path through the Kripke Structure that does not satisfy the formula). In CEGAR, one starts with a Kripke Structure $\hat{\kappa}$ such that every execution of κ can be appropriately viewed 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 model checking succeeds, then κ must satisfy ψ as well. If 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 (a) the counterexample provided by model checking the abstraction is also a counterexample on the "real" system, that is, κ or (b) one finds an abstraction such that the model checking succeeds.

2.3 Program Logics, Axiomatic Semantics, and Verification Conditions

The use of decision procedures is one approach to scaling up formal verification. The idea is to automate verification by expressing the problem in a decidable formalism. The use of *program logics* is 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 formalize the executions in terms of *states*.

The states are valuations of the different components of the *machine* executing the program. Thus, 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 are called *operational semantics* [161].[5] 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. 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 program logics [67, 103] 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 $\{\mathscr{P}\}I\{\mathscr{Q}\}$, where $\mathscr{P}$ and $\mathscr{Q}$ are (first order) predicates over the program variables. Such a formula can be read as: "If $\mathscr{P}$ holds for the state of the machine when the program is poised to execute I then, after the execution of I, $\mathscr{Q}$ holds." Predicates $\mathscr{P}$ and $\mathscr{Q}$ 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, also known as the "axiom of assignment."

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

We can use this schema to derive, for example, $\{\mathtt{a} > 1\}\mathtt{x} := \mathtt{a}\{\mathtt{x} > 1\}$ which is interpreted as saying that if the machine is in a state s in which the value of `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, as well as its application, involve syntactic manipulation of the program constructs without requiring any insight into the operational details of the machine.

Hoare [103] provides five 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*.

[5] We use "operational semantics" to denote the semantics derived from the definition of a formal interpreter in a mathematical logic. This terminology is common in program verification, in particular in mechanical theorem proving. However, this usage is somewhat dated among programming language communities, where "operational semantics" now refers to Plotkin's "Structural Operational Semantics" (SOS) [201], a style of specifying language semantics through inductively defined relations and functions over the program syntax. In current programming language terminology, our approach might more appropriately be referred to as "abstract machine semantics."

Rule of Composition.
Infer $\{\mathscr{P}\}\langle i_1; i_2\rangle\{\mathscr{Q}\}$ from $\{\mathscr{P}\}i_1\{\mathscr{R}\}$ and $\{\mathscr{R}\}i_2\{\mathscr{Q}\}$

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

Rule of Implication.
Infer $\{\mathscr{P}\}i\{\mathscr{Q}\}$ from $\{\mathscr{R}_1\}i\{\mathscr{R}_2\}$, $\mathscr{P} \Rightarrow \mathscr{R}_1$, and $\mathscr{R}_2 \Rightarrow \mathscr{Q}$

Here, "$\Rightarrow$" is simply logical implication. First-order logic, together with the Hoare axioms and inference rules, forms a proof system in which we can talk about the correctness of programs. This logic is referred to as a *Hoare logic*. Hoare logic is an instance of *program logic*, *i.e.*, logic tailored for reasoning about program correctness. In particular, Hoare logic is the logic for reasoning about programs with loops (but no pointers). To use Hoare logic for reasoning about programs, one maps assertions to predicates on program variables, and views program instructions as transformations of such predicates. From the examples above, it should be clear that one can capture the semantics of a programming language in terms of how predicates change during the execution of the program instructions. The semantics of a programming language specified by describing the effect executing instructions on assertions (or predicates) about states (rather than states themselves) is known as *axiomatic semantics*. Thus, Hoare logic (or in general a program logic) can be viewed as a proof system over the axiomatic semantics of the programming language.

Suppose we are given a program Π and we want to prove that if the program starts from some state satisfying $\mathscr{P}$, then the state reached on termination satisfies $\mathscr{Q}$. This can be succinctly written in Hoare logic as the statement $\{\mathscr{P}\}\Pi\{\mathscr{Q}\}$. $\mathscr{P}$ and $\mathscr{Q}$ are called the precondition and postcondition *of the program*. One then derives this formula as a theorem.

How do we use Hoare logic for program verification? One annotates the program with assertions at certain locations (*i.e.*, for 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 with a simple one-loop program shown in Fig. 2.1. The program consists of two variables `X` and `Y`, and simply loops ten 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

```
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}
```

Fig. 2.1 A simple one-loop program

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 $1 \rightarrow 3$, identifying the beginning and ending values of the program counter along the execution of the basic block. Then we must prove the following formula as a theorem.

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

Applying the axioms of assignment, composition, and implication above, we can now derive the following simplified obligation.

Simplified Proof Obligation.
$T \Rightarrow (0 + 10) = 10$

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

Note that we have added an assertion at the loop test in addition to the precondition and postcondition. 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 verification conditions above only guarantee *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. For 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 are augmented so as to be able to reason about such ranking functions.

In practice, the verification conditions may be complicated formulas and their proofs might not be trivial. Practical applications of program verification based on axiomatic semantics depend on two tools: (1) a *verification condition generator* (VCG) that takes an annotated program and generates the verification conditions and (2) 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.

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 separate axiomatic semantics (and implement VCG) for each 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. For example, early versions of *separation logic* (*i.e.*, an augmentation of Hoare logic to permit reasoning about pointers) were later found to be unsound [216]. In addition, implementing a VCG for a practical programming language is a substantial enterprise. For example, method invocation in a language like the Java Virtual Machine (JVM) involves complicated nonsyntactic 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 underlying language, 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 [80, 106]. 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 [11, 66, 101, 185]. 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 [133] and Gupta [94]. In addition, a detailed overview on model checking techniques is presented in a book on the subject by Clarke et al. [53].

Theorem proving was started arguably with the pioneering *Logic Theorist System* of Newell et al. [188]. One of the key advocates of using theorem proving for verification of computing systems was McCarthy [161] who wrote: "Instead of debugging programs one should prove that it meets its specification 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 [218] 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, in 1999, 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 [163]. In the context of verification of computing systems, however, nonresolution theorem provers have found more applications. Some of the early work on nonresolution theorem proving were done by Wang [255], Bledsoe [16], and Boyer and Moore [21]. 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 discussed in this monograph. 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 HOL [88], HOL Light [100], Isabelle [189], Nuprl [59], and PVS [194].

The idea of using temporal logics for specifying properties of reactive systems was suggested by Pnueli [202]. Model checking was discovered independently by Clarke and Emerson [49] and Queille and Sifakis [207]. It is one of the most widely used formal verification techniques used in the industry today. Some of the important model checkers include SMV [165], VIS [28], NuSMV [47], SPIN [105], and Murϕ [69]. Most model checkers in practice include several reductions and optimizations [50, 119, 203]. Model checkers have achieved amazing results on industrial problems. While other decision procedures such as *symbolic trajectory evaluation* [44] and its generalization [259] 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 [233].

The notion of axiomatic semantics was made explicit in a classic paper by Floyd [77], although the idea of attaching assertions to program points appears much earlier, for example, in the work of Goldstein and von Neumann [84] and Turing [251]. Program logics were introduced by Hoare [103], Dijkstra [67], and Manna [149]. Assertional reasoning was extended to concurrent programs by Owicki and Gries [193]. Recently, separation logic has been introduced augmenting Hoare logic to reason about languages with pointers [217]. King [134] 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 [66], proof carrying code [185], and SLAM [11].

Chapter 3
Introduction to ACL2

The name "ACL2" stands for *A Computational Logic for Applicative Common Lisp*. It is the name for (1) a programming language based on a subset of Common Lisp, (2) a logic, and (3) 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 monograph 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. In this chapter, we present the logic of ACL2 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 facilitate the understanding of the logic, we discuss its connections with traditional first-order logic. Readers interested in a more thorough understanding of ACL2 are referred to the ACL2 home page [124]. In addition, we list several books and papers about ACL2 in Sect. 3.8.

3.1 Basic Logic of ACL2

Recall from Chap. 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

S. Ray, *Scalable Techniques for Formal Verification*, DOI 10.1007/978-1-4419-5998-0_3,

as $(f\ \ x_1\ \ \ldots\ \ x_n)$ instead of the more traditional $f(x_1, \ldots, x_n)$. However, for this monograph, 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 monograph.
- A string is represented as a sequence enclosed within double quotation marks, such as `"king"`, `"queen"`, `"Alice"`, etc. Thus, the first character of the string `"king"` is the *character* `#\k`.
- ACL2 has a complicated mechanism for representing constant symbols, which 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 monograph, we will use the notation $\langle x, y, z\rangle$ to denote such tuples.

Remark 3.1. We have not talked about the syntax of variables. Throughout this monograph we will talk about terms (and formulas), which include constant, variable, and function symbols. In any formal term that we write, any symbol that is not a constant symbol (according to the description above) or a function symbol (which is 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 "$\neg$," as specified by the following definitions.

Definition 3.1 (Formula). If τ_1 and τ_2 are terms, then $(\tau_1 = \tau_2)$ is an *atomic formula*. A formula is then defined by recursion 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, the latter 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 $v(\tau)$. 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 $v(\tau) \cap dom(\sigma)$ 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.

Remark 3.2 (Conventions). As is customary given these axiom schemas, abbreviations, and inference rules, we will interpret the operators "$\vee$," "$\wedge$," "$\neg$," "$\Rightarrow$," "$\leftrightarrow$," and "$=$" as disjunction, conjunction, negation, implication, equivalence, and equality, respectively.

In addition to the above rules, 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

From the description so far, 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 is 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. Here is an axiom that relates the functions *car* and *cons*.

Axiom.
$car(cons(x, y)) = x$

The axiom can be interpreted as: "For any x and y, the function *car*, when applied to *cons* of x and y, returns x." Observe 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**, *e.g.*, instantiation of the above axiom permits us to prove the theorem $car(cons(2, 3)) = 2$.

GZ characterizes about 200 functions described in the Common Lisp Manual [242], which are (1) free from side effects, (2) independent of the state or other implicit parameters or data types other than those supported by ACL2, and (3) unambiguously specified in a host-independent manner. There are axioms for arithmetic functions and functions for manipulating strings, characters, and lists. A description of all the axioms in GZ is beyond our scope; Fig. 3.1 provides a list of some

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

Fig. 3.1 Some functions axiomatized in GZ

important functions together with a brief description of 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 of GZ have an important property, which we can call "evaluability." Informally, this 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 [*i.e.*, if $v(\tau)$ is empty]. Then, the property is that for any ground term τ expressible in GZ we can determine a constant c such that $(\tau = c)$ is a theorem. The constant c is called the *value* of 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 *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 *car*(x) returns NIL.[1]

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

Similarly, for arithmetic functions (*e.g.*, $+$, $-$, $<$, etc.), if one of the arguments is not numeric, the completion axioms allow us to interpret that argument to be 0.

Remark 3.3. Since the axioms of GZ agree with Common Lisp 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 [127]. This "execution capability" allows ACL2 to deal

[1] 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.

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 [91].

3.2.1 Terms, Formulas, Functions, and Predicates

Before proceeding further, we point out a duality between terms and formulas in ACL2. So far, 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, the intended formula is $\neg(\tau = \texttt{NIL})$. Indeed, a user of the theorem prover only writes terms and never writes formulas. Consequently, if we prove the term τ as a theorem, then we have an alternative interpretation of the theorem as follows: "For any substitution σ that maps each variable in $v(\tau)$ 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 Fig. 3.1. We show the formal built-in axioms below.

Axioms.
$x = y \Rightarrow \mathit{equal}(x, y) = \texttt{T}$
$x \neq y \Rightarrow \mathit{equal}(x, y) = \texttt{NIL}$

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.

Axioms.
$(x = \texttt{NIL}) \Rightarrow \mathit{if}(x, y, z) = z$
$\neg(x = \texttt{NIL}) \Rightarrow \mathit{if}(x, y, z) = y$

Using *if*, we can formalize "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 Fig. 3.1. Here, we show the axioms that allow such interpretation.

Axioms.
$\mathit{and}(x, y) = \mathit{if}(x, y, \texttt{NIL})$
$\mathit{or}(x, y) = \mathit{if}(x, x, y)$
$\mathit{not}(x) = \mathit{if}(x, \texttt{NILT})$
$\mathit{implies}(x, y) = \mathit{if}(x, \mathit{if}(y, \texttt{T}, \texttt{NIL}), \texttt{T})$
$\mathit{iff}(x, y) = \mathit{if}(x, \mathit{if}(y, \texttt{T}, \texttt{NIL}), \mathit{if}(y, \texttt{NIL}, \texttt{T}))$

Remark 3.4 (Conventions). We will use the more standard "(*if* x *then* y *else* z)" instead of "*if*(x,y,z)." Thus, we can write the axioms above equivalently as follows.

Axioms.
$x = \texttt{NIL} \Rightarrow (\textit{if}\ x\ \textit{then}\ y\ \textit{else}\ z) = z$
$x \neq \texttt{NIL} \Rightarrow (\textit{if}\ x\ \textit{then}\ y\ \textit{else}\ z) = y$

We will use the notation "*if* x *then* y *else* z" to describe if-then-else constructs. However, in some cases, we will use the following notation to mean the same term.

$$\begin{cases} y & \textit{if}\ x \\ z & \text{otherwise} \end{cases}$$

This latter notation is useful when we have long nests of if-then-else structures.

We also make use of **let** for abbreviating large terms. Thus, we will write "**let** $x \leftarrow (y - z)$ **in** τ" as a shorthand for the term obtained by replacing each occurrence of x in τ by the term $(y - z)$.

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.

Completion Axiom.
implies(*not*(*or*(*consp*(x),*equal*(x,NIL))),*equal*(*car*(x),NIL))

In this monograph, we will find it convenient to use both terms and formulas depending on context. Thus, when we want to think about *equal*(τ_1, τ_2) as a formula, we 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 [236], 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 [40–42]. 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.

Definition 3.2 (Well-Founded Structure). 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 $w_i \in W$.

Note that 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 $0 < 1 < \cdots < \omega$. The "number" ω is called the first infinite ordinal. We add an infinite number of such "numbers" using a positional ω-based notation namely, $\omega+1, \omega+2, \omega\times 2, \omega^2, \omega^\omega$, 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 monograph whenever we talk about an ordinal we mean a member of this set.

How do we formalize 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 ω and larger are represented as ordered pairs. We show some examples of ordinals and their representations as constants in Fig. 3.2. It is not necessary 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 "$\prec_o$,"[2] 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.

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

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

Fig. 3.2 Examples of ordinal representation in ACL2

- 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: $(\neg\mathscr{C}_1 \wedge \ldots \wedge \neg\mathscr{C}_k) \Rightarrow \Phi$, and
Induction Step: $(\mathscr{C}_i \wedge \Phi/\sigma_{i1} \wedge \ldots \wedge \Phi/\sigma_{ik_i}) \Rightarrow \Phi$ for each $1 \leq i \leq k$.

If there exists some term m such that the following are theorems.

1. $o\text{-}p(m)$
2. $\mathscr{C}_i \Rightarrow m/\sigma_{ik} \prec_o m$, for each $1 \leq i \leq k$, $1 \leq j \leq k_i$.

Remark 3.5 (Conventions). We refer to the substitutions $\sigma_{i,j}$ in the definition of **Induction Rule** as *induction substitutions* (or, when the context is clear, simply *substitutions*), the $\mathscr{C}_i$ terms as *induction conditions*, the term m as the *induction measure* (or simply, *measure*), and **M1** and **M2** as *induction obligations for* m. We refer to a set of pairs $\{\langle \mathscr{C}_i, \sigma_i \rangle : 1 \leq i \leq k, \}$, where $\sigma_i \triangleq \{\sigma_{i,j} : 1 \leq j \leq h_i\}$, as an *induction skeleton*; if $h_i = 1$, then we may abuse notation and write σ_i to denote the substitution $\sigma_{i,1}$. Given a formula φ and an induction skeleton $\mathscr{I}$, we refer to the formulas **Base Case** and **Induction Step** as the *results of applying* $\mathscr{I}$ *on* φ.

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 $(\neg\mathscr{C}_1 \wedge \ldots \wedge \neg\mathscr{C}_k)$ holds and (2) Φ holds whenever $\mathscr{C}_i$ 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 "$\prec_o$" then guarantees that the sequence cannot decrease indefinitely, justifying the rule.

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

Well-Foundedness Requirements:
$o\text{-}p_\prec(x) \Rightarrow o\text{-}p(E_\prec(x))$
$o\text{-}p_\prec(x) \wedge o\text{-}p_\prec(y) \wedge (x \prec y) \Rightarrow (E(x) \prec_o E(y))$

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

ACL2's **Induction Rule** is a formulation of the well-known ε_0-induction [219] but with one subtle difference. Traditionally, induction is formulated as a higher order inference rule, since it involves quantification over function. We justify the rule in ACL2's first-order context by assuming that every ACL2 theory is *inductively complete*. We will discuss theories in Sect. 3.3 in the context of extension principles, but roughly, a theory includes a collection of axioms about specific function symbols. Thus, GZ is a theory. The function symbols individually axiomatized in a theory $\mathscr{T}$ are said to have been *introduced* in $\mathscr{T}$, and the individual axioms are often called *nonlogical axioms* of $\mathscr{T}$. The *language* of a theory $\mathscr{T}$ is the set of function symbols introduced in $\mathscr{T}$. A term (or formula) is *expressible* in $\mathscr{T}$ if and only if all function symbols in the formula have been introduced in $\mathscr{T}$. We already talked about terms and formulas expressible in GZ. We now define the notion of an inductively complete theory, which is used as a foundation for the **Induction Rule**.

Definition 3.3 (Inductively Complete Theory). A theory $\mathscr{T}$ is *inductively complete* if it includes as axiom the following formula, for each formula (or term) φ expressible in $\mathscr{T}$.

Axiom.
$(\forall y \prec_o \varepsilon_0)[((\forall x \prec_o y)\varphi/[x \to y]) \Rightarrow \varphi(y)] \Rightarrow (\forall y \prec_o \varepsilon_0)\, \varphi\, (y)$

The axiom is referred to as an *induction axiom*.

Any nontrivial inductively complete theory, of course, contains an infinite number of induction axioms. Thus, we never write down all the induction axioms, but we implicitly assume their existence, *i.e.*, when we prove a formula as a theorem of a theory we mean that it is a logical consequence of all the axioms including the induction axioms. Each theory that we construct in ACL2 will be inductively complete, as follows. GZ is implicitly inductively complete, and when we talk below about extending the theory $\mathscr{T}$ by a new nonlogical axiom $\mathscr{A}$, the new theory $\mathscr{T}'$ will be taken to be the theory that includes $\mathscr{A}$ together with any induction axioms expressible in $\mathscr{T}'$.

The following theorem then justifies that for any ACL2 theory $\mathscr{T}$, the **Induction Rule** is sound in $\mathscr{T}$, *i.e.*, any theorem derived using the rule is a first-order logical consequence of the axioms of $\mathscr{T}$. The proof is left as a straightforward exercise.

Theorem 3.1 (Soundness of Induction). *Let $\mathscr{T}$ be an inductively complete first-order theory, φ be a formula in the language of $\mathscr{T}$, and $\mathscr{I}$ be a legal induction skeleton in the language of $\mathscr{T}$ that is justifiable in $\mathscr{T}$. If the results of applying $\mathscr{I}$ on φ are theorems of $\mathscr{T}$ then so is φ.*

Finally, it is instructive to compare ACL2's induction principle with traditional ε_0-induction. Traditionally, ε_0-induction is specified as follows:

Traditional ε_0-Induction Rule.
Infer $(\forall y \prec \varepsilon_0)\varphi(y)$ from $(\forall y \prec \varepsilon_0)(((\forall x \prec y)\varphi(x)) \Rightarrow \varphi(y))$

This formulation allows us to assume *all* the smaller instances $\varphi(x)$ in the inductive hypothesis while proving $\varphi(y)$ by induction. On the other hand, the ACL2 formulation only allows a fixed (finite) set of instances of φ as inductive hypotheses. Nevertheless, the ACL2 formulation, even if further restricted to allow only *one* instance of the inductive hypothesis, is sufficient to carry out any proof that can be done with the traditional formulation. We return to this point in Sect. 3.3 when discussing the defchoose principle (cf. Sect. 3.3.3).

3.3 Extension Principles

GZ axiomatizes many Lisp functions. One can prove formulas expressing properties of such functions as theorems. 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 extension principles, we will briefly discuss the notion of theories as promised above. Consider a proof system with the first-order axiom schema (*e.g.*, **Propositional Axiom**, **Identity Axiom**, and **Equality Axiom**) and inference rules, together with as collection $\mathscr{S}$ of axioms specifying individual properties of some of the functions. Then a *first-order theory*, or simply *theory* $\mathscr{T}$, is the set of first-order consequences of $\mathscr{S}$. The set $\mathscr{S}$ itself is, of course, a subset of $\mathscr{T}$ and is referred to as the set of *nonlogical axioms* of $\mathscr{T}$. We say that theory $\mathscr{T}'$ *extends* $\mathscr{T}$ if and only if all the nonlogical axioms of $\mathscr{T}$ are also nonlogical axioms of $\mathscr{T}'$.

The extension principles of ACL2 (discussed below) allow us to extend a theory by new nonlogical axioms. The nonlogical axioms can extend the theory with new function symbols; then, we say that the function symbols are *introduced* by the extension. A theory $\mathscr{T}$ 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 $\mathscr{T}$ in this monograph, we always mean a legal theory.

The 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 monograph. In addition to these, ACL2 provides another extension principle called *defaxiom* for stipulating an arbitrary formula (expressible in the language of the current theory) as an axiom. The use of defaxioms is discouraged because of the obvious risk of inconsistency of the resultant theory; we will avoid the use of defaxioms for the most part in this monograph. The application of an extension principle is called an *axiomatic event*. More generally, *events* are actions that extend ACL2's logical database and include axiomatic events and proven theorems.

3.3.1 Definitional Principle

The *definitional principle* is used to extend an ACL2 theory with *definitional axioms* that axiomatize new total functions.

Definition 3.4 (Definitional Axiom). A *definitional axiom* D is a finite set of equations, where the left-hand side of the ith equation is the application of some function symbol f_i of some arity n_i to a list V_i of n_i distinct variables (called *formal parameters of* f_i), and the set of free variables of the right-hand side is a subset of V_i. We say that D *axiomatizes* or *introduces* the set of function symbols occurring in the left-hand sides.

Mutually Exclusive Definitional Axioms.
odd(x) = *if zp*(x) *then* NIL *else even*$(x-1)$
even(x) = *if zp*(x) *then* T *else odd*$(x-1)$

Fig. 3.3 Example of a mutually recursive definition

As an application of the definitional principle, 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. The axiom is called the *definitional axiom* (or simply *definition*) of *mfact*.

Definitional Axiom.
mfact(x) = *if zp*(x) *then* 1 *else* $(x \times$ *mfact*$(x-1))$

The definitional principle can also introduce mutually recursive definitions. For example, we can define two functions *odd* and *even* as shown in Fig. 3.3.

In general, given a theory $\mathscr{T}$, the definitional principle is used to extend $\mathscr{T}$ by adding axioms of the form $f_i(x_{i1},\ldots,x_{in_i}) = \tau_i$, $i = 1,\ldots,l$. 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.

Definition 3.5 (Admissibility Requirements). Let $\mathscr{T}$ be a theory and D be a definitional axiom. Let $\mathscr{T}$ be the theory obtained by extending $\mathscr{T}$ with the function symbols introduced by D but no axioms. Then, D is admissible if the following conditions are satisfied.

- The function symbols introduced by D are not in the language of $\mathscr{T}$.
- The right-hand side of each equation in D is expressible in the theory $\mathscr{T}'$.
- Certain *measure conjectures* (cf. Definition 3.7) are theorems of theory $\mathscr{T}'$.

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. In particular, one exhibits a term m (called the *measure term*) and shows that (1) m returns a member of a well-founded structure and (2) m decreases at every recursive call. We will discuss these proof obligations more precisely below. For the function *mfact*, one possible measure term is the term *nfix*(x). In ACL2's Lisp syntax, one can introduce the definition of *mfact* with the following axiomatic event.

```
(defun mfact (x)
  (declare (xargs :measure (nfix x)))
  (if (zp x)
      1
    (* x (mfact (- x 1)))))
```

The optional `declare` construct provides directives to the theorem prover. The `xargs` directive, which stands for "extra arguments," is used (in this case) to specify the measure for proving termination. Since no well-founded structure is explicitly provided, ACL2 will use the default well-founded structure $\langle o\text{-}p, \prec_o \rangle$; it is possible for the user to explicitly provide the well-founded structure. Also, it is possible to omit the measure; in that case, ACL2 would attempt to admit the definition using a measure produced by a built-in strategy (which would also have succeeded in this case). However, in practice, the user may have to provide the measure term and the well-founded structure to be used to prove the measure conjectures.

We now discuss the measure conjectures more precisely. In general, the measure conjectures are specified in terms of the *governors* of recursive calls, which are defined as follows.

Definition 3.6 (Governors). Let u be a term and let t be a subterm occurrence in u. The set $G(t,u)$ of governors of t in u is defined recursively as follows.

1. If t is u then $G(t,u) \triangleq \emptyset$.
2. If u is of the form "*if* u_1 *then* u_2 *else* u_3" and the occurrence t is in u_2 (resp., u_3), then $G(t,u) \triangleq \{u_1\} \cup G(t,u_2)$ (resp., $G(t,u) \triangleq \{\neg u_1\} \cup G(t,u_3)$).
3. Otherwise, let t occur in the argument u' of u; then $G(t,u) \triangleq G(t,u')$.

In the definition of *mfact*, the only governor for the call $\mathit{mfact}(x-1)$ is $\neg \mathit{zp}(x)$, and in the definition of *even*, the only governor for the call $\mathit{odd}(x-1)$ is $\neg \mathit{zp}(x)$. ACL2 treats a specific subset of the governors as *rulers* (see below). The governors for each of the calls above will be treated as rulers by ACL2. Definition 3.7 specifies how the rulers are used in measure conjectures.

Definition 3.7 (Measure Conjectures). Let $f_1, \ldots, f_l$ be the set of function symbols introduced by some definitional axiom D, where the body of f_i contains k_i (mutually) recursive calls. Then, admitting the definition using measures $m_1, \ldots, m_l$ generates $(l + \sum_{i=1}^{l} k_i)$ *measure conjectures*. The first l proof obligations are the formulas $\mathit{o\text{-}p}(m_i)$ for $1 \leq i \leq l$. Additionally, there are k_i proof obligations associated with each f_i, one for each recursive call. Let the jth recursive call in the definition of f_i be an invocation of f_t, let the associated substitution mapping the formals of f_t to the actuals of the call be $\sigma_{i,j}$, and let $\gamma_{i,j}$ be the conjunction of rulers for the call. Then, the proof obligation associated with the call is $\gamma_{i,j} \Rightarrow m_t/\sigma_{i,j} \prec m_i$.

How does the definitional principle work with our example *mfact*? Given the measure term $\mathit{nfix}(x)$ above, the following are the measure conjectures, which are easy to prove in GZ. Recall (Fig. 3.1) that $\mathit{nfix}(x)$ returns x if x is a natural number, else 0; therefore, $\mathit{nfix}(x)$ always returns a natural number (and hence an ordinal).

Measure Conjectures for *mfact*:
$\mathit{natp}(\mathit{nfix}(x))$
$\neg \mathit{zp}(x) \Rightarrow \mathit{nfix}(x-1) < \mathit{nfix}(x)$

There is a strong 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:

Theorem `mfact-is-natp`
$natp(mfact(x)) = \texttt{T}$

To prove this, we will use induction as follows:

Base Case: $zp(x) \Rightarrow natp(mfact(x))$
Induction Step: $(\neg zp(x) \wedge natp(mfact(x-1))) \Rightarrow natp(mfact(x))$

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 $mfact(x)$. We will thus refer to this proof as *proof by induction based on* $mfact(x)$. Throughout this monograph, when we talk about proofs by induction we will present the term on which the induction is based.

Remark 3.6. 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 $mfact(6)$ is 120 and the value of $even(3)$ is `NIL`. Perhaps surprisingly, the axioms also specify that the value of $mfact(\texttt{T})$ is 1, and the value of $even(1/2)$ is `T`.

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

Definitional Axiom.
$step(s) = s + 1$

The definition is not recursive and therefore does not produce measure obligation. 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.

Theorem `two-step-theorem`
$\mathit{step}(\mathit{step}(s)) = s + 2$

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 $\mathit{step}(\mathit{step}(s))$ to $\mathit{step}(s + 1)$, and do this again to simplify this term to $s + 2$. This way of simplifying a term that 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.

To illustrate encapsulation, consider introducing three functions, *ac*, *ac-id*, and *ac-p*, axiomatized satisfy the following five constraints.

Encapsulation Axioms.
$\mathit{ac}(\mathit{ac}(x, y), z) = \mathit{ac}(x, \mathit{ac}(y, z))$
$\mathit{ac}(x, y) = \mathit{ac}(y, x)$
$\mathit{ac\text{-}p}(\mathit{ac}(x, y)) = \mathtt{T}$
$\mathit{ac\text{-}p}(\mathit{ac\text{-}id}(\,)) = \mathtt{T}$
$\mathit{ac\text{-}p}(x) \Rightarrow \mathit{ac}(\mathit{ac\text{-}id}(), x) = x$

The axioms stipulate that *ac* is an associative–commutative function always returning an object in the domain recognized by the predicate *ac-p*, and *ac-id* is an identity over that domain. Figure 3.4 shows an ACL2 event introducing the constrained functions above. The event begins by declaring function symbols introduced by the encapsulation. With each declaration is associated a *signature*: the signature for *ac* specifies that it take two arguments and returns a single value.[3] To ensure consistency, the user must exhibit functions (called the *local witnesses*) that satisfy the alleged axioms. The definition events marked *local* direct ACL2 to use the following functions as local witnesses.

[3] ACL2 also supports *multiple* (vector) *values* and *single-threaded objects* [26] and the signature specifies if any of the arguments or return values of a function is a single-threaded object. We ignore these issues in the paper, since they are not relevant to logical foundations.

```
(encapsulate
  (((ac * *) => *)
   ((ac-id) => *)
   ((ac-p *) => *))
  (local (defun ac (x y) (declare (ignore x y)) 42))
  (local (defun ac-p (x) (if (equal x 42) T NIL)))
  (local (defun ac-id () 42))
  (defthm ac-associative
    (equal (ac (ac x y) z) (ac x (ac y z))))
  (defthm ac-commutative
    (equal (ac x y) (ac y x)))
  (defthm ac-recognized-by-ac-p
    (equal (ac-p (ac x y)) T))
  (defthm ac-id-is-ac-p
    (equal (ac-p (ac-id)) T))
  (defthm ac-id-identity
    (implies (ac-p x) (equal (ac (ac-id) x) x))))
```

Fig. 3.4 Event introducing *ac*, *ac-id*, and *ac-p*

$$\begin{aligned} \textit{ac}(x, y) &= 42 \\ \textit{ac-p}(x) &= \textit{if}\ (x = 42)\ \textit{then}\ \texttt{T}\ \textit{else}\ \texttt{NIL} \\ \textit{ac-id}() &= 42 \end{aligned}$$

The `defthm` events direct ACL2 to prove the desired constraints as theorems about the local witnesses. For this example, the proofs are trivial. In general, one might need additional lemmas and definitions; such auxiliary events are marked local, restricting their use to the scope of the encapsulation. After ACL2 proves these theorems, it *admits* the encapsulation, that is, extends the current theory with the function symbols declared in the signature constrained to satisfy the formulas specified as nonlocal theorems. Admissibility of an encapsulation event requires that none of the nonlocal events mentions any locally defined function symbols other than those in the signature.

Definition 3.8 (Encapsulation Admissibility). Given a theory $\mathscr{T}$, the introduction of a collection of function symbols $f_1, \ldots, f_k$ with axioms $\mathscr{C}_1, \ldots, \mathscr{C}_l$ 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 $\mathscr{T}$.
- It is possible to extend $\mathscr{T}$ with function symbols $f_{1w}, \ldots, f_{kw}$ so that the formulas $\mathscr{C}_{1w}, \ldots, \mathscr{C}_{lw}$ are theorems, where $\mathscr{C}_{iw}$ is obtained from $\mathscr{C}_i$ 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 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 functions introduced using encapsulation, the only axioms introduced are the constraints. Thus, any theorem about such functions is valid for any other set of

Definitional Axiom.

ac-list(l) = *if* ¬*consp*(l) *then* NIL
else if ¬*consp*(*cdr*(l)) *then car*(l)
else ac(*car*(l), *ac-list*(*cdr*(l)))

Fig. 3.5 Definition of *ac-list*

functions that also satisfy the constraints. This observation is encoded by a derived rule of inference called *functional instantiation* [19]. Let Φ be a formula in some theory $\mathscr{T}$, which refers to the constrained function symbols $f_1, \ldots, f_k$ that have been introduced in $\mathscr{T}$ with constraints $\mathscr{C}_1, \ldots, \mathscr{C}_l$. Let $g_1, \ldots, g_k$ be functions in $\mathscr{T}$ such that $\mathscr{C}_{1g}, \ldots, \mathscr{C}_{lg}$ are theorems, where $\mathscr{C}_{ig}$ is obtained from $\mathscr{C}_i$ by consistently replacing $f_1, \ldots, f_k$ with $g_1, \ldots, g_k$. Then, functional instantiation says that if Φ is provable in $\mathscr{T}$ then one can infer Φ_g, where Φ_g is obtained by replacing $f_1, \ldots, f_k$ in Φ with $g_1, \ldots, g_k$. To illustrate this, consider the function *ac-list* shown in Fig. 3.5, which returns the result of applying *ac* to a list of objects. It is straightforward to prove the theorem named `ac-reverse-relation`, which states that the repeated application of *ac* along the list l has the same effect as the repeated application of *ac* along the list obtained by reversing l. Here, the list reversing function, *reverse*, is axiomatized in GZ.

Theorem `ac-reverse-relation`
ac-list(*reverse*(l)) = *ac-list*(l)

In order to use functional instantiation, we note that the binary addition function "+" is associative and commutative and always returns a number, 0 is the left identity over numbers, and GZ has a predicate *acl2-numberp* that returns `T` if and only if its argument is a number. Thus, if we define the function *sum-list* that repeatedly adds all elements of a list l, then functional instantiation of the theorem `ac-reverse-relation` enables us to prove the following theorem under the functional substitution of "+" for *ac*, 0 for *ac-id*, *numberp* for *ac-p*, and *sum-list* for *ac-list*.

Theorem `sum-reverse-relation`
sum-list(*reverse*(l)) = *sum-list*(l)

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.

Axiom.
$\Phi(x, y) \Rightarrow \Phi(x, f(x))$

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 $\mathscr{T}$ 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 $\mathscr{T}$ by introducing predicates such as p below.

Defchoose Axiom.
$p(x_1, \ldots, x_n) = \exists y : \Phi$

Here, Φ must be a formula expressible in $\mathscr{T}$, and $\nu(\Phi)$ must be equal to the set $\{x_1, \ldots, x_n, y\}$.

Remark 3.7. When a predicate p is admissible in the current theory with the **Defchoose Axiom** as above, 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 $\neg(\exists x : \neg\Phi)$.

We conclude the discussion of quantification in ACL2 by discussing the point that we postponed during the discussion of ACL2's induction principle in Sect. 3.2.2, *e.g.*, the relation between ACL2's induction principle and traditional ε_0-induction. Recall that the traditional formulation allows us to assume *all* the smaller instances $\varphi(x)$ in the inductive hypothesis while proving $\varphi(y)$ by induction. On the other hand, the ACL2 formulation only allows a fixed (finite) set of instances as inductive hypotheses. Nevertheless, the ACL2 formulation, even if further restricted to allow only *one* instance of the inductive hypothesis, is sufficient to carry out any proof that can be done with the traditional formulation.[4] To see this,

[4] Eric Smith has contributed a mechanical proof in ACL2 showing that if a formula φ is provable by standard ε_0-induction then it is also provable using an ACL2 induction scheme. This proof is distributed with ACL2 in the file `books/misc/transfinite.lisp` and essentially formalizes the argument here.

recall that ACL2 provides Hilbert's choice operator via the defchoose principle. The universally quantified induction hypothesis $(\forall x \prec y)\varphi(x)$ can be written as $c(y) \prec y \Rightarrow \varphi(c(y))$, where $c(y)$ is introduced via defchoose as follows.

Defchoose Axiom.
$(x \prec y \wedge \neg\varphi(x)) \Rightarrow (c(y) \prec y \wedge \neg\varphi(c(y)))$

3.4 The Theorem Prover

ACL2 is an automated, interactive proof assistant. It is *automated* in the sense that no user input is expected once it has embarked on the search for the proof of a conjecture. It is *interactive* in the sense that the search is significantly affected by the previously proven lemmas in its database at the beginning of a proof attempt; the user essentially programs the theorem prover by stating lemmas for it to prove, to use automatically in subsequent proofs. ACL2 also supports a goal-directed interactive loop (called the "*proof-checker*"); but it is much less frequently used and not relevant to the discussions here.

The following is a grossly simplistic model of the interaction of a user with the ACL2 theorem prover, that is nevertheless sufficient for our purpose. The user creates a theory (extending GZ) using the extension principles to model some artifact of interest. Then, she poses a conjecture about the functions in the theory and instructs the theorem prover to prove the conjecture. For instance, if the artifact is the factorial function above, one conjecture might be the formula `mfact-is-natp`, which says that *mfact* always returns a natural number. We repeat it below.

Theorem `mfact-is-natp`
$natp(mfact(x)) = \texttt{T}$

ACL2 attempts to prove a conjecture by applying a sequence of transformations to it, replacing each goal (initially, the conjecture) with a list of subgoals. Internally, ACL2 stores each goal as a *clause* represented as an object in the ACL2 universe. A goal of the form $\tau_1 \wedge \tau_2 \wedge \cdots \wedge \tau_n \Rightarrow \tau$ is represented as a list of terms, $(\neg\tau_1 \ \cdots \ \neg\tau_n \ \tau)$, which is viewed as the disjunction of its elements (*literals*). ACL2 has a *hint* mechanism which the user can use to provide pragmatic advice on proof search at any goal or subgoal; in this example, the user can advise ACL2 to begin the search by inducting based on the term $mfact(x)$.

Once a theorem is proven, it is stored in a database and used in subsequent derivations. This database groups theorems into various *rule classes*, which affects how the theorem prover will automatically apply them. The default rule class is *rewrite*, which causes the theorem prover to replace instances of the left-hand side of an equality with its corresponding right-hand side; if the conjecture `fact-is-natp` above is a rewrite rule, then if ACL2 subsequently encounters a term of the form $natp(fact(\tau))$ then the term is rewritten to `T`.

3.5 Structuring Mechanisms

An important feature of ACL2 is its support for proof structuring. Structuring mechanisms permit designation of some events to be *local* to a specific scope. Above, we described the use of local events while discussing encapsulation; recall that the successful admission of an encapsulation extends an ACL2 theory with only the *nonlocal* events in the encapsulation. A common use of encapsulation is to introduce functions $f_1 \ldots f_k$, where the only properties about $f_1 \ldots f_k$ that are deemed useful subsequently are given by formulas $\varphi_1 \ldots \varphi_m$. To prove $\varphi_1 \ldots \varphi_m$, it might be necessary to introduce auxiliary events (definitions and lemmas) $\psi_1 \ldots \psi_n$. Once $\varphi_1 \ldots \varphi_m$ are proven, the user can "wrap" all the events inside an encapsulation with signatures declaring the introduction of $f_1 \ldots f_k$ and marking each ψ_i as *local*.

Sometimes encapsulation is used to structure proofs without introducing new functions. For instance, one might declare an empty signature and then prove a series of local lemmas leading to a desired (nonlocal) theorem; the effect is to export the theorem without cluttering ACL2's database with the lemmas necessary to prove it.

Besides encapsulation, another common structuring mechanism is a *book*. A book is a file events that can be successfully processed (in order) by ACL2. A book can be *certified* by ACL2 to facilitate its use in subsequent sessions: a certified book can be *included* without rerunning the associated proofs. Like encapsulations, books can contain local events. The local events are processed when a book is certified but skipped when it is included. Before including a certified book, ACL2 only performs a syntactic check to ensure that the names of the nonlocal function symbols in the book do not conflict with those introduced in the current session.

Books and encapsulations are crucial to the success of ACL2 in large-scale verification projects. See Kaufmann's case study [120] for a pedagogical example of their use in developing a mechanical proof of the Fundamental Theorem of Calculus. Structured proofs produce a smaller database of events, making it easier for the user to control the theorem prover. Furthermore, books permit users to combine and build on work of others (as well as their own past work). For instance, if an existing book contains a thousand theorems ten of which are useful for a new project, the user can create a new book that provides only the ten useful theorems as follows. The new book locally includes the old one and then nonlocally states the desired theorems: including this new book imports only the ten desired theorems.

Unfortunately, structuring mechanisms complicate ACL2's logical foundations. Consider a book `book1` which contains a nonlocal theorem φ which is proven by first locally introducing an auxiliary function f and perhaps proving (locally) a series of lemmas about f. Assume that the formula φ itself does not contain the function symbol f; nevertheless, ACL2 might have made use of the definitional axiom of f to prove φ. When `book1` is later included, φ is exported but *not* the local axiom defining f which has been used in the derivation of φ. Indeed, in the new session, one can introduce f with a different definitional axiom from that in `book1`. The above remarks also apply to encapsulations.

Such possibilities require a careful formulation of the logical foundations of ACL2 and, in particular, clarification of the role played by local events. That is, one must clarify under what conditions it is legitimate to mark an event to be local. This question has been answered by Kaufmann and Moore [129] as follows: if a formula ϕ is proven as a theorem in an ACL2 session, then ϕ is in fact first-order derivable from the axioms of GZ together with only the axiomatic events in the session (that is, event commands that extend a theory with new axioms) that (hereditarily) involve function symbols in ϕ or in formulas introduced via the *defaxiom principle*. In particular, every extension principle other than the defaxiom principle produces an extension to the current theory which is *conservative*. Thus, any event that does not correspond to a *defaxiom* and does not involve the function symbols in the nonlocal events can be marked local. ACL2 implements this by making two passes on a book during certification. In the first pass, it proves each theorem (and admits each axiomatic event) sequentially and ensures that no *defaxiom* event is marked local. In the second pass, it performs a *local incompatibility check*, skipping proofs and checking that each axiomatic event involved in any nonlocal event is also nonlocal.

3.6 Evaluators

ACL2 provides a convenient notation for defining an *evaluator* for a set of functions. Evaluators are integral to ACL2's "metareasoning" capabilities and will come in handy in defining the correctness of external tools connected to ACL2.

To understand the role of evaluators, consider the following challenge. Given a term that represents an equality between two sums, how do we arrange to cancel common addends from both sides of the equality? For instance, we want the term $(x + 3y + z = a + y + b)$ to be simplified to $(x + 2y + z = a + b)$. Note that ACL2 does not provide unification for associative–commutative functions. If one knows ahead of time the maximum number of addends that could appear in a sum, then one can write a (potentially large) number of rewrite rules to handle all permutations in which the common addends could appear. But this does not work in general, and is laborious, both for the user in developing the rules, and for ACL2 in sorting through a large database of rules any one of which is unlikely to be useful in most situations.

One solution to the challenge is to write a customized reasoning function (called a *metafunction*) for manipulation of terms. Terms are represented naturally as objects in the ACL2 universe; the term $foo(x)$ is represented as the object `(foo x)`, which is a list of two elements. Metafunctions manipulate this *internal representation* of a term, producing (the internal representation of) a provably equivalent term. It is an easy exercise to write a metafunction that cancels common addends from a term representing equality between two sums.

The notion of an evaluator makes explicit the connection between a term and its internal representation. Assume that $f_1, \ldots, f_n$ are functions axiomatized in some ACL2 theory $\mathscr{T}$. A function *ev*, also axiomatized in $\mathscr{T}$, is called an *evaluator* for

$f_1, \ldots, f_n$, if the axioms associated with *ev* specify a suitable evaluation semantics for the internal representation of terms composed of $f_1, \ldots, f_n$; such axioms are referred to as *evaluator axioms*. A precise characterization of all the evaluator axioms is described in the ACL2 Manual [124] under the documentation topic defevaluator; below we show one of the axioms for illustration, which defines the evaluation semantics for an m-ary function symbol f. Here, $'f$ is assumed to be the internal representation of f, and $'\tau_i$ is the internal representation of τ_i, for $1 \leq i \leq m$.

An Evaluator Axiom.
$ev(list('f, '\tau_1, \ldots, '\tau_m), a) = f(ev('\tau_1, a), \ldots, ev('\tau_m, a))$

In the formula given above, it is convenient to think of a as an a-list that maps the (internal representation of the) variables in $\tau_1, \ldots, \tau_m$ to ACL2 objects. Then, the axiom specifies that the evaluation of the list $('f\ '\tau_1\ \ldots '\tau_m)$ [which corresponds to the internal representation of $f(\tau_1, \ldots, \tau_m)$] under some mapping of free variables is the same as the function f applied to the evaluation of each τ_i under the same mapping.

Evaluators allow us to state (and prove) correctness of metafunctions as formulas in the ACL2 logic. Let *cancel-addends* be the metafunction that cancels the common addends in an equality, and let *ev* be an evaluator. Then, the following theorem is sufficient to justify the use of *cancel-addends* during proofs.

Theorem `cancel-addends-correct`
$ev(cancel\text{-}addends(\tau), a) = ev(\tau, a)$

The notion of evaluators is related to *reflection*, which is essentially the computation of ground terms in the metatheory. Reflection mechanisms are available in most theorem provers, for instance HOL, Isabelle, and PVS. The difference between these mechanisms and the evaluator constructs of ACL2 arises from the fact that the logic of ACL2 is first order. Note that in first-order logic one cannot define a single closed-form function axiomatized to be an evaluator for an arbitrary set of functions. Thus, in ACL2, one must define separate evaluators for different (fixed) sets of functions. However, ACL2 provides a macro, called `defevaluator`, that takes a collection of function names $f_1, \ldots, f_k$, and a new function symbol *ev*, and introduces (via encapsulation) the constraints axiomatizing *ev* to be an evaluator for $f_1, \ldots, f_k$.

3.7 The ACL2 Programming Environment

ACL2 is closely tied with Common Lisp. As mentioned above, it employs Lisp syntax, and the axioms in GZ for the Lisp primitives are crafted so that the return values predicted by the axioms agree with those specified in the Common Lisp Manual for arguments in the intended domains of application. Furthermore, events corresponding to the definitional principle are Lisp definitions. For instance, the formal event introducing *mfact* above also serves as a Common Lisp definition:

```
(defun mfact (n)
  (if (and (natp n) (> n 0))
      (* n (fact (- n 1)))
    1))
```

The connection with Lisp enables users to execute formal definitions efficiently; ACL2 permits the execution of all functions axiomatized in GZ and any function whose definition does not involve any constrained functions. The theorem prover makes use of this connection for simplifying ground terms. For instance, during a proof ACL2 will automatically simplify *mfact*(3) to 6 by evaluation.

ACL2 also provides a logic-free programming environment to facilitate efficient code development. One can implement any applicative Lisp function and mark it to be in *program mode*. No proof obligation is generated for such functions and no logical guarantee (including termination) is provided, but ACL2 can evaluate such functions via the Lisp evaluator. In addition, ACL2 supports a "system call" interface to the underlying operating system, which enables the user to invoke arbitrary executable code and operating system commands.

3.8 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 [21, 23, 25]. Kaufmann introduced interactive enhancements to Nqthm [20]. 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 [121, 122] have been written on the ACL2 theorem prover in addition to about 5 MB of hypertext documentation available from the ACL2 home page [124]. Our description of the ACL2 logic is based on two foundational papers written by Kaufmann and Moore [128, 129]. In addition, several papers describe the many heuristics and implementation details of the theorem prover [26, 111, 127, 176] 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 [40–42]. The general theory of ordinal notations was initiated by Church and Kleene [46], which is recounted by Rogers [219, § 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 [158], replacing the older and less succinct one. Our illustration in Fig. 3.2 was taken from the documentation on ordinal representation in the current version of ACL2 [123]. Manolios and Vroon also provide efficient algorithms for arithmetic on ordinals [157].

Harrison [98] provides a comprehensive survey of reflection in theorem proving. For obvious reasons, when soundness is of great importance, work on reflection leads to the study of the relation between formal terms and the means to compute their values. This insight on reflection has been used in Nqthm and ACL2 to develop a notion of a program designed to compute the value of a given defined function on explicit constants [22]; this program was referred to as the *executable counterpart* of the function. The idea of verified metafunctions that to be subsequently used to simplify terms during proofs was introduced in Nqthm and subsequently carried over to ACL2. Subsequently, *extended metafunctions* were introduced that could make use of certain contextual information during the simplification. Currently, in addition to these facilities, ACL2 provides two other more light-weight facilities for verified term simplification called *syntaxp* and *bind-free* [110]; the latter facilities are also available in "vanilla" and extended incarnations.

Both Nqthm and ACL2 have been used in constructing several nontrivial mechanically checked proofs. Using Nqthm, Shankar formalized and proved Gödel's Incompleteness Theorem [234], Russinoff proved Gauss' Law of quadratic reciprocity [220], and Kunen verified a version of the Ramsey's theorem [135]. Notable among the computing systems formally verified by Nqthm is the so-called CLI stack [15] consisting of the gate-level design of a microprocessor together with the operational semantics of a machine-code instruction set architecture [108]; the operational semantics of a relocatable, stack-based machine language [173]; the operational semantics of two high-level languages [76, 260]; some application programs [257]; and an operating system [14]. Another significant success with Nqthm is the verification by Yu [27] 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 [31, 32] with its ISA, the IEEE compliance of the floating point division algorithm of the AMD K5™ processor [179] and RTL implementations of floating point operations of Athlon™ [223, 224] and Opteron™ processors, modeling and verification of the Rockwell Collins JEM1 processor which is the first Java Virtual Machine implemented in silicon [92], and an analysis of the FIPS-140 level 4 certification of the IBM 4578 secure coprocessor [239]. Currently, a detailed operational semantics of the Java Virtual Machine is being modeled with ACL2 [148], which includes about 700 pages of formal definitions, and several JVM bytecode programs have been proven correct for this model [147].

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 Chaps. 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 Sect. 3.3.1. We now discuss how we do that in general for sequential programs.

Recall from Chap. 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 *pc*(s) and *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 *prog*(s) that is pointed to by *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.

Definition.
instr(s) = *nth*(*pc*(s),*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, *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.

S. Ray, *Scalable Techniques for Formal Verification*, DOI 10.1007/978-1-4419-5998-0_4,

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

Definition.
step(s)=*effect*(s,*instr*(s))

Remark 4.1. 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, *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 steps from s.

Definition.
$run(s, n) = \mathit{if}\ zp(n)\ \mathit{then}\ s\ \mathit{else}\ run(step(s), n-1)$

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

Theorem `run-compose`
$natp(m) \wedge natp(n) \Rightarrow run(s, m+n) = run(run(s, m), n)$

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

Definition.
halting(s)=(*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 Chap. 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.

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

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.

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

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? Here, we discuss two popular approaches, *e.g.*, *stepwise invariants*[1] and *clock functions*.

4.2.1 Stepwise Invariants

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

[1] Stepwise 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 systems.

The function *inv* is often referred to as a *stepwise 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 theorem.

Theorem `inv-run`
$\mathit{inv}(s) \Rightarrow \mathit{inv}(\mathit{run}(s, n))$

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*, 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** to **I5**. Assume for contradiction that termination is 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.

Theorem `halting-identity`
$\mathit{halting}(s) \Rightarrow \mathit{run}(s, n) = s$

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 Comparison of Proof Styles

Operational semantics allow us to specify a clear statement of correctness as a succinct formula in a mathematical logic. Contrast this with program logics as discussed in Chap. 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.

In spite of its clarity, it is generally believed that operational semantics are more difficult to use in the actual verification. The use of stepwise invariants and clock functions constitute two 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, which we showed in Fig. 2.1. Recall that the program consists of a single loop which is executed ten 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 has been loaded in s starting from location 1. The precondition and postcondition for this program are given by the following functions.

Precondition and Postcondition
$\mathit{pre}(s) = (\mathit{pc}(s) = 1) \wedge \mathit{prog\text{-}loaded}(s)$
$\mathit{post}(s) = (X(s) = 10)$

The predicate *prog-loaded* is a boiler-plate 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 stepwise invariant is shown in Fig. 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

Definitions.

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

$$\mathit{inv}(s) = \mathit{inv\text{-}aux}(s) \wedge \mathit{prog\text{-}loaded}(s)$$

Fig. 4.1 Stepwise invariant for the one-loop program

Definitions.

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

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

Fig. 4.2 Clock function for the one-loop program

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 $\mathit{step}(s)$ must do so too. Consider a state $\hat{s}$ so that *inv* asserts $A_{\mathit{step}}(\mathit{step}(\hat{s}))$. Then *inv*, to satisfy **I2**, must also assert $A(\hat{s})$ such that $A(\hat{s}) \Rightarrow A_{\mathit{step}}(\mathit{step}(\hat{s}))$. This requirement complicates the definition of a stepwise invariant and forces one to think 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.

So far we have only talked about the issues involved in defining *inv*. 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 Fig. 4.2. Observe 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. No assertion is involved that characterizes the state reached by the program at every program counter value. It seems, therefore, that coming up with a clock function is significantly easier than coming up with a stepwise invariant. However, the "devil" lies in

Theorem `lpc-characterization`
$natp(Y(s)) \wedge prog\text{-}loaded(s) \wedge (pc(s) = 3) \Rightarrow$
$run(s, lpc(s)) = upd(s, X(s) + lpc(s), Y(s) - lpc(s))$

Fig. 4.3 A key lemma for the one-loop program. Here $upd(s, a, b)$ be the state obtained by assigning the value a to component `X` and the value b to component `Y`, respectively, in state s

the details. First, to admit the definition of *lpc* under the definitional principle, we must prove that the sequence of recursive calls terminates. To show this, we must be able to define a measure m so that following formulas are theorems.

Measure Conjectures.
$o\text{-}p_{\prec}(m(s))$
$\neg zp(Y(s)) \Rightarrow m(run(s, 4)) \prec m(s)$

But this can be achieved only if the loop itself terminates! Indeed, the function seems to precisely characterize the time complexity of the program. Note that for our program the definition can be admitted with the measure $m(s) = nfix(Y(s))$.

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 Fig. 4.3. The formula can be proven as a theorem by induction based on the term $lpc(s)$. **TC1** and **TC2** now follow from this theorem and Theorem `run-compose`.

4.4 Verifying Program Components and Generalized Proof Obligations

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 Theorem `halting-identity` shows, running from a *halting* state for *any* number of steps must leave the machine at the same state. This is an effective characterization 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 now generalize the correctness theorems (and proof styles) so that we can attach postconditions to the exitpoints of components like subroutines.

First, we 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.

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

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 discussed above by replacing the function *halting* with *exit*. Thus, for example, we can think of replacing **TC2** with the following proof obligation.

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

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 correctness statements must be modified so that the postcondition is asserted at the first *exit* state reachable from a *pre* state. This is achieved by the following modification to the statements on partial and total correctness statements.

Generalized Partial Correctness
$$\mathit{pre}(s) \wedge \mathit{natp}(n) \wedge \mathit{exit}(\mathit{run}(s, n)) \\ \wedge (\forall m : \mathit{natp}(m) \wedge (m < n) \Rightarrow \neg \mathit{exit}(\mathit{run}(s, m))) \\ \Rightarrow \mathit{post}(\mathit{run}(s, n))$$

Correspondingly, for total correctness, we will add the following generalized termination requirement.

Generalized Termination
$\mathit{pre}(s) \Rightarrow (\exists n : \mathit{natp}(n) \wedge \mathit{exit}(\mathit{run}(s, n)))$

Note that in the special case of terminating programs, we have the predicate *exit* to be *halting* and we get our original **Partial Correctness** and **Termination** theorems from the generalized conditions mentioned above.

How do we modify the proof obligations for the two proof styles that we introduced above to account for arbitrary *exit* states? This is easy for the clock functions approach. Here are the proof obligations for total correctness.

GTC1: $pre(s) \Rightarrow exit(run(s, clock(s)))$
GTC2: $pre(s) \Rightarrow post(run(s, clock(s)))$
GTC3: $natp(clock(s))$
GTC4: $natp(n) \wedge pre(s) \wedge exit(run(s,n)) \Rightarrow (n \geq clock(s))$

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 $exit(run(s,n))$. The partial correctness obligations are shown below.

GPC1: $pre(s) \wedge exit(run(s,n)) \Rightarrow exit(run(s, clock(s)))$
GPC2: $pre(s) \wedge exit(run(s,n)) \Rightarrow post(run(s, clock(s)))$
GPC3: $exit(run(s.n)) \Rightarrow natp(clock(s))$
GPC4: $natp(n) \wedge pre(s) \wedge exit(run(s,n)) \Rightarrow (n \geq clock(s))$

How do we generalize the stepwise invariant approach to specify correctness of program components? The issues with stepwise 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 stepwise invariant obligations are shown below. The obligations **GI1**–**GI3** are for partial correctness, and **GI4** and **GI5** are additionally required for total correctness.

GI1: $pre(s) \Rightarrow inv(s)$
GI2: $inv(s) \wedge \neg exit(s) \Rightarrow inv(step(s))$
GI3: $inv(s) \wedge exit(s) \Rightarrow post(s)$
GI4: $o\text{-}p_{\prec}(r(s))$
GI5: $inv(s) \wedge \neg exit(s) \Rightarrow r(step(s)) \prec r(s)$

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 Chap. 5. 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.

4.5 Discussion

At least at a first glance, it appears then that stepwise invariants and clock functions have orthogonal advantages and disadvantages as verification strategies. Stepwise 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 steps are remaining before the program terminates. Stepwise 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 nontrivial 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, 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 stepwise invariants and clock functions, which we call the problem of *overspecification* and the problem of *forced homogeneity*. 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.

4.5.1 Overspecification

Both the stepwise invariant and clock function proofs of our simple program seem to require a substantial amount of "unnecessary" manual work. For instance, consider the stepwise 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 Chap. 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 Chap. 5. 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.

4.5.2 Forced Homogeneity

Stepwise 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 (1) CREATE() that creates (initializes) an empty BST and (2) 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 steps without resorting to induction; but coming up with a stepwise invariant usually involves a very clear understanding of the operational details of each instruction. On the other hand, it is probably easier to use stepwise 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? Furthermore, there are extant mechanical proofs of different program components in ACL2, using different strategies. 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. Without that, we are forced to use a single (or homogeneous) proof style for each component, even if that is not natural for the component is question, simply for the purpose of composing with proofs of other components.

The above requirement seems to be difficult 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 steps. Is it possible to view a clock function proof as a stepwise 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 stepwise 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 Chap. 6, we provide answers to these questions. We show that clock functions and stepwise 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. In fact, we do better than that: we show that both strategies, as well as the inductive assertions style that we introduce in Chap. 5, are in fact complete, so that a proof in *any* style can be mechanically translated to a proof in any other style.

4.6 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.

4.7 Bibliographic Notes

McCarthy [161] 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 Chap. 3, have involved operational semantics. Liu and Moore's model of the JVM [148] 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 models in HOL include Norrish's formalization of the semantics of C [190] and Strecker's models of Java and the JVM [243]. In PVS, operational models have been used for UML state charts [96].

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

Chapter 5
Operational Semantics and Assertional Reasoning

In this chapter, we remove one of the deficiencies we identified, namely overspecification. In particular, we show how 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? The key ingredient here is our ability to admit tail-recursive partial functions. In particular, we will define a tail-recursive function that counts the number of steps from one cutpoint to the next. We will show how we can prove certain theorems about this function that enable us to derive the proof obligations involved in assertional reasoning by symbolic simulation. Furthermore, it will be possible to use a theorem prover itself for symbolic simulation with no necessity for any external VCG.

5.1 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 Chap. 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 that are referred to as *cutpoints*. The cutpoints are simply states that 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 Chap. 2, the cutpoints for our one-loop program in Fig. 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).

S. Ray, *Scalable Techniques for Formal Verification*, DOI 10.1007/978-1-4419-5998-0_5,

Let us abstract the details of the specific cutpoints for a program and assume that we can define a unary predicate *cutpoint*, so that *cutpoint*(s) returns T if s is a cutpoint and NIL otherwise. So, for our example, the function will be defined as follows.

Definition.
cutpoint(s) = (*pc*(s) = 1) ∨ (*pc*(s) = 3) ∨ (*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.

Definition.

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

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

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*.

Definition.
csteps-aux(s, i) = *if cutpoint*(s) *then* i *else csteps-aux*(*step*(s), i + 1)
csteps(s) = *csteps-aux*(s, 0)

Of course, before using the above definitions, one must first justify why the function *csteps-aux* would be admissible. To admit it under the definitional principle we should 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 [153]. Briefly, they show that any function axiomatized by a tail-recursive equation is admissible in ACL2. To elaborate a bit, assume that we wish to extend a theory $\mathscr{T}$ by introducing an axiom as follows.

Definitional Axiom.
f(x)=*if* $\alpha(x)$ *then* $\beta(x)$ *else* $\gamma(x)$

Here, $\alpha(x)$, $\beta(x)$, and $\gamma(x)$ are assumed to be any terms expressible in $\mathscr{T}$, and the function symbol f is assumed to be outside the language of $\mathscr{T}$. Then it is possible to find a witness for f using the defchoose principle that satisfies the above axiom. Thus, we can extend $\mathscr{T}$ 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. Defining this function is easy using the choice principle as follows:

Definition.
D-exists() = ($\exists s : \neg$*cutpoint*(s))
dummy() = $wit_{\textit{D-exists}}()$

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

Definition.
nextc(s) = *if cutpoint*(*run*(s, *csteps*(s))) *then run*(s, *csteps*(s)) *else dummy*()

Given functions *cutpoint*, *assertion*, and *nextc*, the formulas **AR1**–**AR4** below are formal renditions of the VCG guarantees for partial correctness. In particular, **AR4** stipulates that if some *cutpoint* s that is not an *exit* state satisfies *assertion*, then the next *cutpoint* encountered in an execution from s must also satisfy *assertion*.

V1: *pre*(s) $\Rightarrow$ *assertion*(s)
V2: *assertion*(s) $\Rightarrow$ *cutpoint*(s)
V3: *exit*(s) $\Rightarrow$ *cutpoint*(s)
V4: *assertion*(s) $\wedge$ *exit*(s) $\Rightarrow$ *post*(s)
V5: *assertion*(s) $\wedge \neg$*exit*(s) $\wedge$ *exit*(*run*(s, n)) $\Rightarrow$ *assertion*(*nextc*(*step*(s)))

Of course, we need to prove that the conditions **V1**–**V5** above imply **Generalized Partial Correctness** condition stated in Chap. 4. We say a few words about the actual theorem we derive in our framework. The theorem is similar to **Generalized Partial Correctness**, but is stated a bit differently. First, we define the following two functions.

Definition.
esteps(s, i) = *if exit*(s) *then* i *else esteps*(*step*(s), $i + 1$)
nexte(s) = *run*(s, *etsteps*(s, 0))

The theorem we prove for program correctness is the following.

Theorem `restated-partial-correctness`
pre(s) $\wedge$ *exit*(*run*(s, n)) $\Rightarrow$ *exit*(*nexte*(s)) $\wedge$ *post*(*nexte*(s))

We will postpone a discussion on how this theorem guarantees the **Generalized Partial Correctness** condition to Chap. 6, where we will take up that question in the context of analyzing soundness and completeness of different proof strategies. In the next section, we will see how this statement helps us to achieve compositionality.

How about total correctness? To obtain total correctness theorems we must also specify a ranking function. Recall from our discussion of stepwise 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 stepwise invariant approach by attaching whatever we need to attach only to cutpoints. Analogously, when designing a ranking function r, we will only require it to decrease (according to some well-founded relation) along cutpoints. This is formalized by the additional proof obligations **V6** and **V7** below, together with the necessary strengthening of **V5** to **V5′**. The strengthening ensures (from **V2**) that for each cutpoint p there *does* exist a subsequent next cutpoint satisfying *assertion*.

V5′: $\mathit{assertion}(s) \wedge \neg \mathit{exit}(s) \Rightarrow \mathit{assertion}(\mathit{nextc}(\mathit{step}(s)))$
V6: $\mathit{assertion}(s) \Rightarrow \mathit{o\text{-}p}_{\prec}(\mathit{rank}(s))$
V7: $\mathit{assertion}(s) \wedge \neg \mathit{exit}(s) \Rightarrow \mathit{rank}(\mathit{nextc}(\mathit{step}(s))) \prec \mathit{rank}(s)$

V6 stipulates that "$\prec$" is well-founded, and **V7** stipulates that the rank decreases along consecutive cutpoints as long as an *exit* state is not reached. The two conditions, therefore, together attach well-founded ranks to the cutpoints. The theorems let us derive the following modified version of correctness theorem for total correctness.

Theorem `restated-total-correctness`
$\mathit{pre}(s) \Rightarrow \mathit{exit}(\mathit{nexte}(s)) \wedge \mathit{post}(\mathit{nexte}(s))$

Again, we postpone to Chap. 6 the proof of this theorem and its connection with the Theorems **Generalized Partial Correctness** and **Generalized Termination**.

Remark 5.1. It is possible to have minor variations of the above verification conditions as follows. (1) We can remove the restriction that the cutpoints include initial states by refining **V1** to be $\mathit{pre}(s) \Rightarrow \mathit{assertion}(\mathit{nextc}(s))$. (2) We can remove **V2** and use the conjunct $\mathit{cutpoint}(s) \wedge \mathit{assertion}(s)$ instead of $\mathit{assertion}(s)$ in the antecedents of **V4**, **V5**, **V5′** **V6**, and **V7**. (3) We can remove **V3** and use the disjunct $\mathit{cutpoint}(s) \vee \mathit{exit}(s)$ instead of $\mathit{cutpoint}(s)$ in **V2**. We ignore such variations in the rest of this presentation.

5.2 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 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

step(*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, **SSR1** and **SSR2** will let us simplify *nextc*(p) to *step*(*step*(p)). Given any state p from which some cutpoint q is reachable in a constant number of steps (and q is the nearest cutpoint from p); **SSR1** and **SSR2** thus let us simplify *nextc*(p) to q. Since this is done by expansion of the state transition function of the operational model, by our discussion on Sect. 3.3.1 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 **V5**, **V5'**, and **V7**, all involve application of the function *nextc* on some argument! If it is the case that whenever the proof obligations are required to be discharged for any state p 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. However, note the requirement that for every *cutpoint* p, the next cutpoint must be reachable in a constant number of steps; 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 from a VCG.

How do we build a generic verification framework from the above results? Recall that the proofs of **SSR1** and **SSR2** can be derived simply from the definition of *nextc* above. Analogously, as we will see in the Chap. 6, the proof of **Generalized Partial Correctness** (respectively, **Generalized Partial Correctness** and **Generalized Termination**) from the verification conditions **V1**–**V5** (respectively, **V1**–**V4**, **V5′**, **V6**–**V7**) is independent of the actual definitions of *step*, *pre*, *post*, *cutpoint*, and *exit* for any specific program (other than the requirement that the verification conditions are theorems about those functions). We carry out the mechanical derivation using the encapsulation principle; that is, we model the "inductive assertion style" proof for partial (respectively, total) correctness by encapsulated functions *step*, *pre*, *post*, *cutpoint*, *assertion*, etc., constrained only to satisfy **V1**–**V5** (respectively, **V1**–**V4**, **V5′**, **V6**–**V7**), and derive the correctness theorems from the constraints.[1] Because of the use of encapsulation, we can now use functional instantiation to instantiare this generic proof for a concrete system description. Indeed, the instantiation can be completely automated in ACL2 by macros.

We have not talked about macros in our overview of ACL2 in Chap. 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

[1] Although we describe all proofs in this book 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.

generates a definition $f(x_1, \ldots, x_n) = \tau$ (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? Assume that the concrete operational semantics are given by the function step_c, and the precondition, postcondition, step invariant, exit point characterization, and we are interested in partial (respectively, total) correctness. We then mechanically generate the functions csteps_c, nextc_c, the concrete versions of *csteps* and *nextc* that use step_c for *step*. The verification then proceeds along the following recipe.

1. Functionally instantiate theorems **SSR1** and **SSR2** for the generated functions.
2. Use the symbolic simulation rules and prove concrete versions of **V1**–**V5** (respectively, **V1**–**V4**, **V5′**, **V6**–**V7**).
3. Functional instantiation of Theorem `restated-partial-correctness` (respectively, `restated-total-correctness`) completes the proof.

5.3 Composing Correctness Statements

The basic method above did not treat subroutines compositionally. Consider verifying a procedure P that invokes a subroutine Q. Symbolic simulation from a cutpoint of P might encounter an invocation of Q, resulting in symbolic execution of Q. Thus, subroutines have been treated as if they were in-lined. We often prefer to separately verify Q, and use its correctness theorem for verifying P. We now extend the method to afford such composition.

We will use the symbols P and Q to refer to invocations of the caller and callee respectively. We also use a subscript to distinguish between predicates about P and Q when necessary, for example referring to the postcondition for P as post_P.

For composition, it is convenient to extend the notion of *exit* states as follows. We define a predicate in_P to characterize states, which are poised to execute an instruction in P or one of its callees. Then define $\mathit{exit}_P(s) \triangleq \neg \mathit{in}_P(s)$. Thus, exit_P recognizes *any* state that does not involve execution of P (or any subroutine), not just those that return control from P. Note that this does not change the notion of the *first* *exit* state from P. With this view, we add the new verification condition **CC** below, stating that no cutpoint of P is encountered during the execution of Q. The condition will be used in the proofs of additional rules **SSR3** and **SSR3′** that we define later, which are necessary for composition.

CC: $\quad \forall s : \mathit{cutpoint}_P(s) \Rightarrow \mathit{exit}_Q(s)$

Another key ingredient for composition is the formalization of *frame conditions* necessary to prove that P can continue execution after Q returns. A postcondition specifying that Q correctly performs its desired computation is not sufficient to guarantee this. For instance, Q, while correctly computing its return value, might corrupt the call stack preventing P from executing on return. To account for this, post_Q needs to characterize the global effect of executing Q, that is, specify how

each state component is affected by the execution of Q. However, such global characterization of the effect of Q might be difficult. In practice, we require that post_Q is strong enough such that for any state s satisfying exit_Q and post_Q we can infer the control flow for continuing execution of P. For instance, if Q updates some "scratch space," which is irrelevant to the execution of P, then post_Q need not characterize such update. Then we prove the additional symbolic simulation rule **SSR3** (respectively, **SSR3′**) below, which (together with **SSR1** and **SSR2**) affords compositional reasoning about total (respectively, partial) correctness of P assuming that Q has been proven totally (respectively, partially) correct based on the restated partial (respectively, total) correctness theorems of Sect. 5.1. Here the function excut_P is given by the following definition.

Definition.
$\mathit{excut}_P(s) \triangleq \mathit{cutpoint}_P(\mathit{nextc}_P(s))$.

SSR3: $\forall s : \mathit{pre}_Q(s) \Rightarrow \mathit{nextc}_P(s) = \mathit{nextc}_P(\mathit{nexte}_Q(s))$
SSR3′: $\forall s : \mathit{pre}_Q(s) \wedge \mathit{excut}_P(s) \Rightarrow \mathit{nextc}_P(s) = \mathit{nextc}_P(\mathit{nexte}_Q(s))$

Proof. We only discuss **SSR3** since the proof of **SSR3′** is similar. By **CC** and the definition of esteps_Q, if s satisfies pre_Q and $n < \mathit{esteps}_Q(s, 0)$, then $\mathit{run}(s, n)$ does not satisfy $\mathit{cutpoint}_P$. Hence, the next $\mathit{cutpoint}_P$ state after s is the same as the next $\mathit{cutpoint}_P$ state after the first exit_Q state reachable from s. The rule now follows from the definitions of *nextc* and *nexte*. □

We prioritize rule applications so that **SSR1** and **SSR2** are tried only when **SSR3** (respectively, **SSR3′**) cannot be applied during symbolic simulation. Therefore, if Q has been proven totally correct and if a noncutpoint state s encountered during symbolic simulation of P satisfies pre_Q, then **SSR3** "skips past" the execution of Q; otherwise we expand the transition function via **SSR2** as desired.

We need one further observation to apply **SSR3′** for composing partial correctness proofs. Note that **SSR3′** has the hypothesis $\mathit{excut}_P(s)$. To apply the rule, we must, therefore, know for a symbolic state s satisfying pre_Q whether some subsequent cutpoint of P is reachable from s. However, such a cutpoint, if one exists, can only be encountered *after* s. The solution is to observe that for partial correctness we can weaken the verification condition **V5** to **V5′** below. For a cutpoint s satisfying assertions, **V5′** requires the next subsequent cutpoint to satisfy the assertion only if some such cutpoint is reachable.

V5′: $\forall s{:}\mathit{assertion}(s) \wedge \neg \mathit{exit}(s) \wedge \mathit{excut}(\mathit{step}(s)) \Rightarrow \mathit{assertion}(\mathit{nextc}(\mathit{step}(s)))$

V5′ allows us to assume $\mathit{excut}_P(\mathit{next}(s))$ for any nonexit cutpoint s of P. Now let b be some pre_Q state encountered during symbolic simulation. We must have previously encountered a nonexit cutpoint a of P such that there is no cutpoint between $\mathit{next}_P(a)$ and b. Assuming $\mathit{excut}_P(\mathit{next}(a))$ we can infer $\mathit{excut}_P(b)$ by the definitions of *excut* and *nextc*, enabling application of **SSR3′**.

Note that while we used the word "subroutine" for presentation, our treatment does not require P or Q to be subroutines. One can mark *any* program block by

defining an appropriate predicate *in*, verify it separately, and use it to compositionally reason about programs that invoke it. In practice, we separately verify callees that (a) contain one or more loops, and (b) are invoked several times, possibly by several callers. If Q is a straight-line procedure with complicated semantics, for instance some complex initialization code, we skip composition and allow symbolic simulation of P to emulate in-lining of Q.

We now turn to recursive procedures. So far we have considered the scenario, where Q has been verified *before* P. This is not valid for recursive programs, where P and Q are invocations of the same procedure. Nevertheless, we can still *assume* the correctness of Q while reasoning about P. The soundness of the assumption is justified by well-founded induction on the number of machine steps needed to reach the first *exit* state for P, and the fact that recursive invocations of P execute in fewer steps than P itself.

Given the above infrastructure for composition, the following list summarizes the mechanical steps to compositionally verify programs using our framework. Again, as for the basic method in the preceding section, we implement a macro that performs the steps below:

1. Mechanically generate concrete versions of the functions *csteps*, *nextc*, *esteps*, etc., for the given operational semantics.
2. Functionally instantiate the generic symbolic simulation rules **SSR1**, **SSR2**, and **SSR3** (respectively, **SSR3′**), and the justification for recursive procedures.
3. Use symbolic simulation to prove the verification conditions.
4. Functional instantiation of Theorem `restated-partial-correctness` (respectively, `restated-total-correctness`) completes the proof.

We end the discussion on the verification framework with a comment on its generality. A criticism might be that our framework does not facilitate reuse in the following sense. Note that we insist on a stylized expression of the correctness statement for each program component, for example, the restated version of the theorems. However, there is a significant body of code proof in ACL2 already that makes use of clock functions. There are also other proofs with stepwise invariants. It might therefore appear that we need to reverify each such previously verified program components with inductive assertions in order to use our framework for verifying (say) a program that calls them. However, we will show in Chap. 6 that this is not the case. In particular, we will show that if a program is verified in *any* manner, then it is possible to mechanically generate an inductive assertion style proof.

5.4 Applications

In this section, we discuss applications of the method in verification of concrete programs. We start with an assembly language Fibonacci program on a simple machine model called TINY [91]. The subsequent examples are JVM bytecodes compiled from Java for an operational model of the JVM in ACL2 called M5 [178].

The details of TINY or M5 are irrelevant to this chapter; we chose them since they are representative of operational machine models in ACL2, and their formalizations were accessible to us.

5.4.1 Fibonacci Implementation on TINY

TINY is a stack-based 32-bit processor developed at Rockwell Collins Inc [91]. The Fibonacci program shown in Fig. 5.1 is the result of compiling the standard iterative implementation for this machine. TINY represents memory as a linear address space. The two most recently computed values of the Fibonacci sequence are stored in addresses `20` and `21`, and the loop counter `n` is maintained on the stack. TINY performs 32-bit integer arithmetic. Given a number k the program computes $\mathit{fix}(\mathit{fib}(k))$, where $\mathit{fix}(n)$ returns the low-order 32 bits of n, and *fib* is the mathematical Fibonacci function defined below:

Definition.
$\mathit{fib}(k) = \mathit{if}\ (zp(k) \vee k = 1)\ \mathit{then}\ 1\ \mathit{else}\ \mathit{fib}(k-1) + \mathit{fib}(k-2)$

The *pre*, *post*, and *exit* predicates for the verification of the Fibonacci program[2] are shown in Fig. 5.2, and the assertions at the different cutpoints in Fig. 5.3. They

```
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*
```

Fig. 5.1 TINY assembly code for computing the nth Fibonacci sequence. The numbers to the *left* of each instruction is the pc value for the loaded program. High-level pseudocode is shown at the extreme *right*. The `add` instruction at pc value `129` removes `0` from the top of stack; this trick is necessary since TINY has no `DROP` instruction

[2] Functions *pre* and *post* here take an extra argument k while our generic proofs used unary functions. This is admissible since one can functionally instantiate constraints with concrete functions

Definition.
$pre(k, s) = (pc(s) = *\texttt{start}*) \wedge (tos(s) = k) \wedge (k \geq 0) \wedge (fib\text{-}loaded(s))$
$post(k, s) = (tos(s) = fix(fib(k)))$
$exit(s) = (pc(s) = *\texttt{halt}*)$

Fig. 5.2 Predicates *pre*, *post*, and *exit* for the Fibonacci program. Here $pc(s)$ and $tos(s)$ return the program counter and top of stack at state s, and *fib-loaded* holds at state s if the program in Fig. 5.1 is loaded in the memory starting at location `*start*`

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

Fig. 5.3 Assertions for the Fibonacci program

$$\begin{array}{l} \cdots \wedge fib\text{-}loaded(s) \wedge pc(s) = *\texttt{loop}* \wedge 0 < tos(s) < k \ \wedge \\ mem(21, s) = fix(fib(k - tos(s) - 1)) \ \wedge \\ mem(20, s) = fix(fib(k - tos(s))) \\ \Rightarrow \\ tos(s) - 1 < k \end{array}$$

Fig. 5.4 Verification condition generated during symbolic simulation of Fibonacci loop

are fairly traditional. The key assertion is the loop invariant, which specifies that the numbers at addresses `20` and `21` are $fix(fib(k - \texttt{n}))$ and $fix(fib(k - \texttt{n} - 1))$, respectively, where `n` is the loop count stored at the top of the stack when the control reaches the loop test. For partial correctness, no further user input is necessary. Symbolic simulation proves the standard verification conditions.

In Fig. 5.4, we show one of the verification conditions generated (and immediately discharged) in the process. This condition checks that the loop count stored at the top of the stack is less than k at the end of the iteration under the assumption that the assertions hold at the beginning of the iteration and the loop is executed (i.e., the loop count is nonzero).

For total correctness, we additionally use the function *rank* below that maps the cutpoints to the well-founded set of ordinals below ε_0.

Definition.
$rank(\mathrm{s}) = \textit{if } exit(s) \textit{ then } 0 \textit{ else } (\omega \cdot_o tos(s)) +_o abs(*\texttt{halt}* - pc(s))$

Here ω is the first infinite ordinal, and $\cdot_o$ and $+_o$ represent ordinal multiplication and addition. Informally, *rank* is a lexicographic ordering of the loop count and the difference between the location `*halt*` and $pc(\mathrm{s})$.

---

having extra arguments, as long as such arguments do not affect the parameters (in this case s) involved in the constraints [19].

```
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
```

Fig. 5.5 M5 bytecodes for the factorial method

5.4.2 Recursive Factorial Implementation on the JVM

Our next example involves JVM bytecodes for a recursive implementation of the factorial program (Fig. 5.5). We use an operational model of the JVM called M5, developed at the University of Texas [178]. M5 defines the semantics for 138 JVM instructions, and supports invocation of static, special, and virtual methods, inheritance rules for method resolution, multithreading, and synchronization via monitors. The bytecodes in Fig. 5.5 are produced from the Java implementation by disassembling the output of `javac` and can be executed with M5.

The example is an entertaining illustration of our treatment of recursion. With the exception of the recursive call, the procedure involves a straight-line code. Thus, we only need to specify the precondition and the postcondition. The precondition posits that the state s is poised to start executing the bytecodes for `fact` on argument k; the postcondition specifies that the return state pops the top frame from the call stack and stores $\mathit{fix}(\mathit{fact}(k))$ on the frame of the caller, where *fact* is the mathematical factorial function. No further annotation is necessary. When symbolic simulation reaches the state in which the recursive call is invoked, it skips past the call (inferring the postcondition for the recursive call) and continues until the procedure exits. This stands in stark contrast to all the previously published ACL2 proofs of the method [177, 178], which require complex assertions to characterize each recursive frame in the call stack.

5.4.3 CBC-Mode Encryption and Decryption

Our third example is a more elaborate proof of functional correctness of a Java program implementing encryption and decryption of an unbounded array of bits. By *functional correctness*, we mean that the composition of encryption and decryption yields the original plaintext. Functional correctness of cryptographic protocols has received considerable attention recently in formal verification [238, 249]. We refer the reader to Schneier [232] for an overview of cryptosystems.

Cryptographic protocols use a *block cipher* that encrypts and decrypts a fixed-size *block* of bits. We use blocks of 128 bits. Encryption and decryption of large data streams additionally require the following operations.

- A *mode of operation* extends the cipher from a single block to arbitrary block sequences. We use *Cipher Block Chaining* (CBC), which "xor's" a plaintext block with the previous ciphertext in the sequence before encryption.
- *Padding* expands a bit sequence to one which is a multiple of a block length, so as to apply a block cipher; *unpadding* drops the padding during decryption.
- *Blocking* involves transforming an array of bits to an array of blocks for use by CBC encryption; *unblocking* is the obvious inverse.

Our Java implementation performs the following sequence of operations on an unbounded bit-array (a) padding, (b) blocking, (c) CBC encoding, (d) CBC decoding, (e) unblocking, and (f) unpadding. It follows Slind and Hurd's HOL model of the operations [238], adapted for bit arrays. However, we do *not* implement a practical block cipher; our cipher "xor's" a 128-bit block with a key based on a fixed key schedule. The program constitutes about 300 lines of Java (with 18 subroutines), which compiles to 600 bytecodes.

We verify both partial and total functional correctness. The precondition specifies that the class table containing the routines is loaded and the current call frame contains a reference to an array a of bits in the heap; the postcondition requires that the array on termination is the same as a. Using ACL2 libraries on arithmetic and arrays, the only nontrivial user inputs necessary for the proofs are the loop invariants for the associated procedures. Furthermore, the only property of the block cipher used in reasoning about the CBC methods is that the encryption and decryption of 128-bit blocks are inverses. Thus, it is now possible to independently prove this invertibility property for a practical block cipher, and "plug in" the cipher to obtain a proof of the corresponding unbounded bit-array encryption.

5.5 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 [168] to reason about pointers in Isabelle, and Norrish [190] to reason about C programs in HOL. The bibliographic notes in Sect. 5.7 list some of these approaches.

Two recent research results in ACL2 that are most related to our approach are the work of Moore [177] for reasoning about partial correctness of operational models, and the work of Matthews and Vroon [160] for reasoning about termination. Neither method addresses compositionality and handles recursive programs the way we do, but our basic approach draws inspiration from both of these methods. To the knowledge of the author, these are the only previous approaches that 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 toward 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.

Definition.
$inv(s) = \mathit{if}\ cutpoint(s)\ \mathit{then}\ assertion(s)\ \mathit{else}\ inv(step(s))$

The method then attempts to prove that *inv* is a stepwise invariant. However, instead of defining separate symbolic simulation rules, it uses the definition of *inv* itself. 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 stepwise invariance of *inv* we must prove $inv(step(s))$. If $step(s)$ is not a *cutpoint*, then the definition of *inv* merely says that $inv(step(s)) = inv(step(step(s)))$. Applying the definition repeatedly will lead us simply to a term containing repeated composition of *step*s until we reach a *cutpoint*. Say $step(step(s))$ is a *cutpoint*. Then the expansion above leads us to the following proof obligation.

Proof Obligation.
$cutpoint(s) \wedge assertion(s) \Rightarrow assertion(step(step(s)))$

But that this is simply our obligation **V5** for this case. The approach can be used to prove partial correctness, when (as in our case) 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? The symbolic simulation rules in this approach are "merely" by-products for showing that *inv* is a stepwise 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 [160] 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 cannot 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.

5.6 Summary

We have presented a method to apply assertional reasoning for verifying sequential programs based on operational semantics, that is suitable for use in mechanical theorem proving. Symbolic simulation is used for generating and discharging verification conditions, which are then traded for the correctness theorem by automatically generating a tail-recursive clock. Partial and total correctness are handled uniformly. The method is compositional in that individual procedures can be verified component-wise to prove the correctness of their composition. It also provides a natural treatment of recursive procedures.

The method unifies the clarity and concreteness of operational semantics with the abstraction provided by assertional methods without requiring a VCG for the target language. To understand why implementing a VCG for a realistic programming language is difficult, consider the method invocation instruction of the JVM. This instruction involves method resolution with respect to the object on which the method is invoked, and side effects on many parts of the states such as the call frames, heap (for synchronized methods), and the class table (for dynamic methods). Encoding such operations as predicate transformation instead of state transformation is nontrivial. Furthermore, most VCGs perform on-the-fly formula simplifications to generate manageable verification conditions. As a measure of the complexity, the VCG for the Java certifying compiler ran to about 23,000 lines of C in 2001 [58]. In our approach, only one trusted tool, namely an off-the-shelf theorem prover, is necessary, while still inheriting the benefits of a VCG; the requisite simplifications are performed with the full power of the theorem prover.

It is possible to think (as some researchers do) 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.

Note, however, that practical VCGs may implement substantial static analysis on the control flow of the program. For instance, the VCG for ESC/Java performs static analysis to elide assertions from join points of conditionals without incurring exponential case blow-up [75]. To emulate them with a theorem prover, the simplification engine and lemma libraries must be powerful enough to encode such transformations. This is an avenue for significant future research.

5.7 Bibliographic Notes

The notion of assertions was made explicit in a classic paper by Floyd [77], although the idea of attaching assertions to program points appears much earlier, for example, in the work of Goldstein and von Neumann [84] and Turing [251]. Program logics

were introduced by Hoare [103] to provide a formal basis for assertional reasoning. Dijkstra [67] introduced weakest preconditions and guarded commands for the same purpose. King [134] wrote the first mechanized VCG. Implementations of VCGs are abound in the program verification literature. Some of the recent substantial projects involving complicated VCG constructions include ESC/Java [66], proof carrying code [185], SLAM [11], 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 [80] uses a verified VCG to derive the correctness of code proofs of an imperative language in PVS. Homeier and Martin [106] verify VCGs for an imperative language in HOL. Assertional methods have been applied, using a verified VCG, to reason about pointers in Isabelle [168], and C programs in HOL [190].

Moore [177] 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 [160] present a related approach to reason about termination. A comparison of the method presented here with these two approaches appears in Sect. 5.5.

There are parallels between this work and research on proof-carrying code (PCC) [186]. VCGs are the key trusted components in PCCs. Similar to our work, foundational PCC research [4] ensures reliability of verification condition generation by relying only on a general-purpose theorem prover and the operational semantics of a machine language. However, while PCCs focus on automatic proofs of fixed safety properties (such as type and memory safety), our approach is geared toward verifying functional program correctness that requires more general-purpose assertions. We achieve this by using the simplification mechanisms of a theorem prover to automate verification condition generation.

An early implementation of our ACL2 macro is currently distributed with ACL2. Several researchers have personally communicated to the authors independent endeavors applying and extending the method. At Galois Connections Inc., Pike has applied the macro to verify programs on the Rockwell Collins AAMP7^{TM} processor. At the National Security Agency, Legato has used it to verify an assembly language multiplier for the Mostek 6502 microprocessor. At Rockwell Collins Inc., Hardin et al. are independently extending the method and using it for AAMP7 and JVM code verification [97]. Fox has formalized the method in HOL4 and is applying it on ARM assembly language programs.

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 [159], with permission from the other coauthors.

Chapter 6
Connecting Different Proof Styles

In Chap. 5, we showed how to overcome one of the deficiencies that we identified in Chap. 4 as overspecification. In this chapter, we address the other issue, for example, forced homogeneity. In particular, we will see how we can mix different proof strategies for sequential programs modeled using an operational semantics. For instance, 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. We actually prove the following stronger result. Recall the three proof styles (1) stepwise invariants, (2) clock functions, and (3) inductive assertions. We prove that each style is both *sound* and *complete*. Completeness means that if there is a proof of correctness of the program using *any* strategy, there is a way to mechanically translate the proof into one in any of the three strategies. Our results on composition in Chap. 5 then permit us to mix different proof styles for different program components for free.

In addition to the obvious practical effect of affording the use of different proof styles on different components of the same program, the results we derive have ramifications in our understanding of program verification theory. In spite of the obvious importance of mathematical theories underlying program correctness, there has been relatively little research on analyzing the expressive power of such theories.[1] However, it has been informally believed that the logical guarantees provided by the strategies are different. Our results show that such informal beliefs are flawed. Indeed, the results themselves are not mathematically deep; a careful formalization of the strategies essentially leads to the results. Nevertheless, our approach clarifies the workings of the different strategies to the practitioners of program verification: without the requisite formalization, it is easy to accept apparently reasonable but flawed notions of program correctness. We discuss this point while explaining the implications of the results in Sect. 6.4.

[1] One notable exception is the Hoare logic. There has been significant work on analysis of soundness and completeness of Hoare axioms for various programming language constructs. See the bibliographic notes for this chapter.

S. Ray, *Scalable Techniques for Formal Verification*, DOI 10.1007/978-1-4419-5998-0_6,

Quantification and Skolemization are crucial to the results in this chapter. In particular, we make critical use of the fact that whenever a quantified predicate is defined, the logic introduces the corresponding Skolem witness that can be used in subsequent definitions.

6.1 Soundness of Proof Styles

The soundness of a proof strategy entails showing that if the corresponding proof obligations are met, then one can derive the correctness statements, for example, one can mechanically prove Theorem **Generalized Partial Correctness** (and, for total correctness, also **Generalized Termination**). Since each strategy has been extensively used for mechanized program verification, these results are not surprising; we discuss the proofs mainly to provide a flavor of the reasoning involved in doing them formally. We have carried out the soundness proofs in ACL2 both to validate our formalization of the correctness statements and to facilitate implementation of macros for switching strategies.

We start with the stepwise invariants. Recall that the stepwise invariants strategy involves the following proof obligations, where 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: $\mathit{inv}(s) \wedge \neg \mathit{exit}(s) \Rightarrow r(\mathit{step}(s)) \prec r(s)$

We first prove that **Generalized Partial correctness** follows from **GI1** to **GI3**. To prove this, we must show that for each state s from which some *exit* state is reachable, the first reachable *exit* state from s satisfies *post*. The key lemmas for the proof are shown in Fig. 6.1. The lemmas are interpreted as follows. Let s be an arbitrary

Definition.
$\mathit{n\text{-}exit}(s,n) = \mathit{if}\ (\neg \mathit{natp}(n) \vee (n=0))\ \mathit{then}\ \neg \mathit{exit}(s)\ \mathit{else}\ (\neg \mathit{exit}(\mathit{run}(s,n)) \wedge \mathit{n\text{-}exit}(s, n-1))$

Theorem `inv-for-internal`
$\mathit{inv}(s) \wedge \mathit{n\text{-}exit}(s,n) \Rightarrow \mathit{inv}(\mathit{run}(s,n))$

Theorem `inv-for-first-exit`
$\mathit{inv}(s) \wedge \mathit{natp}(n) \wedge \mathit{n\text{-}exit}(s,n) \wedge \mathit{exit}(\mathit{run}(s,n+1)) \Rightarrow \mathit{inv}(\mathit{run}(s,n+1))$

Fig. 6.1 Key Lemmas in the Proof of Soundness of Stepwise invariants. Theorem `inv-for-internal` can be proven using **GI2** by induction on n. Theorem `inv-for-first-exit` follows from Theorem `inv-for-internal`, **GI2** and the fact that if $\mathit{n\text{-}exit}(s,n)$ holds then $\neg \mathit{exit}(\mathit{run}(s,n))$

non-*exit* state satisfying *inv*, and n be a natural number such that (a) there is no *exit* state reachable from s in n or fewer steps, and (b) the state s' reachable from s in $(n+1)$ steps satisfies *exit*. Consider the sequence $\langle s, \mathit{step}(s), \ldots, s' \rangle$. Each non-*exit* state in the sequence satisfies *inv* (Theorem `inv-for-internal`), and in addition, the first *exit* state s' satisfies *inv* as well (Theorem `inv-for-first-exit`). Then by **GI1** and **GI3**, for each non-*exit* state s satisfying *pre*, s' satisfies *post*. Finally, if s is an *exit* state that satisfies *pre*, then by **GI1** and **GI3**, s (which is the first reachable *exit* state from s) also must satisfy *post*.

We now show that total correctness follows from **GI1** to **GI5**. It suffices from the above to show that **Generalized Termination** follows from these obligations. To prove the latter, assume by way of contradition that no *exit* state is reachable from a state s satisfying *pre*. By **GI1** and **GI2**, each state in the sequence $\langle s, \mathit{step}(s), \ldots \rangle$ satisfies *inv*. Thus, by **GI4** and **GI5**, the sequence forms an infinite decreasing chain on O with respect to the relation $\prec$, violating well-foundedness.

We now discuss the soundness of clock functions. Recall that total correctness clock functions involve the following proof obligations.

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

GTC1–**GTC4** guarantee total correctness. By **GTC1** and **GTC3**, for each state s satisfying *pre*, there exists some natural number m, namely $\mathit{clock}(s)$, such that $\mathit{run}(s,m)$ satisfies *exit*. Finally, **GTC1**, **GTC3**, and **GTC4** guarantee that the state $\mathit{run}(s, \mathit{clock}(s))$ is the first *exit* state reachable from s, and, by **GTC2**, this state satisfies *post*.

For partial correctness, we have the following weaker set of obligations.

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

Partial correctness follows from **GPC1** to **GPC4**. This is trivial since even with the weakening, if, for a state s satisfying *pre*, there exists a natural number n such that $\mathit{run}(s,n)$ satisfies *exit*, then there still exists a natural number m, namely $\mathit{clock}(s)$, such that after m steps the first *exit* state is reached and this state satisfies *post*.

We next consider the inductive assertions method discussed in Chap. 5. The proof obligations we had for partial correctness are the following:

V1: $\mathit{pre}(s) \Rightarrow \mathit{assertion}(s)$
V2: $\mathit{assertion}(s) \Rightarrow \mathit{cutpoint}(s)$
V3: $\mathit{exit}(s) \Rightarrow \mathit{cutpoint}(s)$

V4: $\mathit{assertion}(s) \wedge \mathit{exit}(s) \Rightarrow \mathit{post}(s)$
V5: $\mathit{assertion}(s) \wedge \neg \mathit{exit}(s) \wedge \mathit{exit}(\mathit{run}(s, n)) \Rightarrow \mathit{assertion}(\mathit{nextc}(\mathit{step}(s)))$

Partial correctness follows from **V1** to **V5**. The proof is analogous to that for stepwise invariants, but here we focus only on cutpoints. First observe that if s is not an *exit* state and some *exit* state is reachable from s then, by **V3** and the definition of *nextc*, some *exit* state is reachable from $\mathit{nextc}(\mathit{step}(s))$. Now, by **V1**, the initial states (those satisfying *pre*) satisfy *assertion*. By **V5** and the observation above, it follows that if a state s satisfies *assertion*, it follows that every reachable cutpoint from s up to (and, by **V3**, including) the first reachable *exit* state s' satisfies *assertion*. Then from **V4**, we infer that s' satisfies *post*.

For total correctness, we additionally attach a well-founded ranking function at cutpoints, as formalized by the following proof obligations (in addition to the strengthening of **V5** to **V5′**).

V5′: $\mathit{assertion}(s) \wedge \neg \mathit{exit}(s) \Rightarrow \mathit{assertion}(\mathit{nextc}(\mathit{step}(s)))$
V6: $\mathit{assertion}(s) \Rightarrow \mathit{o\text{-}p}_{\prec}(\mathit{rank}(s))$
V7: $\mathit{assertion}(s) \wedge \neg \mathit{exit}(s) \Rightarrow \mathit{rank}(\mathit{nextc}(\mathit{step}(s))) \prec \mathit{rank}(s)$

Total correctness follows from **V1** to **V4**, **V5′**, **V6** to **V7**. It suffices to show that some *exit* state is reachable from each cutpoint p. By **V2** and **V5′**, for each cutpoint p, the state $\mathit{nextc}(\mathit{step}(p))$ is a subsequent cutpoint reachable from p. But by well-foundedness, **V6**, and **V7**, one of these cutpoints must be an *exit* state, proving the claim.

6.2 Completeness

We now turn our attention to proving that each of the three proof strategies above is complete. By completeness, we mean that given any proof of correctness of a program we can mechanically construct a corresponding proof in any of the strategies. The completeness results do not depend on the structure or style of the alleged original proof; given that the obligation **Partial Correctness** is a theorem (together with **Termination** for total correctness proofs) for some operational model defined by *step* and corresponding predicates *pre*, *post*, and *exit*, we define appropriate functions and predicates that meet the obligations necessary for each strategy.

We start with *clock functions*. The function *clock* below satisfies the relevant obligations of a clock function proof. Note that the function *esteps* is the same function as the one we used in Chap. 5.

Definition.
$\mathit{esteps}(s, i) = \mathit{if}\ \mathit{exit}(s)\ \mathit{then}\ i\ \mathit{else}\ \mathit{esteps}(\mathit{step}(s), i + 1)$
$\mathit{clock}(s) = \mathit{esteps}(s, 0)$

Assuming that **Generalized Partial Correctness** is provable, we now show that the conditions **GPC1**–**GPC4** are provable for this definition. A key lemma in the proof

Theorem `esteps-characterization`
$$
\begin{array}{l}
\mathit{exit}(\mathit{run}(s,n)) \wedge \mathit{natp}(i) \Rightarrow \\
\quad \textbf{let rslt} \leftarrow \mathit{esteps}(s,i) \textbf{ in} \\
\quad\quad \textbf{let stps} \leftarrow (\textbf{rslt} - i) \textbf{ in} \\
\quad\quad\quad \mathit{natp}(\textbf{rslt}) \wedge \\
\quad\quad\quad \mathit{natp}(\textbf{stps}) \wedge \\
\quad\quad\quad \mathit{exit}(\mathit{run}(s,\textbf{stps})) \wedge \\
\quad\quad\quad (\mathit{natp}(n) \Rightarrow (\textbf{stps} \leq n))
\end{array}
$$

Fig. 6.2 Key lemma about *esteps*. Theorem `esteps-characterization` is proven by induction on n. For the base case, note that if $\neg \mathit{natp}(n) \vee (n = 0)$ holds then the lemma is trivial. For the induction hypothesis we assume the instance of the formula under the substitution $[s \rightarrow \mathit{step}(s), n \rightarrow (n-1), i \rightarrow (i+1)]$. The induction step follows from the definition of *esteps*

is `esteps-characterization`, which is shown in Fig. 6.2. The lemma can be interpreted as follows. Let s be an arbitrary state and i be a natural number, and assume that there is some *exit* state reachable from s in n steps. Let (a) **rslt** be the value returned by $\mathit{esteps}(s,i)$, and (b) **stps** be the difference $(\textbf{rslt} - i)$. Then **rslt** and **stps** are natural numbers, executing the machine for **stps** times from s results in an *exit* state, and **stps** is less or equal to n. Thus, if there is some n such that $\mathit{run}(s,n)$ is an *exit* state then $\mathit{clock}(s)$ returns a natural number that counts the number of steps to the first reachable *exit* state from s, satisfying **GPC1**, **GPC3**, and **GPC4**. Finally, from **Generalized Partial Correctness**, if s satisfies *pre*, then $\mathit{run}(s, \mathit{clock}(s))$ must satisfy *post*, proving **GPC2**.

Remark 6.1. The reader familiar with ACL2 will note that since *esteps* has been defined with tail-recursion (and might not always terminate), there can be no induction scheme associated with the recursive structure of this definition. The proof of Theorem `esteps-characterization` requires an explicit induction scheme with the extra parameter n, namely the number of steps to reach some *exit* state from s if such a state exists.

Finally, we now prove the stronger total correctness obligations **GTC1**–**GTC4** as follows assuming that both **Generalized Partial Correctness** and **Generalized Termination** are provable. Let $n(s)$ be the Skolem witness for the existential predicate in the **Generalized Termination** formula. By **Generalized Termination** and the properties of Skolemization, we know that for a state s satisfying *pre*, $\mathit{run}(s, n(s))$ satisfies *exit*. The obligations **GTC1**, **GTC2**, and **GTC3** now follow by instantiating the variable n in **GPC1**, **GPC2**, and **GPC3** with $n(s)$.

Remark 6.2. The above arguments justify the use of the restated correctness obligations, for example, Theorem `restated-partial-correctness` for partial correctness and Theorem `restated-total-correctness` for total correctness. The key observation here is that $\mathit{esteps}(s,0)$ is a clock function. Thus, we can mechanically derive the obligations **Generalized Partial Correctness** and **Generalized** Termination from the restated versions from the soundness proof of the clock functions.

We now consider *stepwise invariants*. From the results above, we can assume without loss of generality that the correctness proof has been translated to a proof involving clock functions using the definitions of *esteps* and *clock*. We define the relevant invariant *inv* below.

Definition.

$$\begin{aligned} \mathit{inv}(s) = (\exists p, m : \ & \mathit{pre}(p) \wedge \\ & \mathit{natp}(m) \wedge \\ & (s = \mathit{run}(p, m)) \wedge \\ & ((\exists \alpha : \mathit{exit}(\mathit{run}(p, \alpha))) \Rightarrow (m \leq \mathit{clock}(p)))) \end{aligned}$$

The definition can be read as follows. A state s satisfies *inv* if s is reachable from some *pre* state p and the path from p to s contains no *exit* state, except perhaps for s itself.

We now derive **GI1**–**GI3** from the definition of *inv*. For **GI1** note that given a state s satisfying *pre* we can prove $\mathit{inv}(s)$ by choosing the existentially quantified variables p and m to be s and 0, respectively. For **GI3**, let s be an *exit* state satisfying *inv* and let $p(s)$ and $m(s)$ be the Skolem witnesses for p and m, respectively. Then by Theorem `esteps-characterization` and definitions of *inv* and *clock*, $p(s)$ satisfies *pre* and s is the first *exit* state reachable from $p(s)$; **GI3** follows from **Partial Correctness**. Finally, to prove **GI2**, we assume that *inv* holds for some non*exit* state s, and show that *inv* holds for $\mathit{step}(s)$. For this, we must determine a state q satisfying *pre* and a natural number n such that $\mathit{step}(s) = \mathit{run}(q, n)$; furthermore, if there is an *exit* state reachable from $\mathit{step}(s)$, then $n \leq \mathit{clock}(q)$. Let $p(s)$ and $m(s)$ be the Skolem witnesses for $\mathit{inv}(s)$ as above. Then we choose q to be $p(s)$ and n to be $(m(s) + 1)$. Note that by the definition of *inv*, we have $s = \mathit{run}(p(s), m(s))$; thus by definition of *run*, $\mathit{step}(s) = \mathit{run}(p(s), m(s) + 1)$. Finally, if some *exit* state is reachable from $\mathit{step}(s)$, then it is also reachable from s. Since s is not an *exit* state, by Theorem `esteps-characterization` and definition of *clock*, we know (a) $\mathit{natp}(\mathit{clock}(p(s)))$ and (b) $m(s) < \mathit{clock}(p(s))$. Thus $(m(s) + 1) \leq \mathit{clock}(p(s))$, proving **GI2**.

For total correctness, we define the following *measure*.

Definition.

$$\mathit{measure}(s) \triangleq \mathit{clock}(s)$$

We can now prove **GI4** and **GI5**. Let s be an arbitrary state satisfying *inv*. Then as above, s must be reachable from some state $p(s)$ satisfying *pre*, and there is no *exit* state in the path from $p(s)$ to s. By **Termination**, some *exit* state is reachable from $p(s)$; thus, some *exit* state is reachable from s. Now, by Theorem `esteps-characterization` and definition of *clock*, $\mathit{clock}(s)$ is a natural number (and hence an ordinal), proving **GI4**. Furthermore, for *any* state s, $\mathit{clock}(s)$ gives the number of transitions to the first reachable *exit* state. Thus, if s is not an *exit* state and an *exit* state is reachable from s (by *inv* and **Termination**), then $\mathit{clock}(\mathit{step}(s)) < \mathit{clock}(s)$, proving **I5**.

We now consider inductive assertions. Obviously, this strategy would reduce to stepwise invariants if each state were a cutpoint. However, our goal is to attach assertions (and ranking functions) to a collection of cutpoints *given a priori*. We use the following definitions for *assertion* and *rank*. Here *inv* is the same predicate that we used in proving completeness of stepwise invariants above. Note that the definition of *assertion* contains a case split to account for **V2**.

Definition.
assertion(s) = *if cutpoint*(s) *then inv*(s) *else* `NIL`
rank(s)*clock*(s)

The proof of completeness of assertional reasoning is analogous to that for stepwise invariants. The only nontrivial lemma necessary for partial correctness requires establishing the following. "Suppose that a non-*exit* cutpoint s satisfies *assertion* and let s' be *nextc*(*step*(s)). Then if some *exit* state is reachable from s' there exists a state p and a natural number m such that $s' = \mathit{run}(p, m)$ and $m \leq \mathit{clock}(p)$." We exhibit such p and m as follows. Assume $p(s)$ and $m(s)$ to be the Skolem witnesses for *inv*(s) as in the proof of completeness of stepwise invariants; note from above that *assertion* is defined in terms of *inv*. We then take p to be $p(s)$ and m to be $(m(s) + 1 + \mathit{csteps}(\mathit{step}(s), 0))$. The proof thus reduces to showing $(m(s) + 1 + \mathit{csteps}(\mathit{step}(s), 0)) \leq \mathit{clock}(p(s))$ for each non-*exit* cutpoint s. We prove this by first showing that there is no intermediate cutpoint between s and s'; this follows from the definition of *csteps* and is derived by induction analogous to Theorem `esteps-characterization` in Fig. 6.2 but using *csteps* instead of *esteps*. Thus, since some *exit* state is reachable from s it must also be reachable from s'. The lemma now follows from the definitions of *esteps* and *clock*. Finally, for total correctness, we note that since there is no intermediate cutpoint between s and s' the number of steps to the first *exit* state (which is what the function *clock* counts) must be less for s' than for s.

Our results above establish that if one can prove the correctness of an operationally formalized program *in any manner*, then one can mechanically derive the proof obligations of each strategy. However, the results should be regarded with a caveat. They do not imply that in practice one technique might not be easier than the other. For instance, manually writing a stepwise invariant for each pc value *is* more tedious than attaching assertions only at cutpoints. Also, the functions and predicates that we used to prove the completeness theorems might not be directly used to verify a program from scratch. For instance, the *clock* function we used essentially runs the program until some *exit* state is reached; as we saw in Chap. 4, using a clock function in practice requires a careful reflection of the control structure so that properties about loops can be proven by induction. However, our results perhaps *do* indicate that the difficulty in practical code proofs stems from the inherent complexity in reasoning about complex computing systems rather than the nuances of a particular proof style.

6.3 Remarks on Mechanization

The proofs of soundness and completeness discussed above are independent of the details of the operational model (that is, the formal definition of *step*), the precondition, and the postcondition. In ACL2, we carry out the reasoning in the abstract, essentially formalizing the proof sketches outlined in the last two sections. The relevant abstraction is achieved using encapsulation.

We use encapsulation to reason about proof strategies as follows. Consider the soundness theorem for the partial correctness of stepwise invariants. We encapsulate functions *step*, *pre*, *post*, *exit*, and *inv* to satisfy **GI1**–**GI3**, and derive the formula **Generalized Partial Correctness**. For the completeness proof we do the opposite, namely encapsulate *pre*, *step*, and *post* constrained to satisfy partial correctness and derive **GI1**–**GI3**.

We note that the mechanical proofs are not trivial, principally because ACL2 provides limited automation in reasoning about quantified formulas. The proofs, therefore, require significant manual guidance and care needs to be taken in formalizing concepts such as "the first reachable *exit* state from s." However, the basic structure of the proofs follow the high-level descriptions provided in the preceding two sections.

The generic nature of the proofs enables us to mechanically switch strategies by functional instantiation. For instance, let *step-c*, *pre-c*, *post-c*, *exit-c*, and *inv-c* be the functions involved in a stepwise invariant (partial correctness) proof of some program. We can derive the corresponding clock function proof by the following two steps:

1. Derive **Generalized Partial Correctness** by functionally instantiating the soundness theorem for stepwise invariant; the instantiation substitutes the concrete functions *step-c* for *step*, etc. ACL2 must prove that the concrete functions satisfy the constraints. But the constraints are **GI1**–**GI3** which are what have been proven for the concrete functions.
2. Derive the clock obligations by generating the *clock* described in Sect. 6.2, and functionally instantiating the completeness theorem. The constraint is exactly the statement of partial correctness for the concrete functions, which has been proven in Step 1.

We have developed a macro in ACL2 to automate the above steps for switching between any two of the strategies. Given a correctness proof for a program in a particular style and a keyword specifying the target strategy, the macro performs the above steps, generating the requisite functions and functionally instantiating the generic theorems.

6.4 Discussion

The soundness and completeness theorems are not conceptually deep, once the strategies are formalized. The only nontrivial insight in the derivations is the understanding that quantification and Skolemization can be used *in the logic* to define

functions characterizing the set of reachable states and the number of steps to termination. In spite of the simplicity, however, several researchers have been surprised when shown the completeness theorems (before seeing the proofs) precisely because of the apparent dissimilarities in the workings of the different proof strategies. For instance, *clock functions* were considered to be significantly different from stepwise invariants and assertional reasoning. Indeed, clock functions had often been criticized before on the grounds that they require reasoning about efficiency of the program when "merely" a correctness theorem has been desired.[2]

Our results also provides the important foundational basis for building deductive strategies for program verification. In spite of significant research on program correctness, it is still surprisingly easy to create apparently reasonable but flawed proof strategies. To illustrate this, we consider a strategy used by Manolios and Moore [153] for reasoning sequential programs via symbolic rewriting. The strategy involves defining a function *stepw* as follows via tail-recursion:

Definition.
$stepw(s) = \mathit{if}\ halting(s)\ \mathit{then}\ s\ \mathit{else}\ stepw(step(s))$

The function induces the following equivalence relation $\hookrightarrow$ on states[3]:

Definition.
$(s_0 \hookrightarrow s) = (stepw(s_0) = stepw(s))$

The relation $\hookrightarrow$ has nice algebraic properties; for instance, the following formula is a theorem:

Theorem `step-stepw`
$s \hookrightarrow step(s)$

Let $modify(s)$ be the desired modification to s after execution of the program: if the program is a JVM method computing the factorial, then $modify(s)$ might involve popping the current call frame off the call stack and storing the factorial of the argument involved in the invocation at the appropriate local of the call frame of the caller. Manolios and Moore use the following notion to relate s with $modify(s)$.

Correctness Statement.
$pre(s) \Rightarrow (s \hookrightarrow modify(s))$

The obligation seems apparently reasonable, and substantial automation can be achieved in proving this statement for operationally modeled sequential programs. In particular, using the theorem $s \hookrightarrow step(s)$ and the fact that $\hookrightarrow$ is transitive, the

[2] This criticism has been rarely written in print, but usually expressed in conference question–answer sessions. However, there has been a nagging feeling that clock functions require more work. The absence of published criticism and the presence of this "nagging feeling" have both been confirmed by an extensive literature search and private communications with some of the authors of other theorem provers.

[3] The actual symbol they use is $==$ instead of $\hookrightarrow$. We use the latter to avoid confusion between this relation and equality.

statements can be proven and composed by symbolic simulation. However, does the above theorem imply that the program under inspection is (partially) correct in general? The answer is "no." To see this, first consider a program that never terminates. Such a program is, by definition, partially correct. However, note that for any s such that no terminating state is reachable from s, the return value of $\mathit{stepw}(s)$ is not defined by the above defining equation; thus, obligation 1 cannot be established. A stronger objection is that $s \hookrightarrow s'$ may be proved for any two states that reach the same halting state; there is no requirement or implication that s' is reachable from s. Consider a pathological machine in which all halting computations collapse to the same final state. Then, for any two states s and s' poised to execute two (perhaps completely different) halting programs, we can prove $s \hookrightarrow s'$. Thus, any two halting programs are "equivalent," and any terminating program can be proven "correct" in the sense of being equivalent to any desired halting specification.

The lesson from above is that it is imperative to validate proof strategies against a formal, clearly specified notion of correctness. We should note that the machine used in Manolios and Moore [153] was not pathological, and the programs analyzed by Manolios and Moore can also be proven correct with respect to our formalization.

6.5 Summary and Conclusion

We have formalized three proof strategies commonly employed in mechanical verification of programs modeled using operational semantics. We have shown how to mechanically derive the proof obligations for each strategy (in a logic allowing first-order quantification and arbitrary tail-recursive definitions) from any proof of correctness of a program. The results hold for both partial and total correctness. We have implemented macros to switch proof strategies in ACL2. We have also illustrated how it is possible, in absence of such a formal framework, to develop apparently reasonable but flawed proof strategies.

We do not advocate one proof strategy over another. Our goal is to enable practitioners working in program verification to go "back and forth" between the different strategies; thus, one can focus on reasoning about a program component using the strategy most suitable, independent of other components. This ability is particularly important in theorem proving, where the user needs to guide the theorem prover during a proof search. Our results and macros free the user from adhering to a monolithic strategy for any one program component solely for compatibility with proofs done for other components.

Our results also indicate that the complications in program verification arise not from the specific proof styles used but from the inherent complexity involved in reasoning. Note that if there exists *any* proof of correctness of a program, we can mechanically generate the obligations for each proof strategy. Thus, the core creative reasoning involved does not change by simply switching from one strategy to another. On a positive note, this indicates that techniques for automating code proofs in one strategy are also perhaps likely to carry over to the others.

Finally, our work emphasizes the utility of quantification in an inductive theorem prover, in particular ACL2. While ACL2 has always supported full first-order quantification via Skolemization, its expressive power has gone largely unnoticed among the users, the focus being on constructive, recursive definitions. The chief reasons for this focus are that recursive definitions are amenable to efficient executability and play to the strength of the theorem prover in doing automatic well-founded induction. These are certainly useful qualities. However, we have often found it instructive to be able to reason about a generic model of which different concrete systems are instantiations. Quantifiers are useful for reasoning about such generic models. We contend that the principal reason why the connections between the different proof strategies discussed here has gone unnoticed so far in the ACL2 community, in spite of each strategy being used extensively, is the disinclination of the ACL2 users to reason in terms of generic models and quantified predicates. We have also found quantification useful in other circumstances, for example in formalizing weakest preconditions and in reasoning about pipelined machines [210].

6.6 Bibliographic Notes

Stepwise 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 [14, 108, 173, 260], Yu's proofs of Berkeley C-string library [27,261], and proofs of JVM byte-codes [147,175,178] involve clock functions. Relatively few researchers outside this community have used clock functions, though there have been some proofs done with PVS [258]. The reason is possibly that relatively few researchers outside the Boyer–Moore community have done proofs of large programs based on operational semantics.

Analysis of the expressive power of mathematical theories for program verification has traditionally been confined to Hoare logics. The original Hoare axioms provided proof rules formalizing a few elementary programming constructs such as assignments, conditionals, and loops. The proof rules were extended to incorporate virtually all programming constructs including goto's, coroutines, functions, data structures, and parallelism [55, 56, 104, 192, 193]. Soundness of axiomatic semantics has often been established by appealing to an operational model of the underlying language. De Bakker [64] provides a comprehensive treatment of the issues involved in developing axiomatic semantics of different programming constructs. The soundness theorems for these different flavors of Hoare logic were usually established by careful mathematical arguments, but the arguments were rarely mechanized in a theorem prover. One notable exception is Harrison's [99] formalization of Dijkstra's [68] monograph "A Discipline of Programming" in HOL.

In addition, the VCG verification work cited above has often involved mechanically checking that the formula transformations implemented by the VCG are sound with respect to the operational semantics.

In addition to soundness, there has been research analyzing completeness of the different proof rules for axiomatic semantics. Clarke [48] provides a comprehensive treatment of completeness (and incompleteness) results for Hoare-like axiomatic proof systems; in addition, Apt's survey [5] provides an interesting analysis of both soundness and completeness results. Cook [60] presents a sound and complete assertional system for certain classes of programs. Wand [254] shows that the axiomatic formulation of certain constructs are incomplete. Sokolowski [240] proves the completeness of a set of axioms for total correctness.

The results described in this chapter is in collaboration with Warren A. Hunt, Jr., John Matthews, and J Strother Moore. The presentation here uses some of the text from the previous papers on the subject [211, 213] with permission from the coauthors.

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: we will formalize the correctness statements and then define and prove several proof rules to facilitate derivation of the correctness statement for concrete 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 nonterminating 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 characterize the correct executions of a reactive system is to relate them with the executions of a "high-level model" called the *specification system* ($\mathscr{S}$ for short). To distinguish $\mathscr{S}$ from the system we are interested in verifying, let us call the latter the *implementation system* ($\mathscr{I}$ for short). The (infinite) executions

S. Ray, *Scalable Techniques for Formal Verification*, DOI 10.1007/978-1-4419-5998-0_7,

of $\mathscr{S}$ define the desirable (infinite) executions of $\mathscr{I}$. Verification then entails a formal proof showing that every execution of $\mathscr{I}$ can be appropriately viewed as some execution of $\mathscr{S}$.

The above approach requires that we define some notion of correspondence that can be used to relate the execution of $\mathscr{I}$ with those of $\mathscr{S}$. 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 $\mathscr{I}$ there exists an execution of $\mathscr{S}$ that has the same visible behavior up to finite stuttering." This notion is well studied and is referred to as *stuttering trace containment* [3, 144]. We will formalize this notion in ACL2 and show how it affords definition of intuitive specifications.

7.1 Modeling Reactive Systems

We 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.

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*.[1] However, nondeterminism 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 nondeterministic 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 Chap. 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*. Second, 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

[1] Any nondeterministic system that accepts a sequence of external inputs can be modeled as a deterministic system if any of the external inputs to the system do not depend somehow on any intermediate output produced by the system. This is because we can model the system with all the external inputs as being stored initially in one of the state component; whenever an external input is required that component is consulted for the appropriate stimulus.

equivalent. For instance, given a state transition function *next*, a transition relation can be defined as follows.

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

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*.

Definition.
$M.\mathit{exec}[\mathit{stimulus}](n)$
$= \mathit{if}\ \mathit{zp}(n)\ \mathit{then}\ M.\mathit{init}()\ \mathit{else}\ M.\mathit{next}(M.\mathit{exec}[stimulus](n-1), \mathit{stimulus}(n-1))$

Observe the nonstandard naming of the function, in particular, the use of 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 scheme provides two different definitions, $M.\mathit{exec}[\mathit{stimulus}_1]$ and $M.\mathit{exec}[\mathit{stimulus}_2]$. If the logic of ACL2 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 useful (but trivial) observation, which we will make 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, *i.e.*, 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 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 $\mathscr{S}$ is related to $\mathscr{I}$ by trace containment if and only if for every execution σ of $\mathscr{I}$ there exists an execution π of $\mathscr{S}$ with the same (infinite) sequence of labels. Notice that the notion requires quantification over functions since it talks about "for all executions of $\mathscr{I}$" and "there exists an execution of $\mathscr{S}$," 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 given by the following definition.

Definition 7.1 (Trace Containment). We say that system $\mathscr{S}$ is a *trace containment* of system $\mathscr{I}$ if and only if there exists some unary function *stim* such that the following is a theorem.

Trace Containment Obligation (TC).
$\mathscr{I}.\mathit{label}(\mathscr{I}.\mathit{exec}[\mathit{ustim}](n)) = \mathscr{S}.\mathit{label}(\mathscr{S}.\mathit{exec}[\mathit{stim}](n))$

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

The characterization above 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. The idea of this sequence is made more precise in the following definition.

Definition 7.2 (Stuttering Controller). A unary function *ctr* will be called a *stuttering controller* if it always returns the member of some well-founded set,[2] *i.e.*, if the following formula is a theorem.

Well-foundedness Requirement.
$o\text{-}p_{\prec}(\mathit{ctr}(n))$

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 Fig. 7.1. By this

Definition.

$$\mathit{stutter}[ctr](n) = (\mathit{ctr}(n) \prec \mathit{ctr}(n-1))$$

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

Fig. 7.1 Definition of a stuttering trace

[2] Throughout this chapter, we will make use of well-foundedness to formalize several arguments. We assume that we use a fixed well-founded structure $\langle o\text{-}p_{\prec}, \prec \rangle$. In general, our framework may be thought to be parameterized over the well-founded structure used.

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 **Well-foundedness Requirement** in the stuttering controller. This condition, together with the definition of $\mathit{stutter}[ctr]$, guarantees that stuttering is 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 Sect. 7.4.

We now formalize the notion of stuttering trace containment in the following definition.

Definition 7.3 (Stuttering Refinement). 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 $\mathscr{I}$ is a *stuttering refinement* of $\mathscr{S}$ if and only if there are unary functions *stim* and *ctr* such that *ctr* is a stuttering controller and the following condition is satisfied.

Stuttering Trace Containment Condition (STC).
$\mathscr{I}.\mathit{label}(\mathscr{I}.\mathit{trace}[\mathit{tstim}, \mathit{tctr}](n)) = \mathscr{S}.\mathit{label}(\mathscr{S}.\mathit{trace}[\mathit{stim}, \mathit{ctr}](n))$

We write $(\mathscr{S} \rhd \mathscr{I})$ to mean that $\mathscr{I}$ is a stuttering refinement of $\mathscr{S}$. We refer to the system $\mathscr{S}$ as a *stuttering abstraction* of $\mathscr{I}$.

For a given implementation system $\mathscr{I}$ and a specification system $\mathscr{S}$, our notion of correctness is to show $(\mathscr{S} \rhd \mathscr{I})$.

7.3 Fairness Constraints

As a final point in our formalization, we will consider the issue of *fairness* and discuss how fairness constraints can be integrated with stuttering trace containment.

Why do we need fairness? Recall that the notion of stuttering trace containment stipulates that for *every* *trace* of $\mathscr{I}$ there exists a *trace* of $\mathscr{S}$ with the same *label*s up to finite stuttering. However, there are situations when we are not interested in *every* trace (or computation path) of $\mathscr{I}$ but only in certain *fair* traces. 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 Fig. 7.1, different traces correspond to different *stimulus* sequences. To consider only fair traces, we will constrain the *stimulus* to satisfy certain "fairness requirements." Informally, we want 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.

Proposed Obligation.
$natp(n) \Rightarrow (\exists m : natp(m) \wedge (m > n) \wedge \neg stutter[ctr](m) \wedge (stimulus(m) = i))$

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*. Observe that the condition $\neg stutter[ctr](m)$ 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? 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 Chap. 3, *i.e.*, 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, which is infeasible.

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 Chap. 3.[3] 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. Although this is a restriction on the inputs, we believe it is not a restriction in the practical sense since it is difficult to imagine a case in which one is interested in the behavior of a systems on inputs which are not good objects. 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 the **Well-foundedness Requirement** given above).

Fairness Condition.
1. $good\text{-}object(stimulus(n))$
2. $natp(n) \wedge good\text{-}object(i) \Rightarrow$
   $(\exists m : natp(m) \wedge (m > n) \wedge \neg stutter[ctr](m) \wedge (stimulus(m) = i))$

Is it possible to define fair input selectors? The affirmative answer comes from Sumners [245]. To do so, Sumners first shows that there is a bijection 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. $natp(g\text{-}to\text{-}n(x))$
2. $good\text{-}object(n\text{-}to\text{-}g(x))$
3. $good\text{-}object(x) \Rightarrow n\text{-}to\text{-}g(g\text{-}to\text{-}n(x)) = x$

[3] One can extend 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. It is easy to see that the extended theory is consistent: there are models of GZ which are enumerable. The restriction of the legal inputs to only constitute good objects for the purpose of formalizing fairness is necessary if such an axiom is added. However, we did not go along that path since adding axioms incurs a certain amount of logical burden for soundness and also induces practical problems, for example, introduction of axioms cannot be local (to avoid risk of inconsistency), and hence, a book in which an axiom is defined cannot be locally included.

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

Theorem `natural-fairness`
$\mathit{natp}(n) \wedge \mathit{natp}(i) \Rightarrow (\exists m : \mathit{natp}(m) \wedge (m > n) \wedge (\mathit{natenv}(m) = i))$

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 countdown 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 objects is now easy. First, we define the function *nostutter*[*ctr*] below that selects the "nonstuttering" points from the stuttering controller.

Definition.

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

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

The **Well-foundedness Requirement** 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:

Definition.
$\mathit{e\text{-}ns}[\mathit{ctr}](n) = (\exists k : \mathit{nostutter}(k) = n)$
$\mathit{fstimulus}[\mathit{ctr}](n) = \mathit{if}\ \mathit{e\text{-}ns}[\mathit{ctr}](n)\ \mathit{then}\ \mathit{n\text{-}to\text{-}g}(\mathit{wit}_{\mathit{e\text{-}ns}[\mathit{ctr}]}(n))\ \mathit{else}\ \mathit{dummy}()$

The function *fstimulus*[*ctr*] returns good objects at nonstuttering points, *dummy*( ) otherwise. It is possible, though not trivial, to prove that this function does indeed satisfy both the **Well-foundedness Requirement** and the **Fairness Condition**.

Given that there exists at least one fair selector, we modify the notion of stuttering refinement as follows to incorporate fairness. In the definition, we think of *cfsim* as a constrained fair input stimulus and *cfctr* as the corresponding stuttering selector.

Definition 7.4 (Stuttering Refinement with Fairness). Let *cfstim* and *cfctr* be unary functions constrained to satisfy the **Well-foundedness Requirement** and **Fairness Condition**. We will say that $\mathscr{I}$ is a stuttering refinement of $\mathscr{S}$ *under fairness assumption* [denoted $(\mathscr{S} \rhd_F \mathscr{I})$] if there exist functions *stim* and *ctr* such that the following formula is a theorem.

Fair Assumption for Stuttering Trace Containment (FASTC)
$\mathscr{I}.\mathit{label}(\mathscr{I}.\mathit{trace}[\mathit{cfstim}, \mathit{cfctr}](n)) = \mathscr{S}.\mathit{label}(\mathscr{S}.\mathit{trace}[\mathit{stim}, \mathit{ctr}](n))$

Analogously, we say that $\mathscr{I}$ is a stuttering refinement of $\mathscr{S}$ *with fairness requirement*, written ($\mathscr{S}_F \rhd \mathscr{I}$), if there exist fair selector functions *fstim* and *fctr* such that the following formula is a theorem (where *ustim* and *uctr* are uninterpreted).

Fair Requirement for Stuttering Trace Containment (FRSTC)
$\mathscr{I}.label(\mathscr{I}.trace[ustim, uctr](n)) = \mathscr{S}.label(\mathscr{S}.trace[fstim, fctr](n))$

We write ($\mathscr{S}_F \rhd_F \mathscr{I}$) to mean that both ($\mathscr{S} \rhd_F \mathscr{I}$) and ($\mathscr{S}_F \rhd \mathscr{I}$) hold. While fairness *assumptions* are necessary for ensuring progress properties of implementations, fairness *requirements* are usually necessary for composition. We explain this point when we discuss proof rules for stuttering refinements in Chap. 8.

How do the definitions of fairness above 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 [43]. 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 nondeterministically selecting one of the actions whose guard evaluates to true on 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 nondeterministic choice the system must make to select among the possible transitions. The restriction of the inputs to good objects 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 are 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 fairness constraints are 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 [245] 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 monograph.

We end the discussion on fairness by pointing out one deficiency in our formalization.[4] In this monograph, we typically use fairness as a way of *abstraction*, that is, hiding the traces 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, called *assume-guarantee reasoning* [167], affords the proof of correctness of each component separately under the assumption that the corresponding environment behaves properly and then the composition of these individual proofs to a proof of the entire system. But consider what happens when we can show that a component $\mathscr{I}_c$ of the implementation $\mathscr{I}$ is a refinement of a component $\mathscr{S}_c$ of the specification $\mathscr{S}$ 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 $\mathscr{I}$. On the other hand, it is possible that we can prove that $\mathscr{I}_c$ is a refinement of $\mathscr{S}_c$ under less restrictive assumptions on the environment (say an assumption that the environment selects inputs from numbers 1 to 10 infinitely often), which may be satisfied by the other components of $\mathscr{I}$. 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 circumvent it. While the current work does not address this, we believe that it is possible to extend our work to accommodate 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 **Well-foundedness Requirement** and **Fairness Condition** by replacing good objects with any other arbitrary (countable) set. With this modification, it will be possible to use compositional reasoning with fairness.

7.4 Discussion

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 need a number of definitions for doing it in ACL2, the notion itself is simple and easily understood. In Chaps. 8 and 9, we will design proof rules to facilitate proofs of such refinements and see how such rules help automate verifi-

[4] 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.

cation 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 $\mathscr{AP}$. The following proposition can be found in textbooks on temporal logic [53], but has been restated in terms of our terminology.

Proposition 7.1. *Let ψ be an LTL formula such that (1) the atomic propositions mentioned in ψ are from $\mathscr{AP}$, and (2) ψ does not have any occurrence of the operator* X*. If a system $\mathscr{S}$ satisfies ψ and $(\mathscr{S} \rhd \mathscr{I})$, then $\mathscr{I}$ satisfies ψ.*

Proof. We prove this by induction on the structure of ψ and noting from the semantics of LTL (Definition 2.3) that all the operators other than X are insensitive to finite stuttering. □

Observe that the guarantee is one-sided. Since trace containment only guarantees that every *trace* of $\mathscr{I}$ corresponds to some trace of $\mathscr{S}$, one cannot infer that an (LTL\X) formula ψ is satisfied by $\mathscr{S}$ if we know that $\mathscr{I}$ satisfies ψ.

The connection suggests that if we write the desired property of a reactive system as an (LTL\X) formula ψ, then we can check whether a property holds for $\mathscr{I}$ by checking if it holds for $\mathscr{S}$. In particular, if $\mathscr{S}$ is finite state, then we can check if ψ holds in $\mathscr{I}$ by applying model checking on $\mathscr{S}$. Doing this has been one of the key methods for integrating model checking and theorem proving. Indeed, using ACL2, Manolios et al. [154] do exactly this to verify the Alternating Bit Protocol [12]. In Chap. 13, 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 $\mathscr{I}$ to the more abstract system $\mathscr{S}$. Temporal logic specifications have a number of advantages [143], 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 $\mathscr{I}$ as encoded by the executions of the specification system $\mathscr{S}$. 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 Chap. 13, this is nontrivial, and in fact the natural semantics of LTL cannot be defined in ACL2. This limitation can, of course, be circumvented in a more expressive proof system such as HOL or PVS [45]. However, that does not mitigate the general problem of complexity. Recall 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 Chap. 8, the number of processes is left unbounded. As we discussed in Chap. 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. Furthermore, model checking is typically not applicable when reasoning about parameterized systems. Parameterized model checking is undecidable in general [6], and decidable solutions are known only under restrictions on the class of systems and properties [71], which might not be viable in practice.

On the other hand, as noted by several researchers [3, 144], 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. Furthermore, 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 $\mathscr{S}$. If we did not want stuttering to be finite, then the following trivial implementation system $\mathscr{I}$ would be considered a refinement of $\mathscr{S}$.

Trivial System.
$\mathscr{I}.\mathit{init}() = \mathscr{S}.\mathit{init}()$
$\mathscr{I}.\mathit{next}(s, i) = s$

The system $\mathscr{I}$ here always "loops" at the initial state. Note that for every trace of $\mathscr{I}$, there *is* a trace of $\mathscr{S}$ (namely the one that always stutters) that has the same label. However, if $\mathscr{S}$ satisfies a progress property (that is, a property of the form something good eventually happens), we cannot deduce that property for $\mathscr{I}$. 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 the notion of refinements given above with another related notion called Well-founded Equivalence Bisimulations (WEBs) that has been successfully used in ACL2 to reason about reactive systems [154]. 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 example, in Chap. 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 $\mathscr{S}$ and $\mathscr{I}$, WEB (and in general bisimulation) proofs involve showing that for each execution of $\mathscr{S}$ there is a corresponding execution of $\mathscr{I}$ and vice versa, while we require only that the executions of $\mathscr{I}$ can be appropriately viewed as executions of $\mathscr{S}$. 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.[5] One of the motivations for using a branching time notion of correspondence is that in the special case when $\mathscr{I}$ and $\mathscr{S}$ 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. Since we are interested in formalizing a notion of correctness rather than designing efficient algorithms, we prefer trace containment as the definition is more intuitive. As an aside, many of the proof rules we present in Chaps. 8 and 9 do preserve a branching time correspondence. Indeed, in Chap. 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.

Clearly 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

[5] Manolios [152] achieves this one-sided abstraction for branching time, by introducing proof rules for stuttering simulation.

and well studied, formalizing it in the logic takes some work. Nevertheless, at least for what we have seen, 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. Furthermore, in Chaps. 8 and 9, we will prove several proof rules that 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 later discuss model checking. In Chap. 13, we will consider one way of circumventing the limitation through a connection between ACL2 and HOL logics. At the time of this writing, this connection is in a very nascent stage; however, it has the promise to overcome the logical limitations specified here.

7.6 Bibliographic Notes

The notions of trace containment, trace equivalence, and trace congruence are discussed in a well-cited paper by Pnueli [204]. The corresponding branching time notions of simulation and bisimulation are due to Milner and Park [170, 195]. Stuttering bisimulation was introduced by Browne et al. [34].

Lamport [144] argued in favor of defining specifications that are invariant up to stuttering. Abadi and Lamport [3] presented several properties of stuttering trace containment. Several researchers have since worked on building and extending theories for reasoning about reactive systems using stuttering-invariant specifications [8, 73, 102, 117]. Namjoshi [183] presents sound and complete proof rules for symmetric stuttering bisimulation. Manolios et al. [154] 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 Sect. 7.4.

Fairness has been dealt with extensively in the context of model checking [53] and also forms an important component of logics like Unity [43] and TLA [145] 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 [245]. We briefly compared our models of fairness with Unity in Sect. 7.3.

There has been research on formalizing the metatheory of abstraction in a theorem prover. The proof rules of Unity have been formalized in Nqthm [83] and Isabelle [199]. Chou [45] formalizes partial order reductions in HOL.

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

Chapter 8
Verifying Concurrent Protocols Using Refinements

In Chap. 7, we formalized (fair) stuttering trace containment in ACL2. In this chapter, we will use it to verify several concurrent protocols. Concurrent (or multi-process) 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. Furthermore, the nondeterminism induced by independent computations of the processes makes it very difficult to detect or diagnose an error.

Irrespective of the protocol being verified, we use the same notion of correctness, for example, stuttering refinement (possibly under fairness assumption) with a specification system. 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 Sect. 8.4, we will see applications of the reduction theorems on system examples. Note 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.

Finally, 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 proven 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.

S. Ray, *Scalable Techniques for Formal Verification*, DOI 10.1007/978-1-4419-5998-0_8, 109

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 proof rule, which is called the *stepwise refinement rule*.

Stepwise Refinement Rule.
Derive $(\mathscr{S} \rhd \mathscr{I})$ from $(\mathscr{S} \rhd \mathscr{I}_1)$ and $(\mathscr{I}_1 \rhd \mathscr{I})$

The system $\mathscr{I}_1$ is called an *intermediate model*. Application of these proof rules allows us to introduce a series of intermediate models at different levels of abstraction starting from the implementation $\mathscr{I}$ and leading up to the specification $\mathscr{S}$. We then show refinements between every pair of consecutive models in this "refinement chain" and finally functionally instantiate the **Stepwise Refinement Rule** to derive the correspondence between $\mathscr{S}$ and $\mathscr{I}$.

Stepwise refinement also works with fairness. The following rules are augmentations of the basic one above, accounting for fairness assumptions and requirements.

Fair Stepwise Refinement Rules.
Derive $(\mathscr{S} \rhd_F \mathscr{I})$ from $(\mathscr{S} \rhd \mathscr{I})$.
Derive $(\mathscr{S} \rhd_F \mathscr{I})$ from $(\mathscr{S} \rhd_F \mathscr{R})$ and $(\mathscr{R} \rhd_F \mathscr{I})$
Derive $(\mathscr{S}_F \rhd_F \mathscr{I})$ from $(\mathscr{S}_F \rhd_F \mathscr{R})$ and $(\mathscr{R}_F \rhd_F \mathscr{I})$
Derive $(\mathscr{S}_F \rhd \mathscr{I})$ from $(\mathscr{S}_F \rhd_F \mathscr{R})$ and $(\mathscr{R}_F \rhd \mathscr{I})$

We end the discussion of stepwise refinements by returning to the point we postponed in Chap. 7 in the discussion of fairness, for example, the use of fairness requirements in composition. Assume that we want to prove $(\mathscr{S} \rhd \mathscr{I})$ by introducing an intermediate model $\mathscr{R}$. If we need a fairness assumption in the proof of correspondence between $\mathscr{S}$ and $\mathscr{R}$, then chaining the sequence of refinements requires that we prove the correspondence between $\mathscr{R}$ and $\mathscr{I}$ with fairness requirement.

8.2 Reduction to Single-Step Theorems

The use of stepwise refinements decomposes 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.

Definition 8.1 (Well-Founded Refinement). Given two systems $\mathscr{S}$ and $\mathscr{I}$, we will say that $\mathscr{I}$ is a *well-founded refinement* of $\mathscr{S}$, written $(\mathscr{S} \trianglerighteq \mathscr{I})$ if and only if there exist functions *inv*, *skip*, *rep*, *rank*, and *pick* such that the following formulas are theorems.

SST1: $\mathit{inv}(s) \Rightarrow \mathscr{I}.\mathit{label}(s) = \mathscr{S}.\mathit{label}(\mathit{rep}(s))$
SST2: $\mathit{inv}(s) \wedge \mathit{skip}(s,i) \Rightarrow \mathit{rep}(\mathscr{I}.\mathit{next}(s,i)) = \mathit{rep}(s)$
SST3: $\mathit{inv}(s) \wedge \neg\mathit{skip}(s,i) \Rightarrow \mathit{rep}(\mathscr{I}.\mathit{next}(s,i)) = \mathscr{S}.\mathit{next}(\mathit{rep}(s), \mathit{pick}(s,i))$

SST4: $\mathit{inv}(\mathscr{I}.\mathit{init}())$
SST5: $\mathit{inv}(s) \Rightarrow \mathit{inv}(\mathscr{I}.\mathit{next}(s,i))$
SST6: $\mathit{o\text{-}p}_{\prec}(\mathit{rank}(s))$
SST7: $\mathit{inv}(s) \wedge \mathit{skip}(s,i) \Rightarrow \mathit{rank}(\mathscr{I}.\mathit{next}(s,i)) \prec \mathit{rank}(s)$

Although well-founded refinements comprise of several conditions, they are easy to interpret. Informally, **SST1–SST7** guarantee the following correspondence between $\mathscr{S}$ and $\mathscr{I}$: "For every *execution* of $\mathscr{I}$, there is a *trace* of $\mathscr{S}$ with the same label up to finite stuttering." Given a state s of $\mathscr{I}$, $\mathit{rep}(s)$ may be thought of as returning a state of $\mathscr{S}$ 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 $\mathscr{I}$, $\mathscr{S}$ either has a "matching transition" or it stutters. The choice of $\mathscr{S}$ is governed by the predicate *skip*; if $\mathit{skip}(s,i)$ holds then **SST3** guarantees that $\mathscr{S}$ has a transition (namely by choosing the input $\mathit{pick}(s,i)$) that matches the transition of $\mathscr{I}$ from state s on input i, otherwise **SST3** guarantees that $\mathscr{S}$ 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** and **SST5** guarantee that the *inv* is an *invariant*, that is, it holds for every reachable state of $\mathscr{I}$. 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 $\mathscr{S}$ to stutter, and not $\mathscr{I}$, although the definition of stuttering trace containment allowed both. We have not yet found it necessary to use two-sided stuttering for verification of actual systems. Since $\mathscr{S}$ is the more abstract system, we want it to stutter so that several transitions of $\mathscr{I}$ can correspond to one transition of $\mathscr{S}$. 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.

Well-Founded Refinement Rule.
Derive $(\mathscr{S} \rhd \mathscr{I})$ from $(\mathscr{S} \trianglerighteq \mathscr{I})$

The proof of this rule follows by showing that the conditions **SST1–SST7** guarantee that for every execution of $\mathscr{I}$ there is a matching trace of $\mathscr{S}$.

This proof rule is essentially a formalization of a corresponding rule that has been defined for well-founded bisimulations [154, 183]. 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 $\mathscr{I}$ and $\mathscr{S}$.

Remark 8.1. The notion of well-founded refinement is *stronger* than that of **STC** in a technical sense. Well-founded refinement actually guarantees that $\mathscr{S}$ is a

[1] The function *rep* is often called the *abstraction function* in the literature on abstract interpretation [61] and is referred to as α. We prefer the more verbose name *rep* in this monograph.

simulation of $\mathscr{I}$ up to finite stuttering. Simulation is a branching time notion of correspondence. It is well known that the notion of simulation is stronger than trace containment in the sense that there are systems $\mathscr{S}$ and $\mathscr{I}$ such that $\mathscr{S}$ is related to $\mathscr{I}$ by trace containment but not by simulation [53]. 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 Chap. 9, we will use the notion of simulation itself directly as a proof rule in order to reason about pipelined machines.

How do we make fairness constraints "work" with the single-step conditions for well-founded refinements? Consider fairness assumptions. Let $\mathscr{I}^f$ be an augmentation of system $\mathscr{I}$ with an auxiliary "clock" parameter that is 0 at the initial state and incremented by 1 at each transition. We then use encapsulation to introduce a binary function *frnk* constrained to satisfy the following two conditions.

Single-Step Fair Selection.

1. $\mathit{good\text{-}object}(j) \Rightarrow \mathit{natp}(\mathit{frnk}(s, j))$
2. $\mathit{good\text{-}object}(i) \wedge \mathit{good\text{-}object}(j) \wedge (i \neq j) \Rightarrow \mathit{frnk}(\mathscr{I}^f.\mathit{next}(s, i), j) < \mathit{frnk}(s, j)$

Thus for any legal input j and any state s, if j is not selected in the transition from s then *frnk* decreases. Since $\mathscr{I}^f$ tracks the "current time," *frnk* can be constructed from the constrained fair selector using **Well-Foundedness Requirement** and **Fairness Condition** that we introduced in Chap. 7 as follows. $\mathit{frnk}(s, j)$ merely the number of transitions after s till j is selected.

Definition 8.2 (Well-Founded Refinement under Fairness Assumption). We will say that $\mathscr{S}$ is a well-founded refinement of $\mathscr{I}$ under fairness assumption [denoted $(\mathscr{S} \trianglerighteq_F \mathscr{I})$] if the following two conditions hold:

1. $(\mathscr{S} \trianglerighteq \mathscr{I}^f)$
2. The definition of the *rank* function used in the obligations **SST5** and **SST6** involves the constrained *frnk* function above.

The following proof rule connects well-founded refinements with stuttering trace containment under fairness assumptions.

Well-Founded Refinement Rule for Fairness Assumption.
Derive $(\mathscr{S} \rhd_F \mathscr{I})$ from $(\mathscr{S} \trianglerighteq_F \mathscr{I})$

A typical application of well-founded refinement proofs under fairness assumptions appears in asynchronous protocols, where the external stimulus selects the index of process to take the next step from a state s. Suppose that in some state s process 0 holds the lock to some shared resource, and every other process subsequently waits till the lock is released. To prove that such a protocol is a refinement of one in which processes access the shared resource atomically, we must invoke fairness. To do so, we use well-founded refinements under fairness assumption, by defining the *rank* as a lexicographic tuple containing $\mathit{frnk}(s, 0)$ as a component; we call this component the *fairness measure* for process 0. Then, as long as process 0 is not selected, the value of this component decreases along every transition, which

enables us to ensure finiteness of stuttering. We will see a more involved example of this in the proof of the Bakery algorithm in Sect. 8.4.2.

Analogously, for proving well-founded refinements under fairness requirements, we need a *fairness guarantee*.

Definition 8.3 (Fairness Guarantee from Input). We say that a binary function *fsel* provides a fairness guarantee with respect to binary functions *skip* and *pick* if the following conditions hold:

1. $\mathit{o\text{-}p}_{\prec}(\mathit{fsel}(s, j))$
2. $\neg \mathit{skip}(s, i) \wedge (\mathit{pick}(s, i) \neq j) \Rightarrow \mathit{fsel}(\mathscr{I}.\mathit{next}(s, i), j) \prec \mathit{fsel}(s, j)$
3. $\neg \mathit{skip}(s, i) \Rightarrow \mathit{fsel}(s, j) \not\prec \mathit{fsel}(\mathscr{I}.\mathit{next}(s, i), j)$

Recall that for well-founded refinements, the function *skip* stipulates whether $\mathscr{S}$ stutters on a transition of $\mathscr{I}$ and *pick* determines the matching input for $\mathscr{S}$ on non-stuttering steps. The conditions specify that $\mathit{fsel}(s, j)$ returns an ordinal which never increases at any transition until j is used by $\mathscr{S}$ to match a transition from $\mathscr{I}$, while it strictly decreases in the stuttering steps of $\mathscr{S}$. Thus j must be selected eventually by *pick* to match a transition of $\mathscr{I}$.

Definition 8.4 (Well-Founded Refinement under Fairness Requirement). We say that $\mathscr{I}$ is a well-founded refinement of $\mathscr{S}$ under fairness requirement [written $(\mathscr{S}_F \trianglerighteq \mathscr{I})$], if the following conditions hold.

1. $\mathscr{S} \trianglerighteq \mathscr{I}$
2. There exists a fairness guarantee function *fsel* with respect to the functions *skip* and *pick* used in the obligations for condition 1.

Finally, we can relate well-founded refinements with stuttering trace containment with fairness requirement by the following proof rule.

Well-Founded Refinement Rule for Fairness Requirement.
Derive $(\mathscr{S}_F \triangleright \mathscr{I})$ from $(\mathscr{S}_F \trianglerighteq \mathscr{I})$

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-wise 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 $\mathscr{I}$ if and only if it satisfies conditions **SST4** and **SST5**. These two conditions, of course, guarantee that *inv* holds for every reachable states of $\mathscr{I}$ 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 Chap. 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.
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.

Proof Obligation for Invariance.
RI1: $\mathit{inv}(\mathscr{I}.\mathit{init}())$
RI2: $\mathit{inv}(s) \Rightarrow \mathit{inv}(\mathscr{I}.\mathit{next}(s,i))$
RI3: $\mathit{inv}(s) \Rightarrow \mathit{good}(s)$

Clearly if we can do this then it follows that $(\mathscr{S} \trianglerighteq \mathscr{I})$. 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 Chap. 5 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 nondeterminism 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

In this section, consider a special (but important) case of refinement, for example, equivalence.

Definition 8.5 (Equivalence up to Stuttering). If $(\mathscr{S} \triangleright \mathscr{I})$ and $(\mathscr{I} \triangleright \mathscr{S})$ both hold then we say that $\mathscr{S}$ is *equivalent* to $\mathscr{I}$ (up to stuttering). We write $(\mathscr{S} \diamond \mathscr{I})$ to mean that $\mathscr{S}$ are $\mathscr{I}$ are equivalent.

One way of proving equivalence is through oblivious well-founded refinement, as discussed below. For the definition below, recall that one of the conditions for well-founded refinements (*e.g.*, **SST3**) required the existence of a function *pick*: given s and i, $\mathit{pick}(s,i)$ returned the matching input for the specification system $\mathscr{S}$ corresponding to a nonstuttering transition of $\mathscr{I}$ from state s on input i.

Definition 8.6 (Oblivious Refinement). We call $\mathscr{I}$ an *oblivious refinement* of $\mathscr{S}$, written $(\mathscr{S} \trianglerighteq_o \mathscr{I})$ if the following two conditions hold:

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

The following proof rule connects oblivious refinement with stuttering equivalence.

Oblivious Refinement Rule.
Derive $(\mathscr{S} \diamond \mathscr{I})$ from $(\mathscr{S} \trianglerighteq_o \mathscr{I})$

Why is oblivious refinement useful? Consider a simple example due to Abadi and Lamport [3]. System $\mathscr{I}$ is a 1-bit digital clock and system $\mathscr{S}$ is a 3-bit clock and the *label* of a state in each system is simply the low-order bit. Clearly, $\mathscr{I}$ implements $\mathscr{S}$ since it has the same behavior as $\mathscr{S}$ up to stuttering. However, it is easy to see that we cannot show $(\mathscr{S} \trianglerighteq \mathscr{I})$; 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 $(\mathscr{I} \trianglerighteq_o \mathscr{S})$. Given a state s of $\mathscr{S}$, (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 oblivious refinement to show the desired result.

In general, we use equivalences to add *auxiliary variables*. Suppose we want to show $(\mathscr{S} \triangleright \mathscr{I})$. We often find it convenient to construct an intermediate system $\mathscr{I}^+$ as follows. A state $s+$ of $\mathscr{I}^+$ has all the components that a state s of $\mathscr{I}$ has, but in addition, it has some more components. Furthermore, the components that are common to $\mathscr{I}$ and $\mathscr{I}^+$ are updated in $\mathscr{I}^+$ along any transition in exactly the same way as they are in $\mathscr{I}$, and the *label* of a state in $\mathscr{I}$ is the same as the label of a state in $\mathscr{I}^+$. 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 $\mathscr{I}$ in reaching from $\mathscr{I}.\mathit{init}()$ 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 $\mathscr{I}$ is matched by some *trace* of $\mathscr{S}$. Then, we can use $\mathscr{I}^+$ to explicitly store such decisions in some additional state component and use them in the proof of correspondence between $\mathscr{S}$ and $\mathscr{I}^+$. However, in doing so, we have had to change the implementation $\mathscr{I}$. To apply transitivity we must now show $(\mathscr{I}^+ \triangleright \mathscr{I})$ [resp., $(\mathscr{I}^+ \triangleright^F \mathscr{I})$]. 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, for example, that $\mathscr{I}^+$ has more state components than $\mathscr{I}$ and it is often impossible to determine a representative mapping from the states of $\mathscr{I}$ to the states of $\mathscr{I}^+$. But we can easily show $(\mathscr{I} \trianglerighteq_o \mathscr{I}^+)$, for example, we define a mapping from the states of $\mathscr{I}^+$ to the states of $\mathscr{I}$ so that only the components common to both $\mathscr{I}$ and $\mathscr{I}^+$ are preserved. Furthermore, in this case, we can prove oblivious refinements without stuttering, that is, we can prove $(\mathscr{I} \trianglerighteq_o \mathscr{I}^+)$ by choosing the predicate $\mathit{skip}(s, i)$ to be identically `NIL`.

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, a 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 one-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 that 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 one-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 one-dimensional array `excl`. For each cache line `c`, `excl[c]` contains a set of process indices that have **E**xclusive access to `c`.
- A two-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 pseudocode 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`

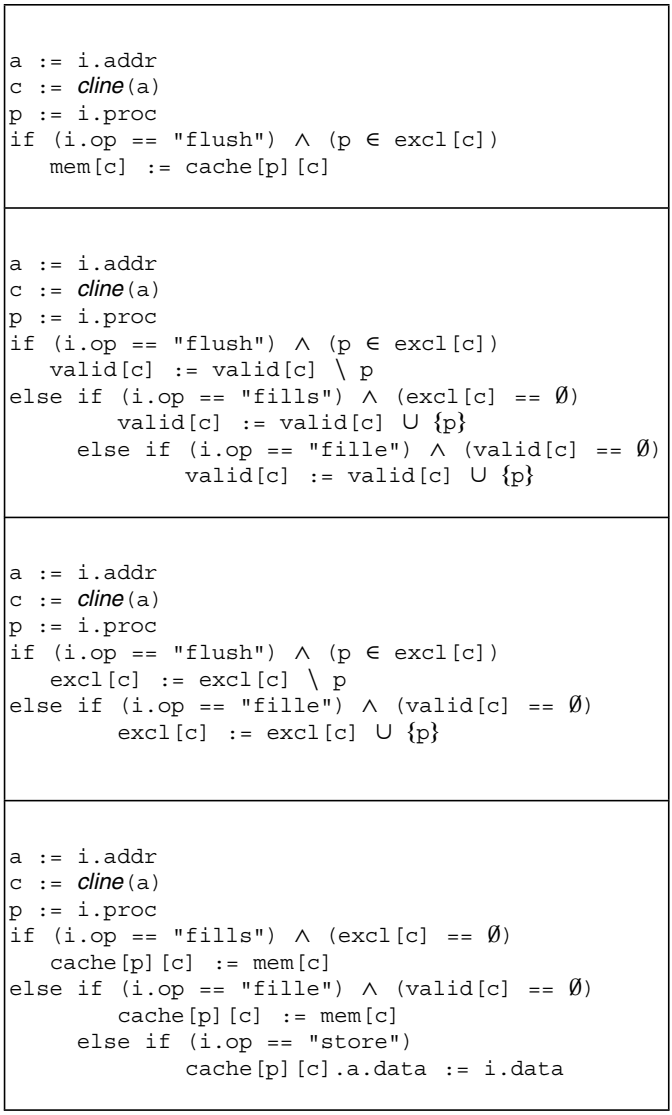

Fig. 8.1 A model of the **ESI** cache coherence protocol

gives the index of the process that makes the transition, and `i.op` stipulates the operation 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

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

Fig. 8.2 Pseudo-code for state transition of system **mem**

removed from `valid` and `excl`, and the contents of `c` in the local cache of `p` are copied to the memory.

The protocol is modeled in ACL2 as a reactive system. We 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 pseudocode in Fig. 8.2. The *label* of a state of **mem**, as for the implementation **esi**, returns the memory component. The goal of our verification is to show (**mem** $\rhd$ **esi**).

We verify this simple system by showing (**mem** $\trianglerighteq_o$ **esi**). Furthermore, we do not need stuttering for this example, that is, we will define *skip*$(s, i) =$ `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]` 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 Sect. 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**; 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 `cache[`p`][c]` has the same value 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. Note that the definition of *inv* forms the crux of the verification effort that 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 [141]. 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 pseudocode 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

```
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
```

Fig. 8.3 The Bakery Program executed by process p with Index j

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 $c << d$.

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

The algorithm is defined in ACL2 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: given a state s, **bakery**.*next*(s, i) returns the state that

[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.

the system reaches when the process with index i executes one instruction from state s.

Finally, 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 the value `"wait"`.
- If the program counter of process p has value `11`, then *bmap*(p) returns the value `"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 jth 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** $\rhd_F$ **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 properties are desirable for a mutual exclusion protocol. Unfortunately, the **progress** property does not hold for every execution of **bakery**. 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

waits for every other process that had already set `choosing` to `T` to pick their `pos` and reset `choosing`. Thus, in our scenario, q indefinitely waits 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 waiting 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 **mutual exclusion** 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 Sect. 8.3 we can justify (**spec** $\rhd_F$ **bakery**) by showing (**spec** $\rhd_F$ **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 (**spec** $\trianglerighteq_F$ **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 that 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 program counter values corresponding to `"wait"` in *label*.

As can be noticed from the descriptions above, these definitions are not very complicated. 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 toward 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.

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* [7]. 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 *nonblocking*, 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 nonblocking nature of this implementation.

A rigorous, though not mechanical, proof of the concurrent deque implementation has been done before [17]. This proof showed that any interleaving execution of the methods shown in Fig. 8.4 could be transformed into a synchronous execution

[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 [244, 246]. 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
```

Fig. 8.4 Methods for the concurrent deque implementation

in which every process invokes the entire program atomically.[4] This transformation is demonstrated by permuting different sequences of program steps, termed *bursts*, of different processes until the resulting execution is synchronous. The

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

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.

Refinement Chain.
(**cspec** $\trianglerighteq$ **icdeq**$^+ \Diamond$**icdeq**$\Diamond$**cdeq**$^+ \Diamond$**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 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 Sect. 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), $\mathit{skip}(s, i)$ 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 $\mathit{skip}(s, i)$ holds.
- The function *rep* is defined component-wise. That is, for each process i in a state s of the implementation, $\mathit{rep}(s)$ "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 $\mathit{rep}(s)$ map process i to this state. In addition, $\mathit{rep}(s)$ 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.

Remark 8.2. 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 motivation for **cdeq**$^+$ stems 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 general, for verifying a subtle system, we decompose 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, one fragment of the proof that the invariant is indeed inductive involved 1,663 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 [3, 144, 152, 154]. 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 Chap. 9, we will see that a different class of systems, namely pipelined machines, can also be verified in the same framework.

Admittedly, as we commented 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 Chap. 9, we will slightly modify the single-step theorems in order 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 proving 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 we will see how decision procedures can be formalized too.

A 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 the recent research in formal verification. Among model checking approaches, there has been work with the Java pathfinder [252] for checking assertions in concurrent Java programs. Model checking has been used to check concurrent programs written in TLA [146]. Among work with theorem proving, Jackson shows how

to verify a garbage collector algorithm using PVS [114], and Russinoff verifies a garbage collector using Nqthm [221]. In ACL2, Moore [174] describes a way of reasoning about nonblocking concurrent algorithms. Moore and Porter [180] also report the proof of JVM program implementing a multithreaded Java class using ACL2.

Abstraction techniques have also been applied to effectively reason about concurrent programs. McMillan [166] 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 [167]. Regular model checking [18] has been used to verify unbounded state systems with model checking by allowing rich assertion languages. Emerson and Kahlon [72] 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 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 (**isa** $\rhd$ **ma**).

9.1 Simulation Correspondence, Pipelines, and Flushing Proofs

In prior work on verification of *non-pipelined* microprocessors [57, 108, 109], the correspondence shown can be easily seen to be the same as trace containment. More precisely, such verification has used *simulation correspondence*. Simulation correspondence for microprocessor implementations can be described as follows.

Definition 9.1 (Simulation Correspondence). A microarchitectural model **ma** satisfies *simulation correspondence* with the corresponding instruction set architecture **isa** if there exists a binary relation *sim* with the following properties.

- *sim*(**ma**.*init*( ), **isa**.*init*( ))
- If *ma* and *isa* are two states such that *sim*(*ma*, *isa*) holds, then

S. Ray, *Scalable Techniques for Formal Verification*, DOI 10.1007/978-1-4419-5998-0_9,

1. **ma**.*label*(ma) = **isa**.*label*(isa)
2. For all i there exists some i' such that *sim*(**ma**.*next*(ma, i), **isa**.*next*(isa, i')) holds

The relation *sim* is then referred to as a *simulation relation*.

The two conditions above imply that for every execution of **ma** there is a matching execution of **isa** having the same corresponding *label*s. Assume that *mstimulus* is an arbitrary function so that *mstimulus*(n) is the stimulus provided to **ma** at time n. Then the above conditions guarantee that there exists some function *istimulus* so that the following is a theorem.

Theorem `simulation-sufficient`
ma.*label*(**ma**.*exec*[*mstimulus*](n)) = **isa**.*label*(**isa**.*exec*[istimulus](n)).

This is trace containment, and in fact there is no necessity for stuttering.

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 not necessarily vice versa. In microprocessor verification, one often uses a specific simulation relation that is based on the so-called "projection functions." In this approach, one defines a function *proj* such that, given a **ma** state ma, *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 showing the correctness of **ma** can be written as follows.

Theorem `projection-sufficiency`
proj(**ma**.*next*(ma, i)) = **isa**.*next*(*proj*(ma, *pick*(ma, i))).

The theorem is shown more often as a diagram such as in Fig. 9.1. This can be cast as a simulation proof by defining the simulation relation to be the following.

Definition.
sim(ma, isa) = (*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? If **ma** is pipelined, when one instruction completes execution, others

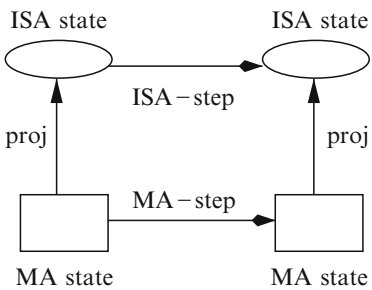

Fig. 9.1 Pictorial representation of simulation proofs using projection

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* [33, 241]. 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 [2]. 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 [150]. 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 Chap. 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 [38] as an approach to compare **ma** states with **isa** states, where the **ma** now is a pipelined machine. The notion is shown pictorially in Fig. 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 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

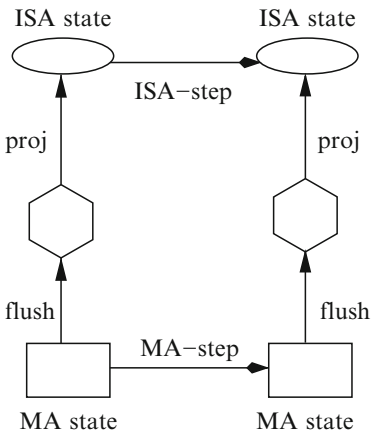

Fig. 9.2 Pictorial representation of flushing proofs

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 microarchitectural 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 Fig. 9.2, unlike the diagram in Fig. 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 flushing operation is defined as a component of the state label. Indeed, as shown by Manolios [150], 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. 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 Sect. 9.4 and discuss ways of adapting the approach to advanced pipelines in Sect. 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 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 Sect. 9.4. Finally, the correspondence between ma'_1 and isa' follows by noting that **ma** states ma_1 and ma'_1, and ISA states *isa* and isa' satisfy the flushing diagram of Fig. 9.2.

The above 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

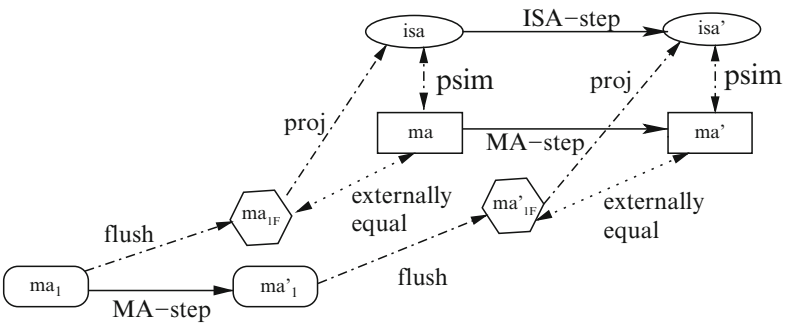

Fig. 9.3 Using flushing to obtain a refinement theorem

[1] Note that *ma* and ma_{1F} would possibly have different values of the program counter (PC). This is normally addressed by excluding the program counter from the *label*s of a state.

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 predicate 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 by the following definition.

Definition 9.2 (Stuttering Simulation Refinement). We say that $\mathscr{I}$ is a stuttering *simulation refinement* of $\mathscr{S}$, written $(\mathscr{S} \trianglerighteq_s \mathscr{I})$, if and only if there exist functions *psim*, *commit*, and *rank* such that the following are theorems:

1. $\mathit{psim}(\mathscr{I}.\mathit{init}(), \mathscr{S}.\mathit{init}())$
2. $\mathit{psim}(s, s') \Rightarrow \mathscr{I}.\mathit{label}(s) = \mathscr{S}.\mathit{label}(s')$
3. $\exists j : \mathit{psim}(s, s') \wedge \mathit{commit}(s.i) \Rightarrow \mathit{psim}(\mathscr{I}.\mathit{next}(s, i), \mathscr{S}.\mathit{next}(s', j))$
4. $\mathit{psim}(s, s') \wedge \neg\mathit{commit}(s, i) \Rightarrow \mathit{psim}(\mathscr{I}.\mathit{next}(s, i), s')$
5. $\mathit{o\text{-}p}(\mathit{rank}(s))$
6. $\mathit{psim}(s, s') \wedge \neg\mathit{commit}(s, i) \Rightarrow \mathit{rank}(\mathscr{I}.\mathit{next}(s, i)) \prec \mathit{rank}(s)$

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 $\mathit{psim}(s, s') = (\mathit{inv}(s) \wedge \mathit{rep}(s) = s')$, and $\mathit{commit}(s, i) = \neg(\mathit{skip}(s, i))$ we can easily show that if $(\mathscr{S} \trianglerighteq \mathscr{I})$, then $(\mathscr{S} \trianglerighteq_s \mathscr{I})$. For turning flushing proofs of pipelined machines to refinement proofs, we find it convenient to prove (**isa** $\trianglerighteq_s$ **ma**) rather than (**isa** $\trianglerighteq$ **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 Rule.
Derive $(\mathscr{S} \rhd \mathscr{I})$ from $(\mathscr{S} \trianglerighteq_s \mathscr{I})$

The proof of this rule follows the same concepts as the proof of well-founded refinements, *e.g.*, to show that for every execution of $\mathscr{I}$ there is a *trace* of $\mathscr{S}$ with the same visible behavior up to stuttering.

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 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 developed in the last chapter are quite adequate in practice for reasoning about microarchitectures.

Remark 9.1. 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

We 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 five-stage pipeline shown in Fig. 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 four 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

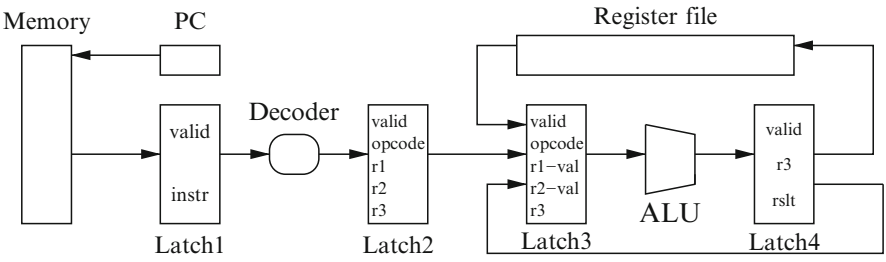

Fig. 9.4 A simple five-stage pipeline

program order until they complete. Every latch has a `valid` bit to indicate if the latch is nonempty. We use a three-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 one-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 (**isa** $\trianglerighteq_s$ **ma**). We define *commit* so that *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 $next_i(ma, inst)$ "runs" instruction *inst* for the first i stages of the pipe and updates the ith latch. For example, $next_2(ma, inst)$ updates latch 2 with the decoded value of the instruction. In addition, we define two functions, *flush* and *stalled*. Given a pipeline state *ma*, *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 *proj*(*ma*) projects the PC, memory, and register file of *ma* to the ISA.

Theorem `Burch-and-Dill` is a formal rendition of the flushing diagram for pipelines with stalls and can be easily proved using symbolic simulation.

Theorem `Burch-and-Dill`

$$proj(flush(\mathbf{ma}.next(ma))) = \begin{cases} \mathbf{isa}.next(proj(flush(ma))) & \text{if } \neg stalled(ma) \\ proj(flush(ma)) & \text{otherwise} \end{cases}$$

We now define *witnessing state*.

Definition 9.3 (Witnessing State). Given states ma_1 and *ma*, ma_1 is a *witnessing state* of *ma*, recognized by the predicate $witness(ma_1, ma)$, 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.

Definition.
$\mathit{psim}(ma, isa) = (\exists ma_1 : (\mathit{witness}(ma_1, ma) \wedge (isa = \mathit{proj}(\mathit{flush}(ma_1)))))$.

Now we show how the predicate *psim* can be used to prove (**isa** $\trianglerighteq_s$ **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 theorem.

Theorem `commit-property`
1. $\mathit{witness}(ma_1, ma) \wedge \neg\mathit{commit}(ma) \Rightarrow \mathit{witness}(ma_1, \mathbf{ma}.\mathit{next}(ma))$
2. $\mathit{witness}(ma_1, ma) \wedge \mathit{commit}(ma) \wedge \mathit{stalled}(\mathbf{ma}.\mathit{next}(ma_1)) \Rightarrow \mathit{witness}(\mathbf{ma}.\mathit{next}(\mathbf{ma}.\mathit{next}(ma_1)), \mathbf{ma}.\mathit{next}(ma))$
3. $\mathit{witness}(ma_1, ma) \wedge \mathit{commit}(ma) \wedge \neg\mathit{stalled}(\mathbf{ma}.\mathit{next}(ma_1)) \Rightarrow \mathit{witness}(\mathbf{ma}.\mathit{next}(ma_1), \mathbf{ma}.\mathit{next}(ma))$

The theorem 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.

Theorem `pipeline-simulation`
1. $\mathit{psim}(ma, isa) \wedge \neg\mathit{commit}(ma) \Rightarrow \mathit{psim}(\mathbf{ma}.\mathit{next}(ma), isa)$
2. $\mathit{psim}(ma, isa) \wedge \mathit{commit}(ma) \Rightarrow \mathrm{psim}(\mathbf{ma}.\mathrm{next}(ma), \mathbf{isa}.\mathit{next}(isa))$

Proof. Let ma_1 be the Skolem witness of $\mathit{psim}(ma, isa)$. Case 1 follows from Theorem `commit-property`, 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 Theorem `Burch-and-Dill` and the definition of *witness*, we have the property $\mathit{proj}(\mathit{flush}(\mathbf{ma}.\mathit{next}(ma_1))) = \mathbf{isa}.\mathit{next}(isa)$. The result now follows from Theorem `commit-property`. □

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 compositional.

We end this section with a brief comparison between our approach and that of Manolios [150]. 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 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 nontrivial. Indeed, in recent work [152], 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

[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**.

states. The predicate *witness* cannot 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 $\mathit{rep}(ma) = \mathit{proj}(\mathit{flush}(ma))$, 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 Theorem `commit-property` 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.

Definition.
$\mathit{clk}(\mathrm{s}) = \mathit{if}\ \neg \mathit{stalled}(s)\ \mathit{then}\ 0\ \mathit{else}\ 1 + \mathit{clk}(\mathbf{ma}.\mathit{next}(s))$.

9.5.2 Interrupts

Modern pipelines allow interrupts and exceptions. To effectively reason about interrupts, we model both MA and **isa** as nondeterministic 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 nondeterministic 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 (**isa** $\trianglerighteq_s$ **ma**) for such nondeterministic 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 simultaneously 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.

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, nondeterministically 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 nondeterministically, 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.

Note 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." Recall 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. [2] provide an excellent survey and comparison of the different techniques. Some of the early studies have used *skewed abstraction functions* [33, 241] to map the states of the pipeline at different moments to a single **isa** state. Burch and Dill [38] 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 [31]. Sawada uses a variant called *flush point alignment* to verify a three-stage pipelined machine [226]. The same approach has been used by Sawada and Hunt [227–229] 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. [107] use another 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 [1]. 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 [115] 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 [36, 136], whereby decision procedures have been used with light-weight theorem proving to verify state invariants.

Manolios [150] shows logical problems with the Burch and Dill criterion as a notion of correctness. He uses WEBs to reason about variants of Sawada's three-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 Sect. 9.4. Manolios and Srinivasan [155, 156] present ways of automating WEB proofs of pipelines by translating them to formulas in UCLID [37].

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 joint paper [210].

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 the application of stuttering refinements, as we saw in the examples in Chap. 8, was in the definition and proof of appropriate inductive invariants. In this part, we will therefore explore ways of facilitating invariant proofs.

Definition 10.1 (Invariant). Let $\mathscr{I}$ be a reactive system, and *stimulus* be a fixed uninterpreted unary function specifying a sequence of inputs to system $\mathscr{I}$. Then *good* is an *invariant* if and only if $\mathit{good}(\mathscr{I}.\mathit{exec}[\mathit{stimulus}](n))$ is a theorem. We write $\mathscr{I}.\mathit{exec}(n)$ as a shorthand for $\mathscr{I}.\mathit{exec}[\mathit{stimulus}](n)$.

An informal way of thinking of the above definition is as follows. 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. 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 inductive invariants in Chap. 8. There we said that one could prove that $\mathscr{I}$ is a well-founded refinement of $\mathscr{S}$ if one could prove the single-step obligations by replacing *inv* with predicate *good* if we can then prove that the following formulas are theorems.

RI1: $\mathit{inv}(\mathscr{I}.\mathit{init}())$
RI2: $\mathit{inv}(s) \Rightarrow \mathit{inv}(\mathscr{I}.\mathit{next}(s,i))$
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 has been of interest to the formal verification community for a long time independent of our framework. Proofs of *safety properties* (*i.e.*, those that say that the "system does not do anything bad" [142]) essentially amount to the proof of an invariant. Even

S. Ray, *Scalable Techniques for Formal Verification*, DOI 10.1007/978-1-4419-5998-0_10, 149

for proving liveness properties (*i.e.*, 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 proving 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 Chap. 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. Consequently, 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 $\mathsf{G}\,\mathit{good}$. Of course, then we need to think of the system as a Kripke Structure and assume that the set $\mathscr{AP}$ 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 $\mathsf{G}\,\mathit{good}$ holds for the system. The model checking approach, when it is applicable, 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 the so-called *predicate abstractions* [90], 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 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 not found an effective way of achieving such extensions. The source of the problem is the nondeterministic nature of reactive systems. The reason that symbolic simulations "worked" for sequential programs is that the cutpoints were relatively few. Recall that we had to manually add assertions to cutpoints for our symbolic simulation to work. If 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 cutpoints. 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, *e.g.*, 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 nondeterministic programs, but we could not adapt them suitably for our work [66]. 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 [90], although the basic approach is an instance of the idea of *abstract interpretation* introduced by Cousot and Cousot [61]. Suppose we have a system $\mathscr{I}$ that we want to analyze. For applying predicate abstractions, first a finite number of predicates is defined over the states of $\mathscr{I}$. These predicates are then used to define a finite-state system $\mathscr{A}$ that can serve as an abstraction of $\mathscr{I}$.

How does this all work? The definition of predicate abstraction below will suggest a construction.

Definition 10.2 (Predicate Abstraction). Let $\mathscr{P} \doteq \{P_0, \ldots, P_{m-1}\}$ be a set of predicates on the states of a system $\mathscr{I}$, $N_1, \ldots, N_{m-1}$ be a collection of functions.

Suppose that we can construct N_j such that the following is a theorem for $j \in \{0, \ldots, m-1\}$.

Predicate Update Condition.
$N_j(P_0(s), \ldots, P_{m-1}(s), i_0, \ldots, i_l) \Rightarrow P_j(\mathscr{I}.\mathit{next}(s, i))$.

Then a predicate abstraction of $\mathscr{I}$ is the finite-state system constructed as follows.

- A state of $\mathscr{A}$ 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 $\mathscr{A}.\mathit{init}()$ if and only if $b_j = P_j(\mathscr{I}.\mathit{init}())$.
- Given a state $\bar{s} \doteq \langle a_0, \ldots, a_{m-1} \rangle$ of $\mathscr{A}$, and an input $\bar{i} \doteq \langle i_0, \ldots, i_l \rangle$, the jth component of $\mathscr{A}.\mathit{next}(\bar{s}, \bar{i})$ is given by $N_j(a_0, \ldots, a_{m-1}, i_0, \ldots, i_l) \Leftrightarrow \mathtt{T}$.
- The *label* of a state $\bar{s}$ is the set of components in $\bar{s}$ that have the value `T`.

In the definition, it is useful to think of the collection of functions N_j as stipulating how each predicate is "updated" in system $\mathscr{I}$.

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 $\mathscr{I}$ is one of the predicates in $\mathscr{P}$. In fact, without loss of generality, assume $P_0 \doteq \mathit{good}$. It is easy to prove that *good* is an invariant of $\mathscr{I}$ if in every state p in $\mathscr{A}$ reachable from $\mathscr{A}.\mathit{init}()$, the 0th component of p is `T`. But since $\mathscr{A}$ 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 $\mathscr{P}$ 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 $\mathscr{P} \doteq \{\mathit{good}\}$ and $N_0(s) = \mathtt{NIL}$ to create a "bogus" abstract system $\mathscr{A}_B$ which is a predicate abstraction of any system $\mathscr{I}$ according to the above conditions but is useless for proving invariance of *good*.

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 $\mathscr{P}$?
Abstraction Given $\mathscr{P}$, how do we construct the abstract transition function?

Let us take the **abstraction** step first. Thus, suppose one has a set $\mathscr{P} \doteq \{P_0, \ldots, P_{m-1}\}$ 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 $\mathscr{I}$ the predicates in $\mathscr{P}$ have the values as specified by b, then is there some i such that in $\mathscr{I}.\mathit{next}(s, i)$ 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 $\mathscr{A}$. In practice, the answer is typically obtained by theorem proving [62, 74, 90, 225].

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 [139] 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 [52,63] for generating effective predicate abstractions incrementally. Another promising approach developed by Namjoshi and Kurshan [184] 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 Discussion

Given the success and popularity of predicate abstractions, it makes sense to ask if we can make use of such work 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 [138] and Qadeer and Flanagan [74] allows 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 [90] 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 Chap. 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

We will provide a technical description of our procedure 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 *C0*(s) returns the value of the component `C0` in state s and *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 *natp*(*C0*(s)) is an invariant. It should be clear, however, that since the value of `C0`

sometimes depends on the previous value of C1, $natp(C0(s))$ is not an inductive invariant. An inductive invariant for this system is given by the following definition.

Inductive Invariant.
$inv(s) = natp(C0(s)) \wedge natp(C1(s))$

Thus, we must "discover" the new predicate $natp(C1(s))$.

We will now see how our method handles this simple system. For technical reason, 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 $\mathscr{I}$ and a state component C, assume that $C(s)$ returns the value of C in state s. Then, we can use the following definition.

Definition.
$\widehat{C}(n) = C(\mathscr{I}.exec[stimulus](n))$.

Recall that *stimulus* is a fixed uninterpreted function stipulating an arbitrary infinite sequence of inputs to the system $\mathscr{I}$. With these conventions, equations 1–4 in Fig. 10.1 shows the equations which stipulate the behavior of our example system as a function of time. Writing $P_0 \doteq natp(\widehat{C0}(n))$, 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 $natp(\widehat{C0}(n+1))$ using equation 3 along with the following equation which is a generic theorem about *natp* and *if*.

Theorem natp-if
$natp(if(x, y, z)) = if(x, natp(y), natp(z))$.

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 $natp(\widehat{C0}(n+1))$ yields the following term.

Definition.
1. $\widehat{C0}(0) = 0$
2. $\widehat{C1}(0) = 0$
3. $\widehat{C0}(n+1) = if \neg stimulus(n)\ then\ \widehat{C0}(n)\ else\ \widehat{C1}(n)$
4. $\widehat{C1}(n+1) = if \neg stimulus(n)\ then\ 42\ else\ \widehat{C1}(n)$

Fig. 10.1 Equations showing the transitions of the two-component system

[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.

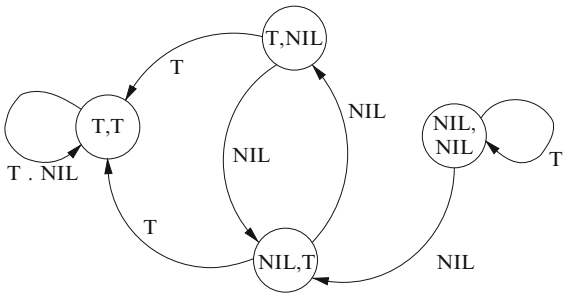

Fig. 10.2 Finite-state abstraction of the two-component system

Rewritten Term.
T0: $\mathit{if}(\mathit{stimulus}(n+1), \mathit{natp}(\widehat{C0}(n)), \mathit{natp}(\widehat{C1}(n)))$.

We will treat this term **T0** as a Boolean combination of terms $\mathit{stimulus}(n+1)$, $\mathit{natp}(\widehat{C1}(n))$, and $\mathit{natp}(\widehat{C0}(n))$. Let us decide to abstract the term $\mathit{stimulus}(n+1)$ and explore the term $P_1 \doteq \mathit{natp}(\widehat{C1}(n))$ 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 $\mathit{natp}(\widehat{C1}(n+1))$ using equations 4 and 5, along with the computed fact $\mathit{natp}(42) = \mathtt{T}$ then yields the following term.

Rewritten Term.
T1: $\mathit{if}(\mathit{stimulus}(n+1), \mathit{natp}(\widehat{C1}(n)), \mathtt{T})$

We treat terms **T0** and **T1** as specifying how the predicates P_0 and P_1 are "updated" at every instant. We now create our finite-state abstract system $\mathscr{A}$ as follows:

- The states of the system are pairs of Booleans. Thus, the system has four 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 $\mathit{natp}(\widehat{C0}(0))$ and $\mathit{natp}(\widehat{C1}(0))$.
- 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 0th component in $\mathscr{A}.\mathit{next}(p, i)$ will be given by the value of the (ground) term $\mathit{if}(i, b_0, b_1)$.

The system is shown pictorially in Fig. 10.2. It is easy to see that $\mathscr{A}$ is a predicate abstraction of our system. We now prove our invariant by checking that the 0th component of every reachable state is 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 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 chapter, we will answer these questions in the affirmative by designing a procedure and demonstrating its use in reasonably complex systems.

10.5 Bibliographic Notes

The idea of predicate abstraction was suggested by Graf and Saidi [90], and it forms an instance of the theory of *abstract interpretation* [61]. Many verification tools for both software and hardware have built predicate abstraction and discovery methods. Notable among software verification tools are SLAM [11], BLAST [101], and ESC/Java [66]. Abstraction techniques in both SLAM and BLAST use Boolean programs, which are formed by replacing the control predicates of the program with Boolean variables [10], and these predicates are then iteratively refined. UCLID [37] has developed predicate abstraction mechanisms where the abstract states and transitions are represented symbolically, to take advantage of algorithms for Boolean satisfiability checking [137,139]. Predicate discovery has involved refinement-based techniques based on model checking counterexamples [52,63] and syntactic manipulation of the concrete program [140,184].

This part is collaborative research with Rob Sumners [214,247,248].

Chapter 11
Predicate Abstraction via Rewriting

The example in Chap. 10, 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 $\mathscr{T}$, and produces a term τ^* as follows. Any formula in $\mathscr{T}$ 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 $\mathscr{C}$ to τ until no rule is applicable. The result is then said to be in *normal form*. Rewriting is, in general, a nondeterministic 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 $\mathscr{T}$ are applied in a predetermined total order. The

[1] The formulas of the form $\alpha = \beta$ are treated as rewrite rules $\mathtt{T} \Rightarrow \alpha = \beta$.

S. Ray, *Scalable Techniques for Formal Verification*, DOI 10.1007/978-1-4419-5998-0_11, 159

theory $\mathscr{T}$, as always, is assumed to be an extension of the ACL2 ground zero theory GZ. In particular, $\mathscr{T}$ 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 $\mathscr{T}$, if rewriting τ produces τ^*, then $(\tau = \tau^*)$ is a theorem in $\mathscr{T}$. This is the only critical property of REWRT that will be relevant 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 $\mathscr{T}$ 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.

Axiom.
$\mathit{hide}(x) = x$
$\mathit{force}(x) = x$

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 Fig. 11.1. Procedure CHOP extracts predicates from a rewritten term τ^* produced by REWRT. CHOP takes a term τ^* (assumed to have a single variable n) and recursively explores the top-level *if-then-else* structure of τ^*, collecting the non-*if* subterms, which are then classified as either *exploration predicates* $\mathscr{E}$ (which correspond to state variables in the abstract system) or *abstraction predicates* $\mathscr{U}$ (which are replaced with free input). In our example, CHOP, given the term **T0**, classified $\mathit{natp}(\widehat{C1}(n))$ as an exploration predicate and $\mathit{stimulus}(n+1)$ as an abstraction predicate. The basis for the classification is very simple. If the term τ given to CHOP is of the form $\mathit{if}(\tau_1, \tau_2, \tau_3)$, then it recursively CHOPs each of these

If τ is of the form $\mathit{if}(\tau_1, \tau_2, \tau_3)$
  $\langle \mathscr{E}_1, \mathscr{U}_1 \rangle :=$ CHOP(τ_1)
  $\langle \mathscr{E}_2, \mathscr{U}_2 \rangle :=$ CHOP(τ_2)
  $\langle \mathscr{E}_3, \mathscr{U}_3 \rangle :=$ CHOP(τ_3)
Return $\langle \mathscr{E}_1 \cup \mathscr{E}_2 \cup \mathscr{E}_3, \mathscr{U}_1 \cup \mathscr{U}_2 \cup \mathscr{U}_3 \rangle$
Else If $(n+1)$ or *hide* occurs in τ then
    Return $\langle \emptyset, \{\tau\} \rangle$
  Else Return $\langle \{\tau\}, \emptyset \rangle$

Fig. 11.1 Chopping a term τ. Here $\emptyset$ is the empty set

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 describing 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.

The top-level predicate discovery procedure is called DISCOVER. The procedure for predicate discovery is described in Fig. 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 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 $P/[n \rightarrow (n+1)]$, 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 $\mathscr{A}$. It has 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 is $\{i_0, \ldots, i_l\}$. 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 Fig. 11.2 that for each state variable v, the term $P_v/[n \rightarrow (n+1)]$ is rewritten in step 2 to produce

Initially $old := \{P_0\}$; $new_s := \emptyset$; $new_i := \emptyset$
Repeat
1. Choose some predicate $P \in old$
2. REWRT $P/[n \rightarrow n+1]$ to obtain term P^*
3. $\langle \mathscr{E}, \mathscr{U} \rangle :=$ CHOP(P^*)
4. $old := (old \backslash \{P\}) \cup \mathscr{E}$;
5. $new_s := new_s \cup \{P\}$;
6. $new_i := new_i \cup \mathscr{U}$

Until $old == \emptyset$
Return $\langle new_s, new_i \rangle$

Fig. 11.2 Procedure DISCOVER for generation of predicates

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 $\mathscr{A}$ 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 $x \in \{\mathtt{T}, \mathtt{NIL}\}$, 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 $\mathscr{A}$, 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 $\mathscr{T}$:

- $I/[n \to 0]$
- $I \Rightarrow P_0$
- $I \Rightarrow I/[n \to (n+1)]$

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 Proposition 11.1 specifies the relation between the minterms of an abstract state and the minterms of its successors.

Proposition 11.1. *Let a be any m-tuple. Then the following formula is a theorem of $\mathscr{T}$*

$$M_a \Rightarrow (\bigvee_{b \in nbrs_a} M_b)/[n \to (n+1)].$$

Proof. 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 Fig. 11.2 produces P' then $(P/[n \to (n+1)]) \Leftrightarrow P'$ 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. Proposition 11.2 shows that we can use the minterms for the states in R_a to construct I.

Proposition 11.2. *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 11.1. □

From Proposition 11.2, it follows that we can construct I above as follows. Let $\hat{a}$ be the initial state of $\mathscr{A}$. Then $I \doteq (\bigvee_{q \in R_{\hat{a}}} M_q)$. Indeed, from our description in Chap. 10, it is easy to see that the abstract system $\mathscr{A}$ forms a predicate abstraction. The differences between $\mathscr{A}$ 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 Fig. 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 Chap. 10, Theorem `natp-if` is critical to rewrite P_0' to **T0**. Otherwise, the normal form would have been the following.

Alternative Normal form.
T0′ : $\mathit{natp}(\mathit{if}(\mathit{stimulus}(n), \widehat{C0}(n), \widehat{C1}(n)))$

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 treatment given to the function *hide*. To see its use, consider a system with components `A0`, `A1`, `A2`, etc., where `A0` is specified as follows.

Definition.
$\widehat{A0}(0) = 1$
$\widehat{A0}(n+1) = \mathit{if}\ \mathit{natp}(\widehat{A1}(n))\ \mathit{then}\ 42\ \mathit{else}\ \widehat{A0}(n)$

Thus, `A0` is assigned 42 if the previous value of `A1` is a natural number, and otherwise is unchanged. Consider proving that $P_0 \doteq \mathit{natp}(\widehat{A0}(n))$ is an invariant. Our procedure will discover the exploration term $P_1 \doteq \mathit{natp}(\widehat{A1}(n))$ and attempt to rewrite $\mathit{natp}(\widehat{A1}(n+1))$, 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 following rewrite rule.

Theorem `hide-A1`
$\mathit{natp}(\widehat{A1}(n)) = \mathit{hide}(\mathit{natp}(\widehat{A1}(n)))$

Since *hide* is logically the identity function, proving the formula above is trivial. But the rule has the effect of "wrapping" a *hide* around $\mathit{natp}(\widehat{A0}(n))$; 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 Sect. 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, in our example of Chap. 10, we could have added the following rule.

Theorem `force-C1`
$\mathit{natp}(\widehat{C1}(n)) = \mathit{force}(\mathit{natp}(\widehat{C1}(n)))$

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 one variable instead of two as shown in Fig. 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 [28], SMV [165], and NuSMV [47]. 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 monograph, 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 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 specified as follows.

Predicates.
$P_v \doteq (f(n) = g(n))$
$P_{i_0} \doteq (f(n) = i(n + 1))$
$P_{i_1} \doteq (g(n) = i(n + 1))$

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 Part V, 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 Chap. 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 Chap. 8. In that chapter, we proved *good* to be an invariant by manually constructing an inductive invariant *inv*. We now use our tool 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 eight 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 [132], 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.

Theorems.
$in(e, insert(a, s)) = in(e, s) \vee (a = e)$
$in(e, drop(a, s)) = in(e, s) \wedge \neg(a = e)$
$get(a, put(b, v, r)) = if\ (a = b)\ then\ v\ else\ get(a, r)$

Fig. 11.3 Generic rewrite rules for set and record operations

Theorem `in1-hide`

$$\mathit{in1}(e,s) = \begin{cases} \texttt{NIL} & \mathit{if\ empty}(s) \\ (e = \mathit{choose}(s)) & \mathit{if\ singleton}(s) \\ \mathit{hide}(\mathit{in1}(e,s)) & \text{otherwise} \end{cases}$$

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.

Our procedure now proves the invariance of *good* (restated in terms of time). The abstract system has nine 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 Fig. 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. Note that tracking which process is relevant at which point has traditionally been a difficult problem for fully automatic decision procedures. This underlines the importance of using an expressive logic for defining predicates.

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. $(D(n) = \mathit{get}(A(),\widehat{\mathit{cache}}(\mathit{choose}(\widehat{\mathit{excl}}(\mathit{cline}(A()),n)),\mathit{cline}(A()),n)))$

Fig. 11.4 State predicates discovered for the **ESI** model

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 [6, 137, 205]. The original implementation has single-entry channels. In UCLID, *indexed predicates* were used [138] 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:

- $\widehat{cache}(p, c, n)$ is the content of line `c` in the cache of client `p` at time n.
- $\widehat{valid}(c, n)$ is the set of clients having a copy of line `c` at time n.
- $\widehat{excl}(c, n)$ is the set of clients that 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:

- $\widehat{h\text{-}valid}(c, n)$ is the set of clients that have access to line `c` at time n.
- $\widehat{h\text{-}excl}(c, n)$ is the client which has exclusive access to line `c` at time n.
- $\widehat{curr\text{-}cmd}(c, n)$ is the pending request for line `c` at time n.
- $\widehat{curr\text{-}client}(c, n)$ is the most recent client requesting for line `c` at n.
- $\widehat{mem}(c, n)$ is the value of line `c` in the memory at time n.
- $\widehat{invalid}(c, n)$ 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):

- $\widehat{ch1}(p, c, n)$ is the requests sent from client p for line c at time n.
- $\widehat{ch2\text{-}sh}(c, n)$ is the set of clients with a shared fill response in channel 2.
- $\widehat{ch2\text{-}ex}(c, n)$ is the set of clients with an exclusive fill response in channel 2.
- $\widehat{ch2\text{-}data}(p, c, n)$ is the data sent to client p with fill responses.
- $\widehat{ch3\text{-}data}(p, c, n)$ 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 shown in Fig. 11.3; we also reuse the "system specific" concept `in1-hide` 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 `in1-hide` but for record operations in order to cause a case-split on $invalid(c, n)$. With these rules, our procedure proves 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 7,000 abstract states and about 300,000 edges, and the proof is completed in less than 2 min on an 1.8 GHz Pentium desktop machine running GNU/Linux. The proof of the bounded channel invariant completed in less time with 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 Part V, our tool has been used by Sumners (private communication) to prove the relevant invariants of the concurrent deque of Chap. 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.

The predicate abstraction tool is not a panacea. As with any deductive approach, it can fail for some systems and reachability analysis will then 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-hide` 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 [184]. 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 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 Kurshan also suggest many of the features we implemented, such as *if-lifting* transformations. No implementation of the approach is available, making it difficult to compare their method with ours. Nevertheless, 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 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 [138], 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 Fig. 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 [9]. 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 [54] also uses a logic based on rewriting to specify next-state functions of computing systems.

Predicate abstraction has had several applications in invariant proofs; the bibliographic notes for Chap. 10 list many papers on the subject. Our implementation above is closely related to the approach suggested by Namjoshi and Kurshan [184]. A comparison of our approach with theirs appears in Sect. 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 et al. [205]. They use a method of *invisible invariants* to simplify the proof of the system. Emerson and Kahlon [72] 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 [137–139] for verification of this protocol, both for bounded and unbounded channel sizes.

Part V
Formal Integration of Decision Procedures

Chapter 12
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 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

S. Ray, *Scalable Techniques for Formal Verification*, DOI 10.1007/978-1-4419-5998-0_12, 175

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 procedures operate. To understand this, let us consider a decision procedure $\mathscr{D}$ that takes three positive rational numbers m, n, and ε, and checks if $|\sqrt{m} - n| \leq \varepsilon$. Thus, it (rightly) passes the check on the triple of number $\langle 2, 1, 1\rangle$. Suppose now we are in a theory $\mathscr{T}$ and encounter the affirmative answer given by $\mathscr{D}$ on $\langle 2, 1, 1\rangle$. How do we interpret this answer? We might be tempted to extend $\mathscr{T}$ by introducing a new unary function *sqrt*, so that *sqrt*(x) can be interpreted as the positive square root of x (i.e., a formalization of $\sqrt{x}$), and then infer, based on the answer provided by $\mathscr{D}$, that $|$*sqrt*$(2) - 1| \leq 1$. Unfortunately, if $\mathscr{T}$ is a legal extension of GZ, then the function *sqrt* cannot be introduced such that the formula *sqrt*$(2) \times$ *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 [78, 79]. Thus, any extension of GZ with a reasonable axiomatization of *sqrt* that lets us interpret the affirmative answer of $\mathscr{D}$ above is inconsistent.

How do we resolve this dilemma? One way might be to simply invoke $\mathscr{D}$ carefully on arguments for which we know how to interpret the answer. For instance, even though we cannot interpret the answer returned from $\mathscr{D}$ 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 Chap. 13 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 the 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 $\mathscr{D}$ above, and prove the following formula as a theorem about it.

Characterization Theorem.
$(\varepsilon = 0) \wedge \textit{approx-sqrt}(m, n, \varepsilon) \Rightarrow m = n \times n$

This theorem 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.

The approach seems simple as a matter of course. But the problem of coming up with characterization theorems, let alone proving them, is nontrivial. 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 Chap. 13. Our experience indicates that it is possible to do it, although some aspects in defining a formal semantics of model checking are nontrivial. 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 the 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 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. Indeed, these considerations have been some of the key motivations in the design of a generic interface between ACL2 and external deduction tools. We discuss the interface in Chap. 14. The design of the interface provides some insight into the kind of consideration that is involved in extending an industrial-scale reasoning system with a nontrivial feature.

Chapter 13
A Compositional Model Checking Procedure

Model checking is one of the most widely used verification methods 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 Chap. 2. As mentioned before, the practical limitation of model checking in practice arises from state explosion. Our approaches in Parts III and IV involved determining techniques for manageable decomposition of the verification problems. As our discussions throughout this monograph 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, etc. 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 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.

Our simple compositional procedure works as follows. 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

S. Ray, *Scalable Techniques for Formal Verification*, DOI 10.1007/978-1-4419-5998-0_13, 179

of these verification problems. That is, for the ith 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 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.

13.1 Formalizing a Compositional Model Checking 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.

13.1.1 Finite State Systems

We have been talking about models of reactive systems for a while in this monograph. We modeled a system $\mathscr{I}$ as three functions, namely $\mathscr{I}.\mathit{init}()$, $\mathscr{I}.\mathit{next}()$, and $\mathscr{I}.\mathit{label}$. 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 $\mathscr{A}$, which represented a predicate abstraction of the implementation. The point of interest here is that $\mathscr{A}$ has a finite number of states. As we saw, we can represent such a system by three components.

- A collection of state variables.
- A collection of input variables.
- For each state variable, a term specifies 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 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 *sysp* so that *sysp*(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.

13.1.2 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 Sect. 2.2. 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.

13.1.3 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* below.

Definition.

$$\mathit{R\text{-}and}(\psi, AP) = \begin{cases} \mathit{list}(\psi) & \mathit{if}\ \neg \mathit{andformula}(\psi, AP) \\ \mathit{append}(\mathit{R\text{-}and}(\mathit{left}(\psi), AP), \mathit{R\text{-}and}(\mathit{right}(\psi), AP)) & \text{otherwise} \end{cases}$$

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 *cone of influence* C of state variables according to the following definition.

Definition 13.1 (Cone of Influence). Let M be a system, ψ be an LTL formula, and V be the set of state variables mentioned in ψ. A set C of state variables is called a *cone of influence* of a system M with respect to a formula ψ if the following conditions hold:

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.

Given the cone of influence C, we 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 $\mathit{R\text{-}cone}(\psi, M)$ 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 Fig. 13.1. Given an LTL formula ψ and a finite state system M, it performs the following steps:

Definition.

$$\mathit{R\text{-}1}(fs, M, \mathscr{A}\mathscr{P}) = \begin{cases} \texttt{NIL} & \mathit{if}\ \neg \mathit{consp}(fs) \\ \mathit{cons}(\mathit{prob}(\mathit{first}(fs), \mathit{R\text{-}cone}(M, \mathit{first}(fs)), \mathscr{A}\mathscr{P}), & \\ \quad \mathit{R\text{-}1}(\mathit{rest}(fs), M, \mathscr{A}\mathscr{P})) & \text{otherwise} \end{cases}$$

$$\mathit{Reduce}(\psi, \mathscr{M}, \mathscr{A}\mathscr{P}) = \mathit{R\text{-}1}(\mathit{R\text{-}and}(\psi), \mathscr{M}, \mathscr{A}\mathscr{P})$$

Fig. 13.1 Formalization of a compositional model checking procedure. Here, $\mathit{prob}(f, \psi, AP)$ is assumed to create the verification problem $\langle f, \psi, AP \rangle$

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 $\mathit{ltlsem}(\psi, \kappa, AP)$ 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 *satisfy* below.

Definition.
$\mathit{satisfy}(\psi, M, AP) = \mathit{ltlsem}(\psi, \mathit{kripke}(M), AP)$

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 *satisfy* ψ. This notion is formalized by the predicate *passes* below.

Definition.
$\mathit{passes}(prb) = \mathit{satisfy}(\mathit{formula}(prb), \mathit{sys}(prb), \mathit{ap}(prb))$

Finally, a collection of verification problems will be said to *pass* if and only if each constituent problem *passes*, as specified by the definition below.

Definition.

$$\mathit{pass}(prbs) = \begin{cases} \mathtt{T} & \text{if } \neg \mathit{consp}(prbs) \\ \mathit{passes}(\mathit{first}(prbs)) \wedge \mathit{pass}(\mathit{rest}(prbs)) & \text{otherwise} \end{cases}$$

With these definitions, we can finally express the correctness of our compositional procedure. The formal statement of correctness is shown below.

Correctness Characterization.
$\mathit{sysp}(M, AP) \wedge \mathit{formula}(\psi, AP) \Rightarrow \mathit{satisfy}(\psi, M, AP) = \mathit{pass}(\mathit{Reduce}(\psi, M, AP))$

13.2 Modeling LTL Semantics

We seem to have our work cut out to achieve the goal of reasoning about our procedure, for example, we should define the predicate *ltlsem* to capture the semantics of LTL and then prove the formula named **Correctness Characterization** above as a theorem. But how do we define *ltlsem*? We have presented the standard semantics of LTL in Sect. 2.2. LTL 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 *pathsem*(ψ, π, 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, *pathp*(π, κ) returns T if π is a path through κ and NIL otherwise. If we can do all that, then we can then define *ltlsem* with the following definition.

Possible Definition.
ltlsem$(\kappa, \psi, AP) = (\forall \pi : \textit{pathp}(\pi, \kappa) \Rightarrow \textit{pathsem}(\psi, \pi, AP))$

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 `len-unbounded`
$\exists y : \textit{len}(y) > \textit{len}(x)$

The theorem says that for any list x, there is some other list y that has a larger length. So there is no infinite list. Indeed, we have found no way of representing infinite paths as formal objects that can be manipulated by ACL2 functions in any theory constructed by extending GZ via extension principles other than defaxiom.

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 introduced in Part II. This function could be *interpreted* as an infinite sequence of input stimuli, so that we could think of *stimulus*(n) as the member of this sequence at position n. But the ACL2 logic is first order, meaning that 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) *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 however, encapsulation cannot be used to do the "trick." Why? Consider the definition schema of M.*exec*[*stimulus*] that we talked about in Chap. 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 cannot be expressed as a formula in the ACL2 logic. Thus, we seem to have hit the limits of ACL2's expressiveness.

It might seem that the answer to the above riddle is an augmentation of GZ with some axiomatization of infinite sequences. After all, ACL2 provides its defaxiom construct for this purpose. However, it is subtle to add nontrivial axioms in a sound manner. Furthermore, even with an axiomatization of infinite sequences, modeling the semantics of LTL would require permitting definitions containing both recursion and quantification. To see why, imagine for the moment that we have axioms for infinite objects and *suffix*(i, π) returns π_i. Consider the recursive characterization of $\mathsf{F}\psi$ we discussed in Sect. 2.2 (point 7), which we reproduce below:

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:

Structure of Definition.

$$\mathit{pathsem}(\psi, \pi, AP) = \begin{cases} \ldots \\ \exists i : \mathit{pathsem}(\mathit{value}(\psi), \mathit{suffix}(i, \pi), AP) \text{ if } \mathit{op}(\psi) = \mathsf{F} \\ \ldots \end{cases}$$

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 Sect. 13.4). 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, defining *ltlsem* directly is tantamount to defining a model checking algorithm for LTL. From informal description of the model checking algorithm in Chap. 2, it is such that 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*. An eventually periodic path is simply a path that comprises of a finite *prefix* followed by a finite *cycle* that is repeated forever. Since 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 Sect. 2.2 so that $\mathit{ppathsem}(\psi, \pi, AP)$ returns $\mathtt{T}$ 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.

Definition.
$\mathit{ltlsem}(\psi, \kappa, AP) = (\forall \pi : \mathit{ppathp}(\pi, \kappa) \Rightarrow \mathit{ppathsem}(\psi, \pi, AP))$

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 Proposition 13.1. 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* [49]. We provide the proof here, for the sake of completeness.

Proposition 13.1. *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. Recall from Chap. 2 that the problem of checking whether ψ satisfies κ can be reduced to checking if the language $\mathscr{L}$ accepted by the Büchi automaton $\mathscr{A}_{\kappa,\psi}$ is empty. Let us assume that κ does not satisfy ψ, and hence there must be an accepting (infinite) sequence ρ for $\mathscr{A}_{\kappa,\psi}$. 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, $\mathscr{L}$ 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 $\mathscr{L}$ is nonempty then there is a counterexample that 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. □

Remark 13.1. The definition of the function *ppathsem* is not trivial. We still have 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 ith suffix of π. Then, instead of writing $\exists i$: ***ppathsem***(***value***(ψ), ***psuffix***$(i, \pi), AP)$ in the recursive call, we write ***ppathsem***(***value***(ψ), ***psuffix***(***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.

13.3 Verification

We now discuss how to prove that the **Correctness Characterization** Theorem above is indeed 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 ***prbs*** below so that given a finite state system M, the atomic propositions A, and a list of formulas $fs \doteq \langle \psi_1, \ldots, \psi_k \rangle$, ***prbs***$(fs, M, A)$ creates a list of verification problems $\langle \psi_i, M, A \rangle$, one for each ψ_i.

Definition.

$$\textit{prbs}(fs, M, A) = \begin{cases} \texttt{NIL} & \textit{if } \neg \textit{consp}(fs) \\ \textit{cons}(\textit{prob}(\textit{first}(fs), M, A), \textit{prbs}(\textit{rest}(fs), M, A)) & \text{otherwise} \end{cases}$$

The following obligation 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.

Theorem `conjunctive-correct`
$\textit{sysp}(M, A) \wedge \textit{formula}(\psi, A) \Rightarrow \textit{satisfy}(\psi, M, A) = \textit{pass}(\textit{prbs}(\textit{R-and}(\psi), M, A))$

The correctness of the cone of influence reduction is specified analogously by the obligation `cone-correct`, which states that M satisfies ψ if and only if the reduced model of M with respect to the variables in ψ satisfies ψ.

Theorem `cone-correct`
$\textit{sysp}(M, A) \wedge \textit{formula}(\psi, A) \Rightarrow \textit{satisfy}(\psi, M, A) = \textit{satisfy}(\psi, \textit{R-cone}(\psi, M), A)$

The proof of the formula labeled **Correctness Characterization** from Theorems `conjunctive-correct` and `cone-correct` is straightforward. The function ***Reduce*** replaces the "system" component of each verification problem in ***prbs***(***R-and***$(\psi), M, A)$ with the reduced model of M with respect to variables in the corresponding LTL formula. To prove **Correctness Characterization** we note that (by `conjunctive-correct`) M satisfies ψ if each of the problems in ***prbs***(***R-and***$(\psi), M, A)$ passes, and (by `cone-correct`) a problem $\langle \psi, M, A \rangle$

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 Theorems `conjunctive-correct` and `cone-correct`? The proof of `conjunctive-correct` is simple. The following sketch follows its mechanical derivation, with some commentaries added for clarity.

Recall from the definition of the semantics of LTL (Sect. 2.2), 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.[1] By induction over the structure of the formula, it 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. This implies that 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, it follows that M satisfies ψ if and only if M satisfies each ψ_i.

The crux of the verification is to prove the theorem `cone-correct` 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 explain the complications, we first review the standard approach to carrying out this proof.

The traditional proof of cone of influence reduction uses an argument based on bisimulation. Following is the definition of bisimulation for Kripke Structures.

Definition 13.2 (Bisimulation). 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 A 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 $B(s_1, s'_1)$.

We call the Kripke Structures κ and κ' *bisimulation equivalent* if and only if there exists some bisimulation predicate B such that $B(s_0, s'_0)$.

Let π and π' be two paths starting from s and s' in κ and κ', respectively. We call π and π' *corresponding paths* if and only if $L(\pi[i])$ is the same as $L'(\pi'[i])$ for each i. Proposition 13.2 is a standard result about bisimulations.

Proposition 13.2. *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.*

[1] 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.

Furthermore, by structural induction on the definition of semantics of LTL, we can now deduce Proposition 13.3.

Proposition 13.3. *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.*

Proposition 13.4 follows from Propositions 13.2 and 13.3 and is the crucial result we will use.

Proposition 13.4. *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 13.4.

Let us now attempt to turn the argument above into a formal proof. The key is to formalize Proposition 13.2. 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 13.2. 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 construction of a path π is similar. □

The argument above is very simple. Indeed, in Chap. 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 them 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 Fig. 13.2, 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

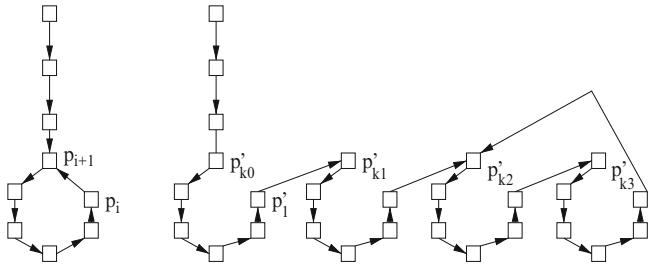

Fig. 13.2 A periodic path and its match

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 (nonempty) 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 nontrivial, to formalize. The rest of the proof of the `cone-correct` theorem follows the traditional structure outlined above, for example, by formalization of Propositions 13.3 and 13.4.

13.4 A Deficiency of the Integration and Logical Issues

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 **Correctness Characterization** Theorem, and most of the time was spent on devising and formalizing 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.

The integration, however, has one serious drawback. 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 explained above, this makes the proof of correctness of the reduction procedure complicated. Furthermore, to successfully using theorem proving on a substantially complex problem, one must 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 $\mathscr{T}$ 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 $\mathscr{S}$ and $\mathscr{I}$, the theorem $(\mathscr{S} \rhd \mathscr{I})$ 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 $\mathscr{T}$ in which we wanted to prove our theorem to a new (extended) theory $\mathscr{T}'$. How do we then know that the formula we wanted to prove is still a theorem in $\mathscr{T}$? This is guaranteed by conservativity. More precisely, a (first order) theory $\mathscr{T}'$ is a *conservative* extension of $\mathscr{T}$ if and only if for any formula Φ that is expressible in $\mathscr{T}$, Φ is (first order) provable in $\mathscr{T}'$ if and only if it is also (first order) provable in $\mathscr{T}$. Kaufmann and Moore [129] prove that the extension principles of ACL2 do have this property. Conservativity 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 $\mathscr{T}$ 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 a formula that is expressible in GZ (and any extension). Thus, by conservativity, `NIL` is provable in some theory $\mathscr{T}$ if and only if `NIL` is provable in GZ. Thus, conservativity reduces the consistency of arbitrary ACL2 theories to the consistency of GZ. Furthermore, conservativity is the crucial property that permits several proof structuring mechanisms in ACL2.

Conservativity is, thus, 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 Fig. 13.3 which would have been possible had ACL2 allowed it.

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*. However, it is well-known that given this "definition" one can prove by induction based on the term *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,

$$\textit{true-p}(\Phi, \sigma, \textit{dfns}) \triangleq \begin{cases} \textit{true-exists-p}(\textit{formals}(\Phi), \textit{body}(\Phi), \sigma, \textit{dfns}) & \textit{if existsp}(\Phi) \\ \textit{true-forall-p}(\textit{formals}(\Phi), \textit{body}(\phi), \sigma, \textit{dfns}) & \textit{if forallp}(\Psi) \\ \textit{evaluate-term}(\Phi, \sigma, \textit{dfns}) & \text{otherwise} \end{cases}$$

$$\textit{true-exists-p}(\textit{fs}, \Phi, \sigma, \textit{dfns}) \triangleq (\exists \textit{val} : \textit{true-p}(\Phi, \textit{map}(\textit{fs}, \textit{val}, \sigma, \textit{dfns})))$$

$$\textit{true-forall-p}(\textit{fs}, \Phi, \sigma, \textit{dfns}) \triangleq (\forall \textit{val} : \textit{true-p}(\Phi, \textit{map}(\textit{fs}, \textit{val}, \sigma, \textit{dfns})))$$

Fig. 13.3 A truth predicate. 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-term*** returns the value of Φ under the assignment σ. 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

by Gödel's Incompleteness Theorem [81, 82], 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.

13.5 A Possible Remedy: Integration with HOL4

As a result of the above argument, it is not possible to extend the definitional principle of ACL2 to allow quantification in recursive definitions. However, the example underlines the need to augment the logic reasoning framework offered by ACL2 to facilitate definition of the semantics of LTL. One approach that has received a lot of attention recently concerns the possibility of integrating ACL2 with the HOL4 theorem prover [87]. HOL4 [118] is a theorem prover for higher-order logic, based on Church's simply typed lambda calculus augmented with Hindley–Milner polymorphism [89]. It is sufficiently powerful to allow one to define functions using recursion, much in the way that is allowed by ACL2. The integration is aimed to allow the user to use the expressiveness of HOL together with the automation and fast execution capabilities of ACL2. The basic idea of the link is based on the definition of the standard ACL2 universe as a HOL datatype, `sexp`, defined below. The recursive HOL datatype definition of `sexp` below uses previously defined HOL types `packagename`, `name`, `string`, `char`, and `complex_rational`. In the definition of the HOL datatype `sexp` below, the types given after `=>` indicate the types of the arguments of the constructors being declared. For example, `ACL2_PAIR` is a (curried) constructor of type `sexp -> sexp -> sexp`.

```
sexp = ACL2_SYMBOL    of packagename => name
     | ACL2_STRING    of string
     | ACL2_CHARACTER of char
     | ACL2_NUMBER    of complex_rational
     | ACL2_PAIR      of sexp => sexp
```

The key observation is the following. Consider a formula φ that is proven in an ACL2 theory $\mathscr{T}$. Then φ holds in the standard model of $\mathscr{T}$, and hence in the HOL theory obtained by extending `sexp` with the (suitably translated) definitions from $\mathscr{T}$.

How could the above integration help in remedying the deficiencies above? Recall that the proof of correctness of cone of influence reduction cleanly separates into two parts, for example, (1) generic properties on bisimulation and the relation between bisimulation equivalence and LTL semantics and (2) the proof that cone of influence reduction preserves bisimulation equivalence. The traditional correctness proofs of the generic properties of bisimulation (Propositions 13.2 and 13.3) are well-suited for formalization in higher-order logic. The connection between ACL2 and HOL, thus, permits the verification of the executable ACL2 definition of the algorithm based on the semantics defined in HOL, exploiting the existing ACL2 proof showing that cone of influence reduction preserves bisimulation, with a standard proof of Propositions 13.2 and 13.3 formalized in HOL4. Indeed, as this book goes to press, such a mechanized proof has been completed. We believe that the connection will facilitate proofs of characterization theorems for other more complex decision procedures. Nevertheless, the connection between ACL2 and HOL4 is nascent and the jury is still out on its robustness and suitability for such applications.

13.6 Summary and Discussion

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 problems, is viable. While our proof of the cone of influence reduction is complicated, the complication arises mainly from the limitations in ACL2's expressiveness that 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.

A 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 nonstandard 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 [71]. It would be useful to integrate such procedures with ACL2. But this is not possible using the current

formulation. However, we believe that the HOL4 integration may provide a robust solution to this problem.

Another key problem involves efficiency. 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. On the other hand, our formalization of the model checker, for example, 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 nonconstructive, it cannot be executed.

The considerations above suggest that it is important to look for ways for integrating external decision procedures (or *oracles*) with ACL2. Developing a general mechanism for such integration will be the subject of Chap. 14.

13.7 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 [45] present a proof of correctness of partial order reductions using HOL. A formalization of TLA in HOL was created by von Wright [253]. Schneider and Hoffmann [231] present a proof of reduction of LTL to ω-automata in HOL. In ACL2, Manolios [151] models a μ-calculus model checker, and proves that it returns a fixpoint.

The general approach of proving formulas in a theory $\mathscr{T}$ as theorems in some (possibly bigger) theory $\mathscr{T}'$ embedding a proof system for $\mathscr{T}$ in $\mathscr{T}'$ is referred to as *reflection*. Reflection is getting increasingly popular in the theorem proving community, and an excellent paper by Harrison [98] provides an extensive survey of the key results in the area. Some of the early uses of embeddings include the work of Weyhrauch [256], which allowed the evaluation of constants in the FOL theorem prover by attaching programs. Metafunctions [22] 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 Chap. 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 [212], and have been incorporated here with permission from the coauthors.

Chapter 14
Connecting External Deduction Tools with ACL2

At the end of Chap. 13, we briefly broached the question of efficiency of decision procedures integrated with ACL2. Our formalization of the model checker, for example, the definition of *ltlsem*, while useful for logical reasoning, was inadequate for execution. One way to resolve this problem is to define an executable model checker for LTL in ACL2, prove that it is equivalent to *ltlsem*, and then use the executable model checker to check the individual verification problems. Defining an LTL model checker involves a complicated tableau construction as we saw in Chap. 2. Formalizing the construction is possible, although nontrivial, 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 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 above discussion suggests a more general problem. Recent years have seen rapid advancement in the capacity of automatic reasoning tools, in particular for decidable theories such as Boolean logic and Presburger arithmetic. For instance, modern BDD packages and satisfiability solvers can automatically solve problems with tens of thousands of variables and have been successfully used to reason about commercial hardware system implementations [165, 181]. The mechanized theorem proving community ought to take advantage of this advancement by developing connections with automatic reasoning tools. Can we develop an interface connecting theorem provers with external reasoning tools?

S. Ray, *Scalable Techniques for Formal Verification*, DOI 10.1007/978-1-4419-5998-0_14, 195

In this chapter, we present one solution to this problem, at least in the context of ACL2: we develop an interface for connecting ACL2 with tools that are external to ACL2's built-in reasoning routines.

It is nontrivial to establish a sound connection between ACL2 and other tools. Recall that ACL2 contains several logical constructs intended to facilitate effective proof structuring [129]. These constructs complicate the logical foundations of the theorem prover. To facilitate connection between another tool and ACL2, it is therefore imperative (1) to determine the conditions under which a conjecture certified by a combination of the theorem prover and the tool is indeed a theorem and (2) to provide mechanisms that enable a tool implementor to meet these conditions.

The interface described in this chapter permits an ACL2 user to invoke an external tool to reduce a goal formula C to a list of formulas L_C during a proof attempt. Correctness of the tool involves showing that the provability of each formula in L_C (in the logic of ACL2) implies the provability of C. We present a sufficient condition (expressible in ACL2) that guarantees such provability claims and discuss the logical requirements on the implementor of external tools for sound connection with ACL2. The interface design illustrates some of the subtleties and corner cases that need to be considered in augmenting an industrial-strength formal tool with a nontrivial feature.

We distinguish between two classes of external tools, namely (1) tools verified by the ACL2 theorem prover and (2) unverified but trusted tools. A verified tool must be formalized in the logic of ACL2 and the sufficient condition alluded to above must be formally established by the theorem prover. An unverified tool can be defined using the ACL2 programming interface and can invoke arbitrary executable programs via a system call interface. An unverified tool is introduced with a *trust tag* acknowledging that the validity of the formulas proven using the tool depends on the correctness of the tool.

14.1 Verified External Tools

We start with a description of our interface to connect *verified* external tools with ACL2. The ideas and infrastructure we develop here will be extended in the next two sections to support unverified tools.

We will refer to external deduction tools as *clause processors*. Recall that ACL2 internally represents terms as clauses, so that a subgoal of the form $\tau_0 \wedge \tau_1 \wedge \cdots \wedge \tau_n \Rightarrow \tau$ is represented as a disjunction by the list $(\neg\tau_0\ \neg\tau_1\ \cdots\ \neg\tau_n\ \tau)$. Our interface enables the user to transform the current clause with custom code. More generally, a *clause processor* is a function that takes a clause C (together with possibly other arguments) and returns a list L_C of clauses.[1] The intention is that if each

[1] The definition of a clause processor is somewhat more complex. In particular, it can optionally take as argument the current state of ACL2 and return, in addition to a clause list, an error message and a new state. We ignore such details here.

clause in L_C is a theorem of the current ACL2 theory then so is C. When we talk about clause processors, we will mean such clause manipulation functions.

Our interface for verified external tools consists of the following components:

A new rule class for installing clause processors. Suppose the user has defined a function *tool0* that she desires to use as a clause processor. She can then prove a specific theorem about *tool0* (described below) and attach this rule class to the theorem. The effect is to install *tool0* in the ACL2 database as a clause processor for use in subsequent proof attempts.

A new hint for using clause processors. Once the function *tool0* has been installed as a clause processor, it can be invoked via this hint to transform a conjecture during a subsequent proof attempt. If the user instructs ACL2 to use *tool0* to help prove some goal G, then ACL2 transforms G into the collection of subgoals generated by executing *tool0* on (the clause representation of) G.

We now motivate the theorem alluded to above for installing *tool0* as a clause processor. Recall that theorems in ACL2 can be thought of in terms of evaluation: a formula Φ is a theorem if, for every substitution σ mapping each free variable of Φ to some object, the instance Φ/σ does not evaluate to `NIL`. Let C be a clause whose disjunction is the term τ, and let *tool0*, with C as its argument, produce the list `(`C_1 ... C_n`)` that represents the conjunction of corresponding terms $\tau_1, \ldots, \tau_n$. (That is, each τ_i is the disjunction of clause C_i.) Informally, we want to ensure that if τ/σ evaluates to `NIL` for some substitution σ then there is some σ' and i such that τ_i/σ' also evaluates to `NIL`.

The condition is made precise by extending the notion of evaluators from terms to clauses. We discussed evaluators in Chap. 3 while discussing the logic of the ACL2 theorem prover. Assume that the ACL2 Ground Zero theory GZ contains two functions *disjoin* and *conjoin** axiomatized as shown in Fig. 14.1. Informally, the axioms specify how to interpret objects representing clauses and clause lists; for instance, *disjoin* specifies that the interpretation of a clause `(`τ_0 τ_1 τ_2`)` is the same as that of `(if` τ_0 `T (if` τ_1 `T (if` τ_2 `T NIL)))`, which represents the disjunction of τ_0, τ_1, and τ_2. Let *ev* be an evaluator for a set of function symbols that includes the function *if*; thus *ev*(*list*(`if`, τ_0, τ_1, τ_2), a) stipulates how the term "*if* τ_0 *then* τ_1 *else* τ_2" is evaluated. For any theory $\mathscr{T}$, a clause processor function *tool0*(C, `args`) will be said to be *legal* in $\mathscr{T}$ if there exists a function *tool0-env* in

Axioms.

disjoin(C) = *if* ¬*consp*(C) *then* `*NIL*`
else list(`if`, *car*(C), `*T*`, *disjoin*(*cdr*(C)))

*conjoin**(L_C) = *if* ¬*consp*(L_C) *then* `*T*`
else list(`if`, *disjoin*(*car*(L_C)), *conjoin**(*cdr*(L_C)), `*NIL*`)

Fig. 14.1 Axioms in GZ for supporting clause processors. `*T*` and `*NIL*` are assumed to be the internal representation of `T` and `NIL`, respectively. The predicate *consp* is defined in GZ such that *consp*(x) returns `T` if x is an ordered pair and `NIL` otherwise

Proof Obligation.
pseudo-term-listp$(C) \wedge$
alistp$(a) \wedge$
ev(*conjoin**(*tool0*$(C, \texttt{args})$), *tool0-env*$(C, a, \texttt{args})$)
$\Rightarrow$ *ev*(*disjoin*$(C), a)$

Fig. 14.2 Correctness obligation for clause processors. Function *ev* is an evaluator for a set of functions that includes *if*; `args` represents the remaining arguments of *tool0* (in addition to clause C); *pseudo-term-listp* is a predicate axiomatized in GZ that returns `T` if its argument is an object in the ACL2 universe representing a list of terms (therefore a clause); and *alistp*(a) returns `T` if a is an association list, otherwise `NIL`

$\mathscr{T}$ such that the formula shown in Fig. 14.2 is a theorem.[2] The function *tool0-env* returns an association list like σ' in our informal example above: it potentially modifies the original association list to respect any generalization performed by *tool0*. We note that a weaker theorem logically suffices, replacing *tool0-env*$(c, a, \texttt{args})$ with an existentially quantified variable.

The obligation shown in Fig. 14.2 (once proven) can be associated with the new rule class for recognizing clause processors. This instructs ACL2 to use the function *tool0* as a verified external tool. Theorem 14.1 guarantees that the obligation is sufficient. An analogous argument justifies ACL2's current meta reasoning facilities, which appears as a comment titled "Essay on Correctness of Meta Reasoning" in the ACL2 source code.

Theorem 14.1. *Let $\mathscr{T}$ be an ACL2 theory for which tool0 is a legal clause processor, and let tool0 return a list L_C of clauses given an input clause C. If each clause in L_C is provable in $\mathscr{T}$, then C is also provable in $\mathscr{T}$.*

Proof. The theorem is a simple consequence of Lemma 14.1, taking τ_i to be v_i and given the obligation shown in Fig. 14.2 along with the definitions of *conjoin** and *disjoin*. □

Lemma 14.1. *Let τ be a term with free variables $v_0, \ldots, v_n$, ev an evaluator for the function symbols in τ, and e a list of cons pairs of the form $(\langle {}'v_0, {}'\tau_0\rangle \ \ldots \ \langle {}'v_n, {}'\tau_n\rangle)$, where ${}'v_i$ and ${}'\tau_i$ are internal representation of v_i and τ_i, respectively. Let σ be a substitution mapping each v_i to τ_i, and let ${}'\tau$ be the internal representation of the term τ. Then the following formula is provable: ev$({}'\tau, e) = \tau/\sigma$.*

Proof. The lemma follows by induction on the structure of term τ. □

In spite of the simplicity of the proof, the statement of Theorem 14.1 is perhaps more subtle than it appears. Note that the theorem restricts the use of a clause processor to *a theory in which the clause processor is legal*. This restriction might appear

[2] For the curious reader, the ACL2 function *pseudo-term-listp* in Fig. 14.2 is so named since it actually only checks if its argument has the *syntactic structure* of a list of terms, for example, it does not check that the functions in the "terms" are defined and called with the correct number of arguments. The reasons for using this predicate are technical and not germane to this paper.

too strong by the following flawed reasoning. We know that each extension principle in ACL2 produces a conservative extension. Also, the function symbols occurring in formulas simplified by the application of a clause processor might not necessarily occur in the definition of the clause processor itself. For instance, consider a clause processor called *generalize* that replaces each occurrence of the object `'(f x)` [which is the internal representation of the term $f(x)$] in a clause with a new free variable. Note that although *generalize* manipulates the internal representation of *f*, the function *f* itself never appears in the formal definition of *generalize*. Thus by conservativity, a theorem proven by the application of *generalize* is valid in a theory in which *generalize* is not defined (e.g., *generalize* does not add any new axiom about *f*), and hence one should be able to mark the definition of *generalize* (and any definition or theorem involving the definition of *generalize*) local in spite of being used in a hint as a clause processor. But this is precluded by Theorem 14.1.

To see the flaw in the reasoning above, consider defining a clause processor *simplify-foo* that replaces terms of the form *foo*(τ) by τ. Suppose a book, `book1`, contains the following event.

Local Definition.
foo$(x) = x$

We can add *simplify-foo* as a legal clause processor in `book1`. Using an evaluator *ev-foo* for the function symbols *foo* and *if*, it is trivial to prove the obligation in Fig. 14.2 and thus install *simplify-foo* as a clause processor. Note that the definition of *foo* is local and also that the definition of *simplify-foo* only manipulates the internal representation $'(\textit{foo}\ \tau)$ of *foo*(τ); thus, the definition of *simplify-foo* does not contain any occurrence of the function symbol *foo*. However, suppose we permit the installation of *simplify-foo* (nonlocally) as a clause processor but mark the corresponding evaluator axioms for *ev-foo* to be local. Then we can create a book, `book2`, with the following events.

Local Definition.
foo$(x) = \textit{cons}(x, x)$
Include `book1`.

We now are in a theory in which the application of *tool0* is inconsistent with the current definition of *foo*, resulting in unsoundness.

The example illustrates some of the subtleties arising because of local events. The problem is that the evaluator *ev-foo* that justifies the use of *simplify-foo* as a verified clause manipulation function is an evaluator for the locally defined function *foo*; for instance, one of the evaluator axioms for *ev-foo* is the following.

Evaluator Axiom for *ev-foo*
ev-foo$(\textit{list}(\textit{foo}, x), a) = \textit{ev-foo}(\textit{foo}(x), a)$

This axiom and the local definition of *foo* are necessary in proving the obligation in Fig. 14.2. The legality requirement ensures that the current theory contains the above axiom, which in turn requires that the functions "interpreted" by the evaluator are nonlocal. We remark that the example above, though simple, illustrates

a soundness bug related to meta rules that existed in ACL2 for many years.[3] This bug resulted from the failure to track the fact that the theory in which the rules are applied is not properly supported by evaluators.

14.1.1 Applications of Verified External Tools

Verified clause processors enable an ACL2 user to bypass ACL2's heuristics and use customized, verified code on large-scale proofs in a specific domain. We now discuss some applications of these capabilities. The examples shown below illustrate the flexibility provided by the interface and serve as a guide for the reader interested in customizing ACL2 for efficiently reasoning about a specific domain.

14.1.1.1 Adding a Hypothesis

Our first example is the trivial clause processor *addHypothesis* below, which weakens a formula with a user-supplied hypothesis.

Definition.
$\mathit{addHypothesis}(C, \tau) = \mathit{list}(\mathit{cons}(\mathit{list}(\mathit{not}, \tau), C), \tau)$

If Φ_C is the formula represented by clause C, then applying the clause processor *addHypothesis* causes it to be replaced by the implication $\tau \Rightarrow \Phi_C$, together with a new proof obligation to show that τ is a theorem. The installation of *addHypothesis* as a clause processor, based on the proof obligations shown in Fig. 14.2, can be achieved by proving the theorem shown in Fig. 14.3, which is proven automatically by ACL2.

One application of the *addHypothesis* tool is to strengthen a formula before applying induction. For example, assume that we wish to prove $a \Rightarrow b$, and b can be proven by induction. One way to prove $a \Rightarrow b$ is to use the *addHypothesis* tool

Theorem `addhypothesis-clause-processor`
pseudo-term-listp$(C) \wedge$
alistp$(a) \wedge$
ev-ifnot(*conjoin**(*addHypothesis*$(C, \tau)), a)$
$Rightarrow$*ev-ifnot*(*disjoin*$(C), a)$

Fig. 14.3 Correctness theorem for the *addHypothesis* clause processor. Here, *ev-ifnot* is an evaluator for the function symbols *if* and *not*. Note that we have used *tool0-env*$(C, a, \tau) = a$

[3] We have checked that the bug exists as far back as in the December 1997 release, and probably it was never absent before its fix in the February 2006 release.

with $\tau := b$. This returns the trivially true subgoal $b \Rightarrow (a \Rightarrow b)$, together with the subgoal b which can now be dispatched by induction.

We should note that the *addHypothesis* clause processor, though useful, is merely pedagogical. Indeed, it is possible to mimic the action of *addHypothesis* with a certain hint available with ACL2. Nevertheless, its use illustrates how verified reasoning code enables the user to control proof search.

14.1.1.2 Equality Reasoning

Our next example is a clause processor that enables the user to control reasoning about a set of equalities. To motivate the use of a clause processor for equality reasoning, consider the following scenario. Let f and g be unary functions, and p be a unary predicate such that the formula $p(g(x), g(y))$ is a theorem. Consider proving the trivial theorem.

Theorem.
$(f(x) = g(x)) \wedge (f(y) = g(y)) \Rightarrow p(f(x), f(y))$

ACL2 cannot prove the above theorem automatically. To understand why, we explain a bit the heuristics involved in the term simplification performed by ACL2. In summary, ACL2 internally has a notion of the syntactic "complexity" of a term, and while simplifying terms that represent equality, it rewrites complex terms to simpler ones. In the above scenario, ACL2 determines the terms involving g to be more complex than the corresponding terms involving f, and thus simplification will attempt to rewrite in the "wrong" order, failing to prove the theorem.[4]

We can, however, bypass these limitations by implementing a clause processor *equalitySubstitution*.[5] Given a clause C and a list of n pairs of terms $\langle\langle \tau_1, \tau_1' \rangle \ldots \langle \tau_n, \tau_n' \rangle\rangle$, the clause processor produces $n + 1$ clauses as follows. The first clause is obtained by replacing each occurrence of τ_i in C with τ_i'. The remaining n clauses are formed by adding the term $(\tau_j = \tau_j')$ to C, for $j = 1, \ldots, n$; hence, if C represents the formula Φ_C, then the jth clause can be viewed as the subgoal $(\tau_j \neq \tau_j') \Rightarrow \Phi_C$. In our scenario, the use of *equalitySubstitution* with the list $\langle\langle f(x), g(x)\rangle \ \langle f(y), g(y)\rangle\rangle$ produces the following three trivial subgoals each of which can be proven automatically by ACL2.

Subgoals.

1. $(g(x) = g(x)) \wedge (g(y) = g(y)) \Rightarrow p(g(x), g(y))$
2. $(f(x) \neq g(x)) \Rightarrow ((f(x) = g(x)) \wedge (f(y) = g(y)) \Rightarrow p(f(x), f(y)))$
3. $(f(y) \neq g(y)) \Rightarrow ((f(x) = g(x)) \wedge (f(y) = g(y)) \Rightarrow p(f(x), f(y)))$

[4] ACL2 also has so-called "cross-fertilization" heuristics that temporarily allow substitution of more complex terms for simpler ones, but the heuristics fail to prove the theorem in the scenario described.

[5] The *equalitySubstitution* clause processor has been implemented and verified by Jared Davis.

The use of *equalitySubstitution* is an interesting example of the use of customized reasoning code to alleviate limitations in the simplification heuristics of the theorem prover. The correctness of *equalitySubstitution* merely depends on the correctness of substitution of equals and is straightforward to verify. On the other hand, it is tricky to craft general-purpose heuristics that automatically perform effective equality-based simplification on arbitrary terms given large databases of applicable theorems, in the presence of induction and other proof techniques outside the domain of decision procedures. Indeed, many ACL2 users over the years have been "stumped" by limitations in the equality reasoning heuristics; the clause processor frees the user from dependence on, and complicated work-arounds for, such limitations.

14.1.1.3 Sorting Bit Vector Addition

Our third example illustrates the use of verified clause processors to implement efficient simplification routines for domain-specific applications. The domain here is bit vector arithmetic, and we consider the problem of normalizing terms composed of bit vector additions. Such normalization is necessary in proofs involving arithmetic circuits, for instance to verify multiplier designs. Furthermore, although our focus is on bit vector addition, the general principle can be used in proofs involving any associative–commutative binary function.

Let $+_{32}$ be the binary function representing 32-bit bit vector addition, and consider the problem of proving the following theorem.

Theorem `bv-add-commutes`
$(a_1 +_{32} (a_2 +_{32} (\cdots + 32a_n))) = (a_n +_{32} (\cdots +_{32} (a_2 +_{32} a_1)))$

The standard approach to proving a theorem as above is to normalize both sides of the equality using a syntactic condition on the terms involved in the summands and finally show that the resulting normal forms are syntactically identical. To do this via rewriting (in the absence of built-in associative–commutative unification), one proves the associativity and commutativity of $+_{32}$ as oriented rewrite rules which are then repeatedly applied to reduce each side of the equality to normal form. A slight reflection would, however, indicate that such an approach is inefficient. In particular, since the rewriting in ACL2 is inside-out, the normalization requires $O(n^2)$ applications of the associativity and commutativity rules.

Our solution is to implement a clause processor *SortBVAdds* that sorts the summands in a term involving bit vector additions using an $O(n \log n)$ mergesort algorithm. The correctness of the clause processor is verified by showing that (1) mergesort returns a permutation of its input and (2) the application of a sequence of $+_{32}$ functions produces the same result on any permutation of the summands. With this clause processor, we can obtain significant efficiency in simplifying terms involving $+_{32}$. As empirical evidence of the efficiency, the equality theorem above can be proven in 0.01 s for 500 summands and in 0.02 s for 1,000 summands on a 2.6-GHz Pentium IV desktop computer with 2.0 GB of RAM; the naive rewriting strategy outlined above takes 11.24 s and 64.41 s, respectively.

We close the discussion on *SortBVAdds* with two observations. First, note that *SortBVAdds* sorts bit vector addition occurring within subexpressions of ACL2 formulas, not merely at the top level. For example, given any unary function f, it can be used to simplify $f(a_2 +_{32} a_1 +_{32} a_0)$ into $f(a_0 +_{32} a_1 +_{32} a_2)$; furthermore, such a function f can be introduced *after* the installation clause processor. Second, it might be possible to mimic the efficiency reported above using ACL2's metareasoning abilities. However, writing such a general-purpose efficient metafunction is awkward, since meta-rules are closely tied to ACL2's inside-out rewriter.

14.2 Basic Unverified External Tools

Verified clause processors enable the user to employ verified reasoning code for customized clause manipulation. However, the main motivation behind the development of our interface is to connect ACL2 with (unverified) tools that are external to the theorem prover, such as Boolean satisfiability solvers and model checkers. In this section and the next, we present mechanisms to enable such connections.

Our interface for unverified tools involves a new event that enables ACL2 to recognize a function *tool1* as an *unverified* clause manipulation tool. Here, *tool1* might be implemented using *program mode* and might also invoke arbitrary executable code using ACL2's system call interface. The effect of the event is the same as if *tool1* were introduced as a verified clause processor: hints can be used to invoke the function during subsequent proof search to replace a goal with a list of subgoals.

Suppose an unverified tool *tool1* is invoked to simplify a conjecture in the course of a proof search. What guarantees must the implementor of *tool1* provide (and the user of *tool1* trust) in order to claim that the conjecture is indeed a theorem? A sufficient guarantee is that *tool1* could have been formally defined in ACL2 together with appropriate evaluators such that the obligation shown in Fig. 14.2 is a theorem about *tool1*. The soundness of the use of *tool1* then follows from Theorem 14.1.

Since the invocation of an unverified tool for simplifying ACL2 conjectures carries a logical burden, the event introducing such tools provides two constructs, namely (1) a *trust tag*[6] for the user of the tool to acknowledge this burden and (2) a concept of *supporters* enabling the tool developer to guarantee that the logical restrictions are met. We now explain these two constructs.

The trust tag associated with the installation of an unverified tool *tool1* is a symbol (the name of the tool itself by default), which must be used to acknowledge that the applicability of *tool1* as a proof rule depends on *tool1* satisfying the logical guarantees above. The certification of a book that contains an unverified tool, or includes (hereditarily, even locally) a book containing an unverified tool, requires the user to tag the certification command with that tool's trust tag. Note that technically

[6] ACL2's trust tag mechanism, introduced in Version 3.1, is quite general, with applications other than to unverified clause processors. See the topic defttag in the ACL2 documentation [124].

the mere act of installing an unverified tool does not introduce any unsoundness; the logical burden expressed above pertains to the *use* of the tool. Nevertheless, we insist on tagging the certification of any book containing an *installation* of an unverified tool (whether subsequently used or not) for implementation convenience. Recall that the local incompatibility check (the second pass of a book certification) skips proofs, and thereby ignores the hints provided during the proof process. By "tracking" the installation rather than application of an unverified tool, we disallow the possibility of certifying a book that *locally* introduces and uses an unverified tool without acknowledging the application of the tool.

Finally, we turn to *supporters*. To understand this construct, recall the problem outlined in the preceding section, demonstrating inconsistency with a verified clause processor if the evaluator axioms could be local. The problem was that the tool *simplify-foo* had built-in knowledge about some function definitions and we wanted to ensure that when it is applied for clause manipulation the axioms in the current theory match this knowledge. In the verified case, this was guaranteed by permitting the use of the tool only in a theory in which the proof obligations for its legality are met. However, suppose we want to use *simplify-foo* as an *unverified* tool in the scenario described. Then there is no obligation proven (or stated) in the logic, other than the (perhaps informal) guarantee from the implementor that it *could* be proven in principle given an appropriate formal definition of *foo*.

The *supporters* enable a tool implementor to insist that the axiomatic events for all functions "understood" by the tool are present in any theory in which it is used. The implementor can list the names of such axiomatic events (typically function symbols that name their definitions, *e.g.*, *foo* in the example for *simplify-foo*) in the *supporters* field of the event installing unverified clause processors. We call these events *supporting events*. Whenever ACL2 encounters an event installing a function *tool1* as an unverified clause processor with a nonempty list of supporters, it checks that *tool1* and all of its supporting event names have already been introduced.

14.2.1 Applications of Unverified External Tools

The connection of ACL2 with external tools has been a topic of extensive interest, and there has been significant work connecting different decision procedures with the theorem prover. These include integration with the Cadence SMV model checker [212], Zchaff and Minisat SAT solvers [215], and UCLID [155]. These connections have been successfully used to automate large-scale verification tasks with ACL2. For instance, the UCLID connection has been used to automatically verify deep pipelined machines [156]. However, in the absence of the disciplined mechanisms that we present here, these connections have necessarily required substantial "hacking" of ACL2's source code. Our interface has been implemented to enable the ACL2 user to connect with tools such as those above without imposing

demands on understanding the inner workings of the theorem prover implementation. In particular, the interface is sufficient for connecting all the tools cited above.[7]

In addition to the above connections, the unverified clause processor interface has been used to implement a decision procedure for the Subclass of Unrollable List Formulas in ACL2 (SULFA) [215]. SULFA is a decidable subclass of ACL2 formulas that is composed of the primitives *if*, *consp*, *cons*, *car*, and *cdr*; a SULFA formula can also involve restricted applications of other primitives, as well as user-defined functions. The subclass is powerful enough to express finite step hardware properties specified by first-order definitions.

The SULFA decision procedure works as follows. A SULFA formula is translated into Boolean conjunctive normal form (CNF) after some (bounded recursive) functions are unrolled; the CNF formula is given to a SAT solver, such as zChaff: if SAT finds a counterexample, then it is converted into a counterexample to the original formula and presented to the user.

The SULFA clause processor has been applied in the verification of components of the TRIPS processor [39], a prototype multicore processor designed and built by the University of Texas at Austin and IBM. In particular, the Verilog implementation of the protocol used to communicate between the four load-store queues was formally verified. The use of the SULFA decision procedure makes it possible to combine SAT-based methods with theorem proving, achieving a higher level of automation than was previously possible with the ACL2 theorem prover [112, 215].

The SULFA clause processor has also been used to develop a general-purpose solver for the standard SMT theory of bit vectors; the solver performs preliminary simplification with ACL2 before invoking the SULFA clause processor to dispatch the resulting simplified problem. This tool has been used to verify fully automatically all the problems in the SMT benchmark suite [209]. We note that the close connection with ACL2 makes this SMT solver more flexible than any other solver we are aware of, albeit perhaps with some loss in efficiency. In particular, one can augment the solver with definitions of additional bit vector primitives and add theorems about such primitives; these theorems are proven correct with ACL2 and then applied to efficiently simplify terms that involve the relevant primitives.

14.3 Unverified External Tools for Implicit Theories

Our view so far of an unverified tool is that if it replaces some clause with a list of clauses, then the provability of the resulting clauses implies the provability of the original clause. Such a tool is thus an efficient procedure for assisting in proofs of theorems that could, in principle, have been proven from the axiomatic events

[7] The sufficiency has been ensured by significant discussions with the authors of these previous connections in the design of the interface. Furthermore, Sudarshan Srinivasan (private communication) is working on connecting an SMT solver with ACL2 and has expressed a desire to use our interface.

of the current theory. This simple view is sufficient in most situations. However, some ACL2 users have found the necessity to use more sophisticated tools that implement their own theories. In this section, we will discuss such tools and the facilities necessary to connect them with ACL2.

To motivate the need for such tools, assume that we wish to prove a theorem about some hardware design. Hardware designs are typically implemented in a Hardware Description Language (HDL) such as VHDL or Verilog. One way of formalizing such designs is to define a semantics of the HDL in ACL2, possibly by defining a formal interpreter for the language [30, 222]; however, the complexity of a modern HDL often makes this approach impractical in practice [85]. On the other hand, most commercial model checkers can parse designs written in VHDL or Verilog. A practical alternative is thus merely to *constrain* some properties of the interpreter and use a combination of theorem proving and model checking in the following manner:

- Establish low-level properties of parts of a design using model checkers or other decision procedures
- Use the theorem prover to compose the properties proven by the model checker together with the explicitly constrained properties of the interpreter to establish the correctness of the design

This approach has shown promise in scaling formal verification to industrial designs. For instance, in recent work, Sawada and Reeber [230] verify a VHDL implementation of an industrial floating-point multiplier using a combination of ACL2 and an IBM internal verification tool called SixthSense [172]. For this project, they introduce two functions, *sigbit* and *sigvec*, with the following assumed semantics:

- *sigbit*(e, s, n, p) returns a bit corresponding to the value of bit signal s of a VHDL design e at cycle n and phase p.
- *sigvec*(e, s, l, h, n, p) returns a bit vector corresponding to the bit range between l and h of s for design e at cycle n and phase p.

In ACL2, these functions are constrained only to return a bit and bit-vector, respectively. The key properties of the different multiplier stages are proven using SixthSense: one of the properties proven is that *sigvec* when applied to (1) a constant `C` representing the multiplier design, (2) a specific signal `s` of the design, (3) two specific values `lb` and `hb` corresponding to the bit-width of s, and (4) a specific cycle and phase, returns the sum of two other bit vectors at the previous cycle; this corresponds to one stage of the Wallace-tree decomposition implemented by the multiplier. Such properties are then composed using ACL2 to show that the multiplier, when provided two vectors of the right size, produces their product after 5 cycles.

How do we support this verification approach? Note that the property above is *not* provable from the constraints on the associated functions alone (namely *sigvec* returns a bit vector). Thus, if we use encapsulation to constrain *sigvec* and posit the property as a theorem, then functional instantiation can easily derive an inconsistency. The problem is that the property is provable from the constraints *together*

with axioms about *sigvec* that are unknown to ACL2 but assumed to be accessible to SixthSense.

Our solution to the above is to augment the extension principles of ACL2 with a new principle called *encapsulation templates* (or simply *templates*). Function symbols introduced via templates are constrained functions analogous to those introduced via the encapsulation principle, and the conservativity of the resulting extension is analogously guaranteed by exhibiting local witnesses satisfying the constraints. However, there is one significant distinction between the encapsulation principle and templates: the constraints introduced are marked *incomplete* acknowledging that they might not encompass all the constraints on the functions. ACL2, therefore, must disallow any functional instantiation that requires replacement of a function symbol introduced via templates.

With templates, we can integrate ACL2 with tools like SixthSense above. Suppose that we wish to connect ACL2 with a tool *tool1* that implements a theory not defined explicitly in ACL2. We then declare *tool1* to be an unverified clause processor by (1) providing a template to introduce the function symbols (say f and g) regarding which the theory of the clause processor contains additional axioms and (2) stipulating that the supporters of the clause processor include f and g.

We now explain the logical burden for the developer of such a connection. Assume that an ACL2 theory $\mathscr{T}$ is extended by a template event E and that the supporting events for *tool1* mention some function introduced by E. Then, the developer of *tool1* must guarantee that one can introduce f and g via the encapsulation principle, which we will refer to as the "promised" encapsulation E_P of the functions, such that the conditions 1–4 below hold. Note that the guarantee is obviously outside the purview of the mechanized reasoning system of ACL2; the obligations codify a set of sufficient conditions under which it is logically sound to use *tool1*:

1. The constraints in E_P include the constraints in E.
2. E_P does not introduce any additional function symbols other than those introduced by E.
3. E_P is admissible in theory $\mathscr{T}$.
4. *Tool1* is correct when E is interpreted as E_P, in the following sense. Suppose *tool1* is applied to a clause C to yield a list L_C of clauses, and at any such point, let $\mathscr{T}$ be the current ACL2 theory augmented with the axioms introduced by E_P. Then, if each clause in L_C is a theorem of $\mathscr{T}$, then C is a theorem of $\mathscr{T}$.

Condition 2 is necessary to ensure that E_P does not implicitly introduce additional function symbols with constraints that might conflict with the later axiomatization of these functions. The conditions above enable us to view any template event as representing its promised encapsulation.

We note that the interface does not permit the introduction of a template that is separated from the declaration of an unverified clause processor. One might think that this is too restrictive and there is no harm in introducing a template as a separate event, with the view that every theorem proven with ACL2 (without a tool) for the introduced functions is valid for any admissible promised encapsulation of the

template. That is, if a tool is later introduced supporting the template, such theorems would seem to be valid for the promised encapsulation of the tool as well. However, we disallow this possibility since it is possible to exploit such "dangling" templates to prove a contradiction.

As a simple example, consider the following scenario which would be possible without such a restriction. Suppose a tool developer develops a book `book1` consisting of the following sequence.

Template.
Introduce f with no explicit constraint.

Template.
Introduce g with no explicit constraint.

Unverified Clause Processor.
Define a tool tool1 *and add g as supporter, with promised encapsulation for g providing the following constraint: $g(x) = f(x)$*

Now suppose a (possibly different) tool developer develops a book `book2` consisting of the following sequence.

Template.
Introduce g with no explicit constraint.

Template.
Introduce f with no explicit constraint.

Unverified Clause Processor.
Define a tool tool2 *and add f as supporter, with promised encapsulation for f providing the following constraint: $f(x) = \neg g(x)$*

Thus, both `book1` and `book2` would be (independently) certifiable. Now imagine an ACL2 session that first includes `book1` and then `book2`. The inclusion of `book2` reintroduces the two template events, but since they are identical to the corresponding template events in `book1` we would not expect any complaint from ACL2. However, one could presumably prove a contradiction in this session using the conflicting promised encapsulations implemented by the two tools.

We can view the above example from a logical perspective, as follows. The logical foundation of ACL2 [129] associates each ACL2 session with a sequence of formal events (a so-called *chronology*) that captures its logical content. When a template is introduced together with a tool, that template can be viewed logically as its promised encapsulation; then no further change to the above association needs to be made. If, however, a template is separated from its tool, the logical view is complicated significantly. Indeed, the example above demonstrates that it is problematic to work out a logical view for templates whose promised encapsulations are delayed; in essence, there is no chronology associated with the above session.

We conclude our discussion of unverified external clause processors by saying a bit more about their logical view. As suggested above, the correspondence between ACL2 sessions and logical chronologies, as laid out in the treatment of ACL2's foundations, remains unchanged even for unverified external tools with their own

theories, by viewing such tools as their promised encapsulations. An important part of the logical view is the justification of local events in a book, since these are ignored when the book is included. Since the logical view remains unchanged with unverified external tools, it would seem to be sound for the ACL2 user to mark as local either the definition or use of such a tool, and to mark as local the inclusion of a book that has such definitions or uses. This soundness claim indeed holds if those external tools are correct, that is, if each deduction they make is sound with respect to the ACL2 theory present in which the deduction is made (see Condition 4 above). So, it is important to have a way of tracking which tools we are trusting to be correct. In Sect. 14.2, we touched on how the *trust tag* mechanism can track the introduction, and hence the use, of unverified clause processors. This tracking must be done (and is done) even through local events. But if all tracked external tools are correct, then our logical view tells us that it is indeed sound to remove local events. The use of trust tags when certifying and including books that contain unverified external tools is a small price for the ACL2 user to pay, since the use of local events is typically a key activity of ACL2 users.

14.4 Remarks on Implementation

Our presentation in the preceding three sections have primarily focused on the different flavors of clause processors from the point of view of a *user* interested in connecting ACL2 with other reasoning tools. In this section, we discuss some aspects of our implementation of these connections. The details of the various implementation considerations touched upon in this section are specific to the ACL2 system. However, we believe that they provide a sense of the issues involved in extending an industrial-strength theorem prover with a nontrivial feature. Note that an understanding and appreciation of some of the considerations discussed here will benefit from some exposure to the ACL2 theorem prover and perhaps some familiarity with ACL2's design architecture. The ACL2 user interested in further details is also encouraged to browse the book `books/clause-processors/basic-example.lisp` in the ACL2 distribution, which exercises several interesting corner cases of the implementation, in particular demonstrating approximately 80 helpful error messages.

14.4.1 Basic Design Decisions

ACL2 is a complex system with many primitive types of event commands (besides just definitions and theorems) and many implicit invariants. Augmenting such a system with a feature like an interface connecting external tools requires strict adherence to good software engineering practice, and significant care must be taken to ensure that the new feature does not interfere with the existing features in a

surprising way or break any of the design invariants. To minimize complexity, both for the implementation and the user experience, the design choices involved in the implementation of any new feature to the theorem prover are based on the following two guiding principles:

- Implement the new feature by reusing as much of the existing features and infrastructures as possible.
- Disallow complex corner cases that are unlikely to be experienced by the user in a typical scenario, adding support for such cases in a demand-driven fashion as a user encounters them in practice.

We now discuss how the architecture and implementation of our interface are affected by these two design principles.

First, recall that the interface for invoking clause processors is implemented as a new *hint*. Thus, one can invoke an external tool (verified or not) to simplify any subgoal that arises during a proof search, possibly a goal generated after several transformations to the user-supplied conjecture by the internal reasoning strategies of ACL2. This is in contrast with the implementation of similar features in HOL and Isabelle [93, 182], where the connection is provided by a special-purpose construct that "ships off" a user-supplied conjecture to an external tool. Our choice is partially motivated to support the application of external tools to simplify subgoals arising in proofs, but it also allows the use of ACL2's fairly mature hint mechanism to be reused to support the connection. The hint mechanism enables the user to provide the theorem prover with pragmatic advice on proof search. Hints can be used to force case splits, add known facts as hypotheses, suggest the use of an appropriate induction, and so on. In addition, ACL2 supports notions of *default* and *computed* hints, which permit the user to write code that generates hints dynamically during proof search, based on the shape of the current subgoal. The reuse of the hint mechanism provides these sophisticated capabilities to the interface for free.

Now we turn to the issues involved in *designating* a tool as a clause processor. For verified clause processors, we reuse ACL2's existing *rule class* mechanism, which has traditionally been used to classify theorems into categories, such as rewrite rules, forward chaining rules, type prescription rules, etc., that enable the theorem prover to apply such theorems in subsequent proofs. The new rule class designating clause processors fits well into this framework. When a theorem is designated as a clause processor, ACL2 associates a special property with the corresponding clause processor function, to be checked when a *clause processor* hint is supplied later.

More interesting is the designation of a function as an *unverified* clause processor, for which there is no associated theorem (or rule class) explicitly stated or stored in the theorem prover's database. In addition, recall that unverified clause processors might be associated with their own implicit theories specified by templates. To support unverified clause processors, we use another general mechanism available in ACL2, called *table events*. A *table* is simply a named finite function mapping keys to values, and *table events* support table update, that is, making a new key/value association. To support unverified clause processors, we use a new built-in table (initially empty) that associates the name of each unverified clause processor with a Boolean value indicating whether it is associated with a template. A clause

processor associated with a template is referred to as a *dependent* clause processor. Support for dependent clause processors, however, is tricky, principally because of the requirement of *atomicity*. Recall from Sect. 14.3 that a template event must not be separated from its dependent clause processor. We now discuss the subtleties involved in implementing this requirement.

To provide user-level support for atomicity, we design a macro that introduces the template event together with the table event designating the clause processor. The subtlety, however, arises from the fact that the logical requirement demands that we *enforce* the atomicity for these two events, not merely support it. For instance, a naive implementation of the macro might be one that expands into a sequence of events that successively introduces the template and the table event. However, in ACL2 macros do not have any semantic content, and the user can introduce the two components of the sequence separately without using the macro.

To resolve this issue, we note that a template is an encapsulation and thus contains a sequence of events. Our macro inserts the table event *inside* the corresponding template. However, this macro is merely syntactic sugar. At a deeper level, ACL2 *defines* the notion of template (that is, defines when the encapsulation introduces incomplete constraints) as follows: an encapsulation is a template if and only if it constrains at least one new function and also introduces an unverified clause processor. Thus, while finishing the admission of an encapsulation, the implementation checks (using the new table) whether the encapsulation introduces at least one dependent clause processor (that was not present before the encapsulation). If so, then the newly constrained functions are all marked as having incomplete constraints.

Finally, consider a subtlety that arises as a result of the above "definition" of templates from the implementation perspective, namely the disambiguation of dependent clause processors in the context of *nested encapsulations*. The sequence of events introduced by an encapsulation can include another encapsulation. Suppose that a clause processor, *tool*, is introduced in the scope of two nested encapsulations, each of which introduces at least one constrained function. Which of these two encapsulations should be considered to be the promised encapsulation associated with that dependent clause processor? The implementor of *tool* needs an answer to this question in order to meet the logical burden explained in the preceding section.

Our "solution" to the above dilemma is to simply disallow the scenario! We have not found an application that provides any compelling ground to support such a situation, so the implementation simply causes an error in this case. We postpone consideration of this complexity until a user provides a specific application where such functionality makes sense, thus providing guidance for the design of the resulting extension.

14.4.2 Miscellaneous Engineering Considerations

Our description above focuses on basic design decisions for the clause processor interface in ACL2. We now mention briefly *engineering considerations* involved in the implementation. These considerations are important both to provide a pleasant user

experience and to enforce critical logical requirements. A comprehensive treatment of the numerous choices made in the implementation of the interface is beyond the scope of this paper; we merely list a few illustrative issues to provide a flavor of the implementation complexity and choices:

Support for user flexibility. In our description of the interface, we confined ourselves to the simplistic view that a clause processor takes a clause and returns a list of clauses. The implementation, however, supports a more general notion of a *clause processor*. In particular, the implementation permits a clause processor to return multiple values, where the first value is an error flag and the second is the clause list of interest. If the application of a clause processor during a proof search causes the error flag to be set then the proof aborts. The implementation allows the error flag to be a string or a formatted string (a list with arguments for formatted output directives), which is printed when a proof aborts. The designer of a connection of ACL2 with a specific external tool can, therefore, use the flag to output a helpful error message. Note that the form of the theorem installing a verified clause processor is suitably adjusted for tools returning multiple values so that the proof obligations are trivial in the case that the tool returns an error. Finally, we note that when a clause-processor hint returns, without error, the one-element list containing the given clause, then the hint is considered a no-op and the proof continues.

Implementing execution support. Note that a tool introduced as a clause processor is a function introduced in ACL2. But a clause processor function requires further restrictions on how it can be introduced. The reason is that a clause processor function is *executed* on the ACL2 subgoal on which it is invoked; however, some functions introduced in ACL2 via the extension principles cannot be executed, in particular those introduced as constrained functions via encapsulation. We therefore check that a function designated to be a clause processor is executable. Furthermore, to enable efficient execution, we check that (1) the *guard* (input constraint) on a verified clause processor is trivially implied by the assumption that it is given a clause and (2) the result of invoking a clause processor is a well-formed list of clauses. The latter is dynamically checked during the invocation of the clause processor.

Syntactic Checks and Error Messages. An acclaimed strength of the ACL2 system is the detailed feedback provided by the theorem prover in an aborted proof attempt. Such feedback is essential in the applicability of a general-purpose theorem prover in large-scale verification projects. We incorporate several syntactic checks with the interface, whose violation produces helpful feedback in the form of error messages. These include checking that (1) the *supporters* field contains a list of function symbols which have been introduced in the current theory, (2) a template event introduces at least one new function, and (3) the theorem installing a verified clause processor uses distinct variables for the clause and the a-list arguments.

Checks in Functional Instantiation. Recall from Sect. 14.3 that due to soundness concerns, functions introduced via templates cannot be functionally instantiated. The implementation enforces such restrictions, as we now explain. Functional

instantiation applies a given functional substitution to a previously proved theorem; but it requires that upon applying the substitution to the constraints on functions in its domain, the resulting formulas are all theorems. So, we need to be able to collect those constraints. During the use of functional instantiation we, therefore, cause an error if (1) there is a partially constrained function in the domain of the given functional substitution or (2) a partially constrained function makes it impossible to collect all the requisite constraints. How might the latter be possible? Consider for example introducing a formula φ as an axiom via the *defaxiom* principle. Suppose φ is allowed to mention a partially constrained function f. Since the set of constraints explicitly specified for f is incomplete, we cannot determine if the *unspecified* constraints involve a function g in the domain of the given functional substitution, whose constraints must be considered as mentioned above. Hence we do not allow a partially constrained function to support a *defaxiom* event.

Signature Checking in Templates. A template with an unverified tool is implemented with an encapsulation together with a table event designating the requisite clause processor. ACL2's encapsulation syntax includes a *signature* that specifies the set of functions being introduced by encapsulation, but as a convenience to the user, ACL2 also allows additional functions to be defined nonlocally in an encapsulation. When this is done, ACL2 attempts to "move" such definitions outside the encapsulation, introducing them before or after the introduction of the functions designated by the signature in the encapsulation itself. If ACL2 fails to move such a definition, then the function is introduced in the encapsulation with its definitional equation stipulated as a constraint. However, for encapsulations designated as templates, it is unclear whether the promised encapsulation intended by the tool implementor involves constraining such nonlocally defined functions or not; so we require that each function introduced by a template must be explicitly documented in the signature of the template event.

Concerns with flattening encapsulations. ACL2 has a command called `:puff` that replaces an encapsulation event with the events introduced in the encapsulation. However, recall that template events must atomically introduce both the partially constrained functions and the dependent clause processor. To enforce this, the implementation of `:puff` is modified to have no effect on encapsulations introducing clause processors. Note, however, that it is possible to "fool" `:puff` into destroying this atomicity when the template is introduced manually rather than via the user-level macro for dependent clause processors mentioned above. Fortunately, the use of `:puff` leaves a mark on the resulting ACL2 session to indicate that the proofs done in the session cannot be trusted; `:puff` is used only for "hacking" during interactive proof development. This restriction on `:puff` is thus merely to support a reasonable hacking experience.

We conclude the discussion of the implementation with a brief remark on its impact. As the different concerns above illustrate, developing a sound interface for connecting a mature theorem prover like ACL2 with other tools requires significant attention to the interplay of the interface with features already existing in the system. It can, therefore, be dangerous (and potentially unsound) to embark on such a

connection by modifying the source code of the theorem prover without adequate understanding of the relevant subtleties. This in turn underlines the significance of providing a disciplined mechanism that enables the user to build such connections without the overhead of acquiring a deep understanding of the internals of the theorem prover's source code and subtle logical issues. However, prior to the development of the interface described here, an ACL2 user endeavoring to use external tools in proofs was left with no choice but to hack the internals of the system. With the new feature, a user now has the ability to integrate ACL2 smoothly with other tools by employing the user-level mechanisms provided by the interface.

14.5 Summary

Different tools bring different capabilities to formal verification. A strength of general purpose theorem provers compared to many tools based on decision procedures is in the expressive power of the logic, which enables succinct definitions. But tools based on decision procedures, when applicable, typically provide more automated proof procedures than general purpose provers. Several ACL2 users have requested ways to connect ACL2 with automated decision procedures. The mechanisms described in this paper support a disciplined approach for using ACL2 with other tools, with a clear specification of the expectations from the tool in order to maintain soundness. Furthermore, the design of verified clause processors allows the user to control a proof through means other than ACL2's heuristics.

We have presented an interface for connecting ACL2 with external reasoning1 tools, but we have merely scratched the surface. It is well known that developing an effective interface between two or more deduction tools is a complicated exercise [126]. Preliminary experiments with our interface have been promising, and we expect that ACL2 users will find ways to decompose problems that take advantage of the new interface, creating new tools that are stronger than their components.

Our interface may perhaps be criticized on the grounds that developing a connection with an external tool requires knowledge of ACL2. But a connection between different formal tools must involve a connection between different logics, and the developer of such a connection must have a thorough understanding of both logics, including the legal syntax of terms, the axioms, and the inference rules. The logic of ACL2 is more complex than many others, principally because it offers proof structuring mechanisms by enabling the user to mark events as *local*. This complexity manifests itself in the interface, for example, the interface requires like *supporters* essentially to enable tool developers to provide logical guarantees in the presence of local events. However, we believe that these constructs make it possible to implement connections with ACL2 without unreasonable demands on understanding the theorem prover implementation or esoteric aspects of logic.

Finally, the restrictions for the tool developers that we have outlined in this paper preclude certain external deduction tools. For instance, recall the connection between HOL4 with ACL mentioned in Chap. 13. As we remarked then, this

connection was primarily to help the HOL user to exploit ACL2's superior execution capabilities and automation. On the other hand, we saw how the ACL2 user could make use of HOL's expressive power, for example, in the verification of compositional model checking reductions. However, HOL cannot be connected using the interface outlined here, in a way in which the obligations outlined for the developer of the connection can be met. To understand why, note that the obligations ensure that the theory used by the external tool could *in principle* be formalized in ACL2. For instance, if template events are used to connect ACL2 with a tool whose theory is incompletely formalized, the burden is on the developer of the connection to guarantee that every theorem proven by the tool is a theorem of the theory obtained by replacing the template with its promised encapsulation. Since the logic of ACL2 is first order, this requirement rules out connections with logics stronger than first-order logic. It may be possible to extend the logical foundations of ACL2 to facilitate such a connection. The key idea is that the ACL2 theorem prover might be viewed as a theorem prover for the HOL logic. If the view is viable then it will be possible for the user of ACL2 to prove some formulas in HOL and use them in an ACL2 session, claiming that the session essentially reflects a HOL session mirrored in ACL2.

14.6 Bibliographic Notes

The importance of providing means for connecting with external tools has been widely recognized in the theorem proving community. Some early ideas for connecting different theorem provers are discussed in a proposal for so-called "interface logics" [95], with the goal to connect automated reasoning tools by defining a single logic L such that the logics of the individual tools can be viewed as sublogics of L. More recently, with the success of model checkers and Boolean satisfiability solvers, there has been significant work connecting such tools with interactive theorem provers. The PVS theorem prover provides connections with several decision procedures such as model checkers and SAT solvers [208, 235]. The Isabelle theorem prover [189] uses unverified external tools as *oracles* for checking formulas as theorems during a proof search; this mechanism has been used to integrate model checkers and arithmetic decision procedures with Isabelle [13,182]. Oracles are also used in the HOL family of higher order logic theorem provers [88]; for instance, the PROSPER project [65] uses the HOL98 theorem prover as a uniform and logically based coordination mechanism between several verification tools. The most recent incarnation of this family of theorem provers, HOL4, uses an external oracle interface to decide large Boolean formulas through connections to state-of-the-art BDD and SAT-solving libraries [86] and also uses that oracle interface to connect HOL4 with ACL2 as mentioned in Chap. 13. Meng and Paulson [169] interface Isabelle with a resolution theorem prover.

The primary basis for interfacing external tools with theorem provers for higher order logic (specifically HOL and Isabelle) involves the concept of "theorem

tagging," introduced by Gunter for HOL90 [93]. The idea is to introduce a tag *in the logic* for each oracle and view a theorem certified by the oracle as an implication with the tag corresponding to the certifying oracle as a hypothesis. This approach enables tracking of dependencies on unverified tools at the level of individual theorems. In contrast, our approach is designed to track such dependencies at the level of files, that is, ACL2 books. Our coarser level of tracking is at first glance unfortunate: if a book contains some events that depend on such tools and others that do not, then the entire book is "tainted" in the sense that its certification requires an appropriate acknowledgment for the tools. We believe that this issue is not significant in practice, as ACL2 users typically find it easy to move events between books. On the positive side, it is simpler to track a single event *introducing* an external tool rather than the *uses* of such an event, especially since hints are ignored when including previously certified books.

There has also been work on using an external tool to search for a proof that can then be checked by the theorem prover without assistance from the tool. Hurd [113] describes such an interface connecting HOL with first-order logic. McCune and Shumsky [164] present a system called Ivy, which uses Otter to search for first-order proofs of equational theories and then invokes ACL2 to check such proof objects.

Finally, as mentioned in Sect. 14.2, several ACL2 users have integrated external tools with ACL2, albeit requiring implementation hacks on the ACL2 source code. The author, in collaboration with Matthews, and Tuttle integrated ACL2 with SMV [212]. Reeber and Hunt connected ACL2 with the Zchaff satisfiability solver [215], and Sawada and Reeber provided a connection with SixthSense [230]. Manolios and Srinivasan connected ACL2 with UCLID [155, 156].

The results in this chapter represent joint work with Matt Kaufmann, J Strother Moore, and Erik Reeber. The interface to external tools was presented in two previous papers [130, 131].

Part VI
Conclusion

Chapter 15
Summary and Conclusion

The focus of this monograph 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. We formalized the semantics of LTL in ACL2 and used it to prove characterization theorems for a compositional model checking procedure. We also created a general interface connecting ACL2 to arbitrary external tools and developed a precise specification for the logical requirement for an external tool to be compatible with the logic of ACL2.

Our approach 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 twofold. First, one can express the statement of correctness concisely as a formal statement in a mathematical logic. Second, 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.

S. Ray, *Scalable Techniques for Formal Verification*, DOI 10.1007/978-1-4419-5998-0_15, 219

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 generated 46 state predicates and 117 input predicates. The abstract system, however, still 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 monograph, 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.

The central thesis of the research here is that it is critical to develop generic, hierarchical reasoning frameworks where different verification problems can be handled at different levels of abstraction. Unfortunately, the key emphasis in the formal verification research has been on raw automation, with other aspects of verification (*e.g.*, control, structuring, decomposition, etc.) often taking the back seat. There are several reasons for it. One important one is that a reusable, hierarchical framework takes time to build, and in the "heat of the battle" with a problem that need to be solved under strict time-to-market constraints such considerations are often foregone in the interest of getting the current job on hand done. However, as we have

seen several times in this book, a generic compositional framework can have significant long-term benefit, with the initial setup cost more than compensated for with reuse. Furthermore, the act of generic formalization exposes the underlying structure of the problems, helps spot connections between various subproblems, and thereby leads to the identification of the verification strategy most suited for the problem at hand. For instance, the connection between clock functions and stepwise invariants that we discussed in Chap. 6 simply fell out once the strategies were formalized appropriately, resolving much (informal) contention regarding the expressive power of the strategies in one fell swoop. Given that there is only so much that raw automation can accomplish, and given that the size of verification problems is growing faster than the capacity of fully automated techniques, a framework that lets us hierarchically decompose verification problems is critical to its scalability of formal verification in large-scale system analysis.

References

1. M. Aagaard, V. C. Ciubotariu, F. Khalvati, and 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.
2. 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.
3. M. Abadi and L. Lamport. The Existence of Refinement Mappings. *Theoretical Computer Science*, 82(2):253–284, May 1991.
4. A. W. Appel. Foundational Proof-Carrying Code. In *Proceedings of the 16th IEEE International Symposium on Logic in Computer Science (LICS 2001)*, pages 247–258, Washington, DC, 2001. IEEE Computer Society Press.
5. K. R. Apt. Ten Years of Hoare's Logic: A Survey – Part I. *ACM Transactions on Programming Languages and Systems (ACM TOPLAS)*, 3(4):431–483, October 1981.
6. K. R. Apt and D. Kozen. Limits for Automatic Verification of Finite-State Concurrent Systems. *Information Processing Letters*, 15:307–307, 1986.
7. N. S. Arora, R. D. Blumofe, and C. G. Plaxton. Thread Scheduling in Multiprogramming Multiprocessors. *Theory of Computing Systems*, 34:115–144, 2001.
8. 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, Atlanta, GA, May 1999. ACM Press.
9. F. Baader and T. Nipkow. *Term Rewriting and All that*. Cambridge University Press, 1998.
10. 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.
11. 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, Toronta, ON, 2001. Springer-Verlag.
12. 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.
13. 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.
14. W. R. Bevier. *A Verified Operating System Kernel*. PhD thesis, Department of Computer Sciences, The University of Texas at Austin, 1987.
15. 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.

16. W. W. Bledsoe. Non-Resolution Theorem Proving. *Artificial Intelligence*, 9(1):1–35, 1977.
17. 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.
18. 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*, pages 403–418, Chicago, IL, July 2000. Springer-Verlag.
19. R. S. Boyer, D. Goldshlag, M. Kaufmann, and J. S. Moore. Functional Instantiation in First Order Logic. In V. Lifschitz, editor, *Artificial Intelligence and Mathematical Theory of Computation: Papers in Honor of John McCarthy*, pages 7–26. Academic Press, 1991.
20. 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.
21. R. S. Boyer and J. S. Moore. *A Computational Logic*. Academic Press, New York, NY, 1979.
22. 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.
23. R. S. Boyer and J. S. Moore. *A Computational Logic Handbook*. Academic Press, New York, NY, 1988.
24. 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.
25. R. S. Boyer and J. S. Moore. *A Computational Logic Handbook*. Academic Press, London, UK, 1997.
26. 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.
27. R. S. Boyer and Y. Yu. Automated Proofs of Object Code for a Widely Used Microprocessor. *Journal of the ACM*, 43(1), January 1996.
28. 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, New Brunswick, NJ, July 1996. Springer-Verlag.
29. 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.
30. B. Brock and W. A. Hunt, Jr. The Dual-Eval Hardware Description Language. *Formal Methods in Systems Design*, 11(1):71–104, 1997.
31. B. Brock and W. A. Hunt, Jr. Formal Analysis of the Motorola CAP DSP. In *Industrial-Strength Formal Methods in Practice*. Springer, 1999.
32. 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, Palo Alto, CA, 1996. Springer-Verlag.
33. 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.
34. M. Browne, E. M. Clarke, and O. Grumberg. Characterizing Finite Kripke Structures in Propositional Temporal Logic. *Theoretical Computer Science*, 59, 1988.
35. R. E. Bryant. Graph-Based Algorithms for Boolean Function Manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, August 1986.

36. 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, Trendo, Italy, 1999. Springer-Verlag.
37. 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, Copenhagen, Denmark, July 2002. Springer-Verlag.
38. 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, Stanford, CA, 1994. Springer-Verlag.
39. 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.
40. G. Cantor. Beitrãge zur Begrũndung der transfiniten Mengenlehre. *Mathematische Annalen*, xlvi:481–512, 1895.
41. G. Cantor. Beitrãge zur Begrũndung der transfiniten Mengenlehre. *Mathematische Annalen*, xlix:207–246, 1897.
42. G. Cantor. *Contributions to the Founding of the Theory of Transfinite Numbers*. Dover Publications Inc., 1952. Translated by P. E. B. Jourdain.
43. K. M. Chandy and J. Misra. *Parallel Program Design: A Foundation*. Addison-Wesley, Cambridge, MA, 1990.
44. 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, Trendo, Italy, 1999. Springer-Verlag.
45. 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.
46. A. Church and S. C. Kleene. Formal Definitions in the Theory of Ordinal Numbers. *Fundamenta Mathematicae*, 28:11–21, 1937.
47. 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* 11*th International Conference on Computer-Aided Verification (CAV 1999)*, volume 1633 of *LNCS*, pages 495–499, Trendo, Italy, 1999. Springer-Verlag.
48. E. M. Clarke. *Completeness and Incompleteness of Hoare-Like Axiom Systems*. PhD thesis, Cornell University, 1976.
49. 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.
50. 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* 11*th International Conference on Computer-Aided Verification (CAV 1998)*, volume 1427 of *LNCS*, pages 147–158, Vancover, BC, 1998. Springer-Verlag.
51. 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.
52. 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, Chicago, IL, 2000. Springer-Verlag.
53. E. M. Clarke, O. Grumberg, and D. A. Peled. *Model-Checking*. The MIT Press, Cambridge, MA, January 2000.

54. 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.
55. M. Clint. Program Proving: Coroutines. *Acta Informatica*, 2:50–63, 1973.
56. M. Clint and C. A. R. Hoare. Program Proving: Jumps and Functions. *Acta Informatica*, 1:214–224, 1971.
57. A. Cohn. A Proof of Correctness of the VIPER Microprocessor. Technical Report 104, University of Cambridge, Computer Laboratory, January 1987.
58. C. Colby, P. Lee, G. C. Necula, F. Blau, M. Plesko, and K. Cline. A Certifying Compiler for Java. In *ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation (PLDI 2000)*, pages 95–107. ACM Press, 2000.
59. R. L. Constable, S. F. Allen, H. M. Bromley, W. R. Cleveland, J. F. Cremer, R. W. Harper, D. J. Howe, T. B. Knoblock, N. P. Mendler, P. Panangaden, J. T. Sasaki, and S. S. Smith. *Implementing Mathematics with the Nuprl Development System*. Prentice-Hall, NJ, 1986.
60. S. A. Cook. Soundness and Completeness of an Axiom System for Program Verification. *SIAM Journal of Computing*, 7(1):70–90, 1978.
61. 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.
62. 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, Trento, Italy, 1999. Springer-Verlag.
63. 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.
64. J. W. de Bakker. *Mathematical Theory of Program Correctness*. Prentice-Hall, 1980.
65. 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.
66. 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.
67. E. W. Dijkstra. Guarded Commands, Non-determinacy and a Calculus for Derivation of Programs. *Language Hierarchies and Interfaces*, pages 111–124, 1975.
68. E. W. Dijkstra. *A Discipline of Programming*. Prentice-Hall, 1978.
69. 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, New Brunswick, NJ, July 1996. Springer-Verlag.
70. 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.
71. 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.
72. 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, L'Aquila, Italy, July 2003. Springer-Verlag.
73. 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.

74. C. Flanagan and S. Qadeer. Predicate Abstraction for Software Verification. In *Proceedings of the* 29*th ACM SIGPLAN SIGACT Symposium on Principles of Programming Languages (POPL 2002)*,pages 191–202, Portland, OR, 2002. ACM Press.
75. C. Flanagan and J. B. Saxe. Avoiding Exponential Explosion: Generating Compact Verification Conditions. In *Proceedings of the* 28*th ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages (POPL 2001)* , pages 193–205, London, UK, 2001. ACM Press.
76. 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.
77. 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.
78. 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.
79. R. Gamboa. *Mechanically Verifying Real-Valued Algorithms in ACL2*. PhD thesis, Department of Computer Sciences, The University of Texas at Austin, 1999.
80. P. Y. Gloess. Imperative Program Verification in PVS. Technical Report, École Nationale Supérieure Électronique, Informatique et Radiocommunications de bordeaux, 1999. See URL http://dept-info.labri.u.bordeaux.fr/imperative/index.html.
81. K. Gödel. Über formal unentscheidbare Sätze der Principia Mathematica und verwandter Systeme I. *Monatshefte für Mathematic und Physik*, 38:173–198, 1931.
82. K. Gödel. *On Formally Undecidable Propositions of Principia Mathematica and Related Systems*. Dover Publications, February 1992.
83. D. M. Goldshlag. Mechanically Verifying Concurrent Programs with the Boyer-Moore Prover. *IEEE Transactions on Software Engineering*, 16(9):1005–1023, 1990.
84. 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.
85. M. J. C. Gordon. The Semantic Challenges of Verilog HDL. In *Proceedings of the* 10*th Annual IEEE Symposium on Logic in Computer Science (LICS 1995)*, pages 136–145, San Diego, CA, 1995. IEEE Computer Society Press.
86. M. J. C. Gordon. Programming Combinations of Deduction and BDD-Based Symbolic Calculation. *LMS Journal of Computation and Mathematics*, 5:56–76, 2002.
87. M. J. C. Gordon, W. A. Hunt, Jr., M. Kaufmann, and J. Reynolds. An Integration of HOL and ACL2. In A. Gupta and P. Manolios, editors, *Proceedings on the* 6*th International Conference on Formal Methods in Computer-Aided Design (FMCAD-2006)*, pages 153–160, San Jose, CA, 2006. IEEE Computer Society Press.
88. M. J. C. Gordon and T. F. Melham, editors. *Introduction to HOL: A Theorem-Proving Environment for Higher-Order Logic*. Cambridge University Press, 1993.
89. M. J. C. Gordon and A. M. Pitts. The HOL Logic and System. In J. Bowen, editor, *Towards Verified Systems*, volume 2 of *Real-Time Safety Critical Systems*, chapter 3, pages 49–70. Elsevier Science B.V., 1994.
90. S. Graf and H. Saidi. Construction of Abstract State Graphs with PVS. In O. Grumberg, editor, *Proceedings of the* 9*th International Conference on Computer-Aided Verification (CAV 1997)*, volume 1254 of *LNCS*, pages 72–83, Haifa, Israel, 1997. Springer-Verlag.
91. 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.
92. D. A. Greve. Symbolic Simulation of the JEM1 Microprocessor. In G. Gopalakrishnan and P. Windley, editors, *Proceedings of the* 2*nd International Conference on Formal Methods in Computer-Aided Design (FMCAD 1998)*, volume 1522 of *LNCS*, Palo Alto, CA, 1998. Springer-Verlag.
93. E. L. Gunter. Adding External Decision Procedures to HOL90 Securely. *Lecture Notes in Computer Science*, 1479:143–152, 1998.

94. A. Gupta. Formal Hardware Verification Methods: A Survey. *Formal Methods in Systems Design*, 2(3):151–238, October 1992.
95. J. D. Guttman. A Proposed Interface Logic for Verification Environments. Technical Report M-91-19, The Mitre Corporation, March 1991.
96. 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, Barcelona, Spain, 2004. Springer-Verlag.
97. D. Hardin, E. W. Smith, and W. D. Young. A Robust Machine Code Proof Framework for Highly Secure Applications. In P. Manolios and M. Wilding, editors, *Proceedings of the 6th International Workshop on the ACL2 Theorem Prover and Its Applications (ACL2 2006)*, pages 11–20, Seattle, WA, July 2006. ACM.
98. J. Harrison. Metatheory and Reflection in Theorem Proving: A Survey and Critique. Technical Report CRC-053, SRI International Cambridge Computer Science Research Center, 1995.
99. J. Harrison. Formalizing Dijkstra. In J. Grundy and M. Newer, editors, *Proceedings of the 11th International Conference on Theorem Proving in Higher Order Logics (TPHOLS 1998)*, volume 1479 of *LNCS*, pages 171–188, Canberra, ACT, 1998, Springer-Verlag.
100. 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. See URL http://www.cl.cam.ac.uk/users/jrh/hol-light/.
101. 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, Portland, OR, 2002. ACM Press.
102. 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.
103. C. A. R. Hoare. An Axiomatic Basis for Computer Programming. *Communications of the ACM*, 12(10):576–583, 1969.
104. C. A. R. Hoare. Proof of Correctness of Data Representations. *Acta Informatica*, 1:271–281, 1972.
105. G. J. Holzmann. *The SPIN Model Checker: Primer and Reference Manual*. Addison-Wesley, November 2003.
106. P. Homeier and D. Martin. A Mechanically Verified Verification Condition Generator. *The Computer Journal*, 38(2):131–141, July 1995.
107. 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*, Chicago, IL, July 2000. Springer-Verlag.
108. W. A. Hunt, Jr. *FM8501: A Verified Microprocessor*, volume 795 of *LNAI*. Springer-Verlag, 1994.
109. 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.
110. W. A. Hunt, Jr., M. Kaufmann, R. Krug, J. S. Moore, and E. Smith. Meta Reasoning in ACL2. In J. Hurd and T. Melham, editors, *Proceedings of the 18th International Conference on Theorem Proving in Higher Order Logics (TPHOLs 2005)*, volume 3603 of *LNCS*, pages 163–178, Oxford, England, 2005. Springer-Verlag.
111. 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, L'Aquila, Italy, July 2003. Springer-Verlag.
112. 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.

113. J. Hurd. An LCF-Style Interface Between HOL and First-Order Logic. In A. Voronkov, editor, *Proceedings of the* 18*th International Conference on Automated Deduction (CADE 2002)*, volume 2392 of *LNCS*, pages 134–138, Copenhagen, Denmark, 2002. Springer-Verlag.
114. 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, Canberra, ACT, 1998. Springer-Verlag.
115. 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*, Paris, France, 2001. Springer-Verlag.
116. R. B. Jones. *Symbolic Simulation Methods for Industrial Formal Verification*. Kluwer Academic Publishers, June 2002.
117. B. Jonsson, A. Pnueli, and C. Rump. Proving Refinement Using Transduction. *Distributed Computing*, 12(2-3):129–149, 1999.
118. HOL 4, Kananaskis 1 release. http://hol.sf.net/.
119. 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.
120. M. Kaufmann. Modular Proof: The Fundamental Theorem of Calculus. In P. Manlolios, M. Kaufmann, and J. S. Moore, editors, *Computer-Aided Reasoning: ACL2 Case Studies*, pages 59–72. Kluwer Academic Publishers, June 2000.
121. M. Kaufmann, P. Manolios, and J. S. Moore, editors. *Computer-Aided Reasoning: ACL2 Case Studies*. Kluwer Academic Publishers, Boston, MA, June 2000.
122. M. Kaufmann, P. Manolios, and J. S. Moore. *Computer-Aided Reasoning: An Approach*. Kluwer Academic Publishers, Boston, MA, June 2000.
123. 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.
124. M. Kaufmann and J. S. Moore. ACL2 Home Page. See URL http://www.cs.utexas.edu/users/-moore/acl2.
125. 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.
126. M. Kaufmann and J. S. Moore. Should We Begin a Standardization Process for Interface Logics? Technical Report 72, Computational Logic Inc. (CLI), January 1992.
127. 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.
128. 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.
129. M. Kaufmann and J. S. Moore. Structured Theory Development for a Mechanized Logic. *Journal of Automated Reasoning*, 26(2):161–203, 2001.
130. M. Kaufmann, J. S. Moore, S. Ray, and E. Reeber. Integrating External Deduction Tools with ACL2. In C. Benzmüller, B. Fischer, and G. Sutcliffe, editors, *Proceedings of the* 6*th International Workshop on Implementation of Logics (IWIL 2006)*, volume 212 of *CEUR Workshop Proceedings*, pages 7–26, Phnom Penh, Cambodia, November 2006.
131. M. Kaufmann, J. S. Moore, S. Ray, and E. Reeber. Integrating External Deduction Tools with ACL2. *Journal of Applied Logic*, 7(1):3–25, March 2009.
132. M. Kaufmann and R. Sumners. Efficient Rewriting of Data Structures in ACL2. In D. Borrione, M. Kaufmann, and J. S. Moore, editors, *Proceedings of* 3*rd International Workshop on the ACL2 Theorem Prover and Its Applications (ACL2 2002)*, pages 141–150, Grenoble, France, April 2002.
133. 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.
134. J. C. King. *A Program Verifier*. PhD thesis, Carnegie-Melon University, 1969.

135. K. Kunen. A Ramsey Theorem in Boyer-Moore Logic. *Journal of Automated Reasoning*, 15(2), October 1995.
136. 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, Boulder, CO, July 2003. Springer-Verlag.
137. 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, Venice, Italy, 2004. Springer-Verlag.
138. 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 Verification (CAV 2004)*, volume 3117 of *LNCS*, pages 135–147, Boston, MA, July 2004. Springer-Verlag.
139. 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* 15*th International Conference on Computer-Aided Verification*, volume 2275 of *LNCS*, pages 141–153, Boulder, CO, 2003. Springer-Verlag.
140. Y. Lakhnech, S. Bensalem, S. Berezin, and S. Owre. Incremental Verification by Abstraction. 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 98–112, Genova, Italy, April 2001. Springer-Verlag.
141. L. Lamport. A New Solution of Dijkstra's Concurrent Programming Problem. *Communications of the ACM*, 17(8):453–455, August 1974.
142. L. Lamport. Proving the Correctness of Multiprocessor Programs. *IEEE Transactions on Software Engineering*, SE-3(2):125–143, March 1977.
143. L. Lamport. Specifying Concurrent Program Modules. *ACM Transactions on Programming Languages and Systems (ACM TOPLAS)* , 5(2):190–222, April 1983.
144. L. Lamport. What Good is Temporal Logic? *Information Processing*, 83:657–688, 1983.
145. L. Lamport. The Temporal Logic of Actions. *ACM Transactions on Programming Languages and Systems (ACM TOPLAS)*, 16(3):827–923, May 1994.
146. L. Lamport, J. Matthews, M. Tuttle, and Y. Yu. Specifying and Verifying Systems with TLA+. In E. Jul, editor, *Proceedings of the* 10*th ACM SIGOPS European Workshop*, pages 45–48, Copenhagen, Denmark, 2002.
147. 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* 17*th International Conference on Theorem Proving in Higher Order Logics (TPHOLs 2004)*, volume 3233 of *LNCS*, pages 184–200, Park City, Utah, 2004. Springer-Verlag.
148. H. Liu and J. S. Moore. Executable JVM model for Analytical Reasoning: A Study. *Science of Computer Programming*, 57(3):253–274, 2005.
149. Z. Manna. The Correctness of Programs. *Journal of Computer and Systems Sciences*, 3(2):119–127, 1969.
150. P. Manolios. Correctness of Pipelined Machines. In W. A. Hunt, Jr. and S. D. Johnson, editors, *Proceedings of the* 3*rd International Conference on Formal Methods in Computer-Aided Design (FMCAD 2000)*, volume 1954 of *LNCS*, pages 161–178, Austin, TX, 2000. Springer-Verlag.
151. 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.
(one more editor s present)
152. P. Manolios. A Compositional Theory of Refinement for Branching Time. In D. Geist, editor, *Proceedings of the* 12*th Working Conference on Correct Hardware Design and Verification Methods*, volume 2860 of *LNCS*, pages 304–218, L'Aquila, Italy, 2003. Springer-Verlag.
153. P. Manolios and J. S. Moore. Partial Functions in ACL2. *Journal of Automated Reasoning*, 31(2):107–127, 2003.

154. 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* 11*th International Conference on Computer-Aided Verification (CAV 1999)*, volume 1633 of *LNCS*, pages 369–379, Trento, Italy, 1999. Springer-Verlag.
155. 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.
156. 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.
157. 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.
158. 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, Austin, TX, November 2004. Springer-Verlag.
159. J. Matthews, J. S. Moore, S. Ray, and D. Vroon. Verification Condition Generation via Theorem Proving. In M. Hermann and A. Voronkov, editors, *Proceedings of the* 13*th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning (LPAR 2006)*, volume 4246 of *LNCS*, pages 362–376, Phnom Penh, Cambodia, November 2006. Springer.
160. J. Matthews and D. Vroon. Partial Clock Functions in ACL2. 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.
161. J. McCarthy. Towards a Mathematical Science of Computation. In *Proceedings of the Information Processing Congress*, volume 62, pages 21–28. North-Holland, August 1962.
162. 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.
163. W. McCune. Solution to the Robbins Problem. *Journal of Automated Reasoning*, 19(3): 263–276, 1997.
164. 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.
165. K. McMillan. *Symbolic Model Checking*. Kluwer Academic Publishers, 1993.
166. 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, Vancouver, BC, 1998. Springer-Verlag.
167. K. McMillan, S. Qadeer, and J. Saxe. Induction in Compositional 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*, Chicago, IL, July 2000. Springer-Verlag.
168. F. Mehta and T. Nipkow. Proving Pointer Programs in Higher Order Logic. In F. Baader, editor, *Proceedings of the* 19*th International Conference on Automated Deduction (CADE 2003)*, volume 2741 of *LNAI*, pages 121–135, Miami, FL, 2003. Springer-Verlag.
169. J. Meng and L. C. Paulson. Experiments on Supporting Interactive Proof Using Resolution. In D. A. Basin and M. Rusinowitch, editors, *Proceedings of the* 2*nd International Joint Conference on Computer-Aided Reasoning (IJCAR 2004)*, volume 3097 of *LNCS*, pages 372–384, Cork, Ireland, 2004. Springer-Verlag.
170. R. Milner. *Communication and Concurrency*. Prentice-Hall, 1990.
171. P. Molitor and J. Mohnke. *Equivalence Checking of Digital Circuits: Fundamentals, Principles, Methods*. Springer-Verlag, 2004.

172. H. Mony, J. Baumgartner, V. Paruthi, R. Kanzelman, and A. Kuehlmann. Scalable Automated Verification via Expert-System Guided Transformations. 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 217–233, Austin, TX, 2004. Springer-Verlag.
173. J. S. Moore. *Piton: A Mechanically Verified Assembly Language*. Kluwer Academic Publishers, 1996.
174. 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.
175. 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.
176. 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, Paris, France, September 2001. Springer-Verlag.
177. 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, L'Aquila, Italy, October 2003. Springer-Verlag.
178. 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.
179. J. S. Moore, T. Lynch, and M. Kaufmann. A Mechanically Checked Proof of the Kernel of the AMD5K86 Floating-Point Division Algorithm. *IEEE Transactions on Computers*, 47(9):913–926, September 1998.
180. J. S. Moore and G. Porter. The Apprentice Challenge. *ACM Transactions on Programming Languages and Systems (ACM TOPLAS)*, 24(3):1–24, May 2002.
181. M. Moskewicz, C. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an Efficient SAT Solver. In *Proceedings of the 38th Design Automation Conference (DAC 2001)*, pages 530–535, Las Vegas, NV, 2001. ACM Press.
182. 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. See URL http://www4.informatik.tu-muenchen.de/ nipkow/pubs/tacas.html.
183. K. Namjoshi. A Simple Characterization of Stuttering Bisimulation. In S. Ramesh and G. Sivakumar, editors, *Proceedings of the 17th International Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS 1997)*, volume 1346 of *LNCS*, pages 284–296, Kharagpur, India, 1997. Springer-Verlag.
184. 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, Chicago, IL, July 2000. Springer-Verlag.
185. G. Necula. *Compiling with Proofs*. PhD thesis, Carnegie-Melon University, September 1998.
186. G. C. Necula. Proof-Carrying Code. In *Proceedings of the 24th ACM SIGPLAN SIGACT Conference on Principles of Programming Languages (POPL 1997)*, pages 106–119, Paris, France, 1997. ACM Press.
187. G. Nelson and D. C. Oppen. Simplification by Cooperating Decision Procedures. *ACM Transactions on Programming Languages and Systems*, 1(2), October 1979.
188. A. Newell, J. C. Shaw, and H. A. Simon. Report on a General Problemn-Solving Program. In *IFIP Congress*, pages 256–264, 1959.
189. T. Nipkow, L. Paulson, and M. Wenzel. *Isabelle/HOL: A Proof Assistant for Higher Order Logics*, volume 2283 of *LNCS*. Springer-Verlag, 2002.
190. M. Norrish. *C Formalised in HOL*. PhD thesis, University of Cambridge, 1998.

191. 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.
192. D. C. Oppen and S. A. Cook. Proving Assertions About Programs that Manipulate Data Structures. In *Proceedings of the 7th Annual ACM Symposium on Theory of Computing (STOC 1975)*, pages 107–116, Albuquerque, NM, 1975. ACM Press.
193. S. S. Owicki and D. Gries. Verifying Properties of Parallel Programs: An Axiomatic Approach. *Communications of the ACM*, 19(5):279–285, 1976.
194. 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.
195. D. Park. Concurrency and Automata on Infinite Sequences. In *Proceedings of the 5th GI-Conference on Theoretical Computer Science*, volume 104 of *LNCS*, pages 167–183, Karlsruhe, Germany, 1981. Springer-Verlag.
196. L. Paulson. The Isabelle Reference Manual. See URL http://www.cl.cam.ac.uk/Research/-HVG/Isabelle/dist/Isabelle2003/doc/ref.pdf.
197. L. Paulson. Set Theory for Verification: I. From Foundations to Functions. *Journal of Automated Reasoning*, 11:353–389, 1993.
198. L. Paulson. Set Theory for Verification: II. Induction and Recursion. *Journal of Automated Reasoning*, 15:167–215, 1995.
199. L. Paulson. Mechanizing UNITY in Isabelle. 1(1):3–32, 2000.
200. L. Paulson. A Simple Formalization and Proof for the Mutilated Chess Board. *Logic Journal of the IGPL*, 9(3):499–509, 2001.
201. G. D. Plotkin. A Structural Approach to Operational Semantics. Technical Report DAIMI FN-19, University of Aarhus, 1981.
202. A. Pnueli. The Temporal Logic of Programs. In *Proceedings of the 18th Annual IEEE Symposium of Foundations of Computer Science*, pages 46–57, Providence, RI, October 1977. IEEE Computer Society Press.
203. 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.
204. A. Pnueli. Linear and Branching Structures in the Semantics and Logics of Reactive Systems. In W. Brauer, editor, *Proceedings of the 12th International Colloquium on Automata, Languages, and Programming (ICALP 1985)*, volume 194 of *LNCS*, pages 15–32, Nafplion, Greece, 1985. Springer-Verlag.
205. A. Pnueli, S. Ruah, and L. Zuck. Automatic Deductive Verification with Invisible Invariants. 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 82–97, Genova, Italy, 2001. Springer-Verlag.
206. President's Information Technology Advisory Committee. Information Technology Research: Investing in Our Future, February 1999. National Coordination Office for Computing, Information, and Communications. See URL http://www.ccic.gov/ac/report.
207. J. P. Queille and J. Sifakis. Specification and Verification of Concurrent Systems in CESAR. In *Proceedings of the 5th International Symposimum on Programming*, volume 137 of *LNCS*, pages 337–351, Colloquium is ter instead of symposium, Torino, Italy, 1982. Springer-Verlag.
208. S. Rajan, N. Shankar, and M. K. Srivas. An Integration of Model-Checking with Automated Proof Checking. In P. Wolper, editor, *Proceedings of the 8th International Conference on Computer-Aided Verification (CAV '95)*, volume 939 of *LNCS*, pages 84–97, Lige, Belgium, 1995. Springer-Verlag.
209. S. Ranise and C. Tinelli. The SMT-LIB Standard: Version 1.2. Technical Report, Department of Computer Science, The University of Iowa, 2006. Available at `www.SMT-LIB.org`.
210. 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 16th International Conference on Computer-Aided Verification (CAV 2004)*, volume 3114 of *LNCS*, pages 31–43, Boston, MA, July 2004. Springer-Verlag.

211. S. Ray, W. A. Hunt, Jr., J. Matthews, and J. S. Moore. A Mechanical Analysis of Program Verification Strategies. *Journal of Automated Reasoning*, 40(4):245–269, May 2008.
212. 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.
213. S. Ray and J. S. Moore. Proof Styles in Operational Semantics. 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 67–81, Austin, TX, November 2004. Springer-Verlag.
214. S. Ray and R. Sumners. Combining Theorem Proving with Model Checking Through Predicate Abstraction. *IEEE Design & Test of Computers*, 24(2):132–139, 2007.
215. E. Reeber and W. A. Hunt, Jr. A SAT-Based Decision Procedure for the Subclass of Unrollable List Formulas in ACL2 (SULFA). In U. Furbach and N. Shankar, editors, *Proceedings of the 3rd International Joint Conference on Computer-Aided Reasoning (IJCAR 2006)*, volume 4130 of *LNAI*, pages 453–467, Seattle, WA, 2006. Springer-Verlag.
216. 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.
217. J. C. Reynolds. Separation Logic: A Logic for Shared Mutable Data Structures. In *Proceedings of the 17th IEEE Symposium on Logic in Computer Science (LICS 2002)*, pages 55–74, Copenhagen, Denmark, July 2002. IEEE Computer Society.
218. J. A. Robinson. A Machine-Oriented Logic Based on the Resolution Principle. *Journal of the ACM*, 12(1):23–41, 1965.
219. H. Rogers, Jr. *Theory of Recursive Functions and Effective Computability*. MIT Press, 1987.
220. D. Russinoff. A Mechanical Proof of Quadratic Reciprocity. *Journal of Automated Reasoning*, 8:3–21, 1992.
221. D. Russinoff. A Mechanically Verified Incremental Garbage Collector. *Formal Aspects of Computing*, 6:359–390, 1994.
222. 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.
223. 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.
224. 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.
225. H. Saidi and N. Shankar. Abstract and Model Check While You Prove. In N. Halbwacha and D. Peled, editors, *Proceedings of the 11th International Conference on Computer-Aided Verification (CAV 1999)*, volume 1633 of *LNCS*, pages 443–453, Trento, Italy, 1999. Springer-Verlag.
226. 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.
227. 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, Haifa, Israel, 1997. Springer-Verlag.
228. 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, Vancouver, BC, 1998.Springer-Verlag.
229. 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.

230. J. Sawada and E. Reeber. ACL2SIX: A Hint Used to Integrate a Theorem Prover and an Automated Verification Tool. In A. Gupta and P. Manolios, editors, *Proceedings of the* 6*th International Conference on Formal Methods in Computer-Aided Design (FMCAD 2006)*, pages 161–168, San Jose, CA, November 2006. Springer-Verlag.
231. 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* 12*th International Conference on Theorem Proving in Higher Order Logics (TPHOLS 1999)*, volume 1690 of *LNCS*, pages 255–272, Nice, France, 1999. Springer-Verlag.
232. B. Schneier. *Applied Cryptography (2nd ed.): Protocols, Algorithms, and Source Code in C*. John Wiley & Sons, Inc., 1995.
233. 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, Boston, MA, July 2004. Springer-Verlag.
234. N. Shankar. *Metamathematics, Machines, and Gödel's Proof*. Cambridge University Press, 1994.
235. N. Shankar. Using Decision Procedures with Higher Order Logics. In R. J. Boulton and P. B. Jackson, editors, *Proceedings of the* 14*th International Conference on Theorem Proving in Higher-Order Logics (TPHOLs 2001)*, volume 2152 of *LNCS*, pages 5–26, Edinburgh, Scotland, 2001. Springer-Verlag.
236. J. R. Shoenfield. *Mathematical Logic*. Adison-Wesley, Reading, MA, 1967.
237. R. E. Shostak. A Practical Decision Procedure for Arithmetic with Function Symbols. *Journal of the ACM*, 26(2):351–360, April 1979.
238. K. Slind and J. Hurd. Applications of Polytypism in Theorem Proving. In D. Basin and B. Wolff, editors, *Proceedings of the* 16*th International Conference on Theorem Proving in Higher Order Logics (TPHOLs 2003)*, volume 2978 of *LNCS*, pages 103–119, Rom, Italy, 2003. Springer-Verlag.
239. S. W. Smith and V. Austel. Trusting Trusted Hardware: Towards a Formal Model of Programmable Secure Coprocessors. In *Proceedings of the* 3*rd USENIX Workshop on Electronic Commerce*, Boston, MA, September 1998.
240. S. Sokolowski. Axioms for Total Correctness. *Acta Informatica*, 9:61–71, 1977.
241. M. Srivas and M. Bickford. Formal Verification of a Pipelined Microprocessor. *IEEE Software*, 7(5):52–64, September 1990.
242. G. L Steele, Jr. *Common Lisp the Language*. Digital Press, 30 North Avenue, Burlington, MA 01803, 2nd edition, 1990.
243. 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.
244. 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.
245. 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.
246. R. Sumners. *Deductive Mechanical Verification of Concurrent Systems*. PhD thesis, Department of Electrical and Computer Engineering, The Unversity of Texas at Austin, 2005.
247. 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.
248. 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.
249. D. Toma and D. Borrione. Formal verification of a SHA-1 Circuit Core Using ACL2. In J. Hurd and T. Melham, editors, *Proceedings of the* 18*th International Conference on Theorem*

Proving in Higher-Order Logics (TPHOLS 2005), volume 3603 of *LNCS*, pages 326–341, Oxford, UK, 2005. Springer-Verlag.
250. A. M. Turing. On Computable Numbers, with an Application to the *Entscheidungsproblem*. *Proceedings of the London Mathematical Society*, 2(42):230–265, 1937.
251. 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.
252. W. Visser, K. Havelund, G. Brat, S. Park, and F. Lerda. Model Checking Programs. *Automated Software Engineering Journal*, 10(2):203–232, April 2003.
253. 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.
254. M. Wand. A New Incompleteness Result for Hoare's System. *Journal of the ACM*, 25(1): 168–175, January 1978.
255. H. Wang. Mechanical Mathematics and Inferential Analysis. In P. Braffort and D. Hershberg, editors, *Computer Programming and Formal Systems*. North-Holland, 1963.
256. R. Weyhrauch. Prolegomena to a Theory of Mechanized Formal Reasoning. *Artificial Intelligence Journal*, 13(1):133–170, 1980.
257. M. Wilding. A Mechanically Verified Application for a Mechanically Verified Environment. In C. Courcoubetis, editor, *Proceedings of the 5th International Conference on Computer-Aided Verification (CAV 1993)*, volume 697 of *LNCS*, pages 268–279, Elounda, Greece, 1993. Springer-Verlag.
258. M. Wilding. Robust Computer System Proofs in PVS. In C. M. Holloway and K. J. Hayhurst, editors, *4th NASA Langley Formal Methods Workshop*, number 3356 in NASA Conference Publication, 1997.
259. 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 4th International Conference on Formal Methods in Computer-Aided Design (FMCAD 2002)*, volume 2517 of *LNCS*, pages 70–87, Portland, OR, 2002. Springer-Verlag.
260. W. D. Young. A Verified Code Generator for a Subset of Gypsy. Technical Report 33, Computational Logic Inc., 1988.
261. 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.

Index

G

H

I

J

K