View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by UT Digital Repository

Copyright

by

Erik Henry Reeber

2007

The Dissertation Committee for Erik Henry Reeber
certifies that this is the approved version of the following dissertation:

Combining Advanced Formal Hardware Verification Techniques

Committee:

Warren A. Hunt, Jr., Supervisor

E. Allen Emerson

Stephen W. Keckler

J Strother Moore

Anna Slobodova

Combining Advanced Formal Hardware Verification Techniques

by

Erik Henry Reeber, B.S., M.S.

Dissertation

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

Doctor of Philosophy

The University of Texas at Austin

December 2007

To my parents, Henry and Karen Reeber, and my fiancée, Carrie Pankrast, for all their love, guidance, and support.

Acknowledgments

Most of all, I would like to thank my thesis advisor, Warren Hunt. Warren always has the amazing ability to give me what I need, before I even ask for it. Furthermore, Warren has been a source of constant encouragement and guidance, without which I never would have started this dissertation, let alone completed it.

I would also like to thank the rest of my dissertation committee, Allen Emerson, Steve Keckler, J Moore, and Anna Slobodova, for all the time and energy they spent reviewing my research and for their great feedback both on the dissertation itself and the earlier dissertation proposal. Anna in particular provided me with copious notes that have significantly improved the quality of this dissertation. Thanks also to Sandip Ray, Simha Sethumadhavan, and Jun Sawada for providing excellent feedback on portions of this dissertation.

A number of professors at the University of Texas have influenced my work. My early collaboration with Calvin Lin and then Kathryn Mckinley helped to shape my research mind set. Later, my classes with Allen Emerson, Warren Hunt, J Moore, and Doron Peled helped to shape my research goals in formal verification. J Moore especially inspired me to work in this field, shaped my research goals, and provided feedback on my early endeavors.

I have had the pleasure of collaborating with and receiving insight from many people while working on my dissertation research. At the University of Texas, I have had the occasional privilege of collaborating with Matt Kaufmann, J Moore, Sandip Ray, Simha Sethumadhavan, and Vinod Viswanath. Also, while at IBM, I benefited greatly from Jun

Sawada's insight, as well as some useful help and guidance from Damir Jamsek, Jason Baumgartner, Hari Mony, and Viresh Paruthi. I have also had many useful discussions with Pete Manolios and Sudarshan Srinivasan.

The ACL2 theorem proving group in Austin is a constant source of encouragement, feedback, and camaraderie. My officemate, Sandip Ray, has served as a valuable sounding board, a tireless collaborator, and a useful skeptic. My lunch outings with Matt Kaufmann and Sandip Ray have served up much fruitful discussion, along with the endless pho. Others in the group whom I have had the pleasure of interacting with a great deal include Jared Davis, John Erickson, Jeff Golden, Warren Hunt, Robert Krug, Hanbing Liu, J Moore, Serita Neleson, Grant Passmore, David Rager, Matyas Sustik, Sol Swords, and Bill Young. While not usually in Austin, I always enjoy the visits from Julien Schmaltz and Eric Smith as well. Also, I want to acknowledge the great hidden supplier of the cookies, Jo O'Neil-Moore.

I also thank the members of the TRIPS project for allowing me access to the TRIPS design, answering my questions, and allowing me to participate some in the design's development. I would especially like to thank Doug Burger, Raj Desikan, Paul Gratz, Steve Keckler, Simha Sethumadhavan, and Bill Yoder.

This dissertation and the research behind it was made significantly easier by the great staff at the University of Texas. I relied on Gloria Ramirez for guidance on all procedural issues, and she never failed to know exactly what I needed to do. I also greatly appreciate the help of staff members Lindy Aleshire, Kata Carbone, Carol Hyink, Gem Naivar, Patti Spencer, and Katherine Utz.

My friends and family have been an essential source of support and inspiration. Thanks to my best friends, Wakova Carter and Judah de Paula; my fiancée, Carrie Pankrast; my dad, Hank Reeber; my mom, Karen Reeber; and my sister, Lisa Reeber. I love you all.

Last but certainly not least, I want to acknowledge the generous financial support I have received. I initially worked with the Tera-op Reliable and Intelligently Adaptive

Processing System, funded by the Defense Advanced Research Projects Agency (DARPA) grant #F33615-01-C-1892. Most of my research was funded by the DARPA's Productive, Easy-to-use, Reliable Computing System (PERCS) project, grant #NBCH30390004. Significant portions of my research were also funded by the National Science Foundation's Cyber Trust program, as part of the University of Texas at Austin's Center for Information Assurance and Security. Some of my research also followed from my employment at International Business Machines (IBM) Corporation.

Erik Henry Reeber

The University of Texas at Austin

December 2007

Combining Advanced Formal Hardware Verification Techniques

Publication No. _____________________

Erik Henry Reeber, Ph.D.
The University of Texas at Austin, 2007

Supervisor: Warren A. Hunt, Jr.

This dissertation combines formal verification techniques in an attempt to reduce the human effort required to verify large systems formally.

One method to reduce the human effort required by formal verification is to modify general-purpose theorem proving techniques to increase the number of lemma instances considered automatically. Such a modification to the forward chaining proof technique within the ACL2 theorem prover is described.

This dissertation identifies a decidable subclass of the ACL2 logic, the Subclass of Unrollable List Formulas in ACL2 (SULFA). SULFA is shown to be decidable, i.e., there exists an algorithm that decides whether any SULFA formula is valid. Theorems from first-order logic can be proven through a methodology that combines interactive theorem proving with a fully-automated solver for SULFA formulas. This methodology has been applied to the verification of components of the TRIPS processor, a prototype processor designed and fabricated by the University of Texas and IBM. Also, a fully-automated procedure for the Satisfiability Modulo Theory (SMT) of bit vectors is implemented by combining a solver for SULFA formulas with the ACL2 theorem prover's general-purpose rewriting proof technique.

A new methodology for combining theorem proving and model checking is presented, which uses a unique "black-box" formalization of hardware designs. This methodology has been used to combine the ACL2 theorem prover with IBM's SixthSense model checker and applied to the verification of a high-performance industrial multiplier design.

A general-purpose mechanism has been created for adding external tools to a general-purpose theorem prover. This mechanism, implemented in the ACL2 theorem prover, is capable of supporting the combination of ACL2 with both SixthSense and the SAT-based SULFA solver.

A new hardware description language, DE2, is described. DE2 has a number of unique features geared towards simplifying formal verification, including a relatively simple formal semantics, support for the description of circuit generators, and support for embedding non-functional constructs within a hardware design.

The composition of these techniques extend our knowledge of the languages and logics needed for formal verification and should reduce the human effort required to verify large hardware circuit models.

Contents

Acknowledgments **v**

Abstract **viii**

Chapter 1 Introduction **1**

Chapter 2 An Overview of Formal Verification **5**

2.1 Introduction 5

2.2 Fully-Automatic Methods 5

2.2.1 Boolean Satisfiability (SAT) Solvers 6

2.2.2 Satisfiability Modulo Theory (SMT) Solvers 7

2.2.3 Temporal Logic 8

2.3 Interactive Techniques 10

2.4 Combined Approaches 11

Chapter 3 The ACL2 Logic and Theorem Prover **13**

3.1 Example 14

3.2 Syntax 16

3.3 Mixing Math and Lisp Notation 21

3.4 Primitives 23

3.5 The Definition Principle 27

3.5.1 Encapsulation 29
3.6 The Theorem Prover 30
3.6.1 Forward Chaining 31
3.6.2 Rewriting 32
3.6.3 Overview of other Main Techniques 33
3.6.4 User Guidance 35
3.6.5 Meta Reasoning 36

Chapter 4 Increasing Free Variable Instantiation During Forward Chaining 39
4.1 Introduction 39
4.2 Example 40
4.3 Forward Chaining Implementation 42
4.4 Results 44
4.5 Summary 46
4.6 Development and Bibliographic Notes 47

Chapter 5 The Subclass of Unrollable List Formulas in ACL2 (SULFA) 48
5.1 Introduction 48
5.2 Intuition Behind SULFA 49
5.3 Simplified ACL2 Logic 51
5.4 SULFA Recognizer 52
5.4.1 Termination 62
5.5 Efficient SULFA Recognizer 64
5.6 Results 70
5.7 Summary 71
5.8 Development and Bibliographic Notes 71

Chapter 6 A SULFA Decision Procedure 73
6.1 Introduction 73

6.2 Unrolling SULFA Formulas 74
6.2.1 Correctness 79
6.3 Removing Uninterpreted Functions 83
6.3.1 Correctness 86
6.4 A Decision Procedure for SULFA Core Primitives 88
6.4.1 Correctness 103
6.5 Counterexample Generation 110
6.6 Summary 111

Chapter 7 Developing a SAT-Based SULFA Solver **113**
7.1 Introduction 113
7.2 Translating Nested If Terms to CNF 114
7.3 Boolean SULFA Predicates 116
7.4 Developing an Efficient SAT-Based Proof Procedure 122
7.5 Example 127
7.6 Notes on Complexity and Efficiency 135
7.7 Uninterpreted Functions 137
7.8 Summary 138
7.9 Development and Bibliographic Notes 138

Chapter 8 The Verification of the TRIPS Processor's Data-Tile Protocol Implementation **140**
8.1 Introduction 140
8.2 Overview of the TRIPS Processor 141
8.3 Overview of the Data-Tile Protocol 143
8.4 Formal Verification Methodology 145
8.4.1 Verification of Safety Properties 147
8.4.2 Verification of Liveness Properties 149

8.5 Verification of the Exception Protocol 151
8.5.1 ACL2 Model of the Exception Protocol 153
8.5.2 Proof of Exception-Safety 154
8.5.3 Proof of Exception-Liveness 155
8.6 Verification of the Store Protocol 157
8.6.1 ACL2 Model of the Store Protocol 159
8.6.2 Proof of Store-Safety 161
8.6.3 Proof of Store-Liveness 163
8.7 Analysis 165
8.8 Summary 167
8.9 Development and Bibliographic Notes 167

Chapter 9 The SULFA SMT Solver **170**
9.1 Introduction 170
9.2 Introductory Example 171
9.3 The Standard Bit-Vector Theory 173
9.4 Implementation 176
9.4.1 Syntactic Translation and Negation 176
9.4.2 ACL2 Simplification 180
9.4.3 ET-ACL2 to NBV Translation 183
9.4.4 Common Subexpression Elimination 183
9.4.5 Uninterpreted Function Removal 184
9.4.6 SAT-Based Procedure 184
9.5 Adding New Functions and Rewriting Strategies 185
9.6 Results 186
9.7 Summary 188
9.8 Development and Bibliographic Notes 188

Chapter 10 Integrating ACL2 with the SixthSense Model Checker **189**

10.1 Introduction 189

10.2 ACL2VHDL Property 190

10.3 Overview of ACL2SIX 192

10.3.1 Soundness 194

10.4 Overview of Multiplier Verification 196

10.5 Summary 197

10.6 Development and Bibliographic Notes 197

Chapter 11 A General-Purpose Mechanism for Integrating External Tools with ACL2 **199**

11.1 Introduction 199

11.2 Verified Clause Processors 200

11.2.1 Example 203

11.3 Basic Unverified Clause Processors 207

11.3.1 Applications 208

11.4 Unverified Clause Processors with Implicit Theories 209

11.5 Summary 210

11.6 Development and Bibliographic Notes 211

Chapter 12 The DE2 Language **213**

12.1 Introduction 213

12.2 Introductory Example 215

12.3 Formal Syntax 218

12.4 DE2 Semantics 218

12.4.1 Limitations of the Two Pass Model 224

12.5 The DE2 Verification System 225

12.6 Circuit Generator Example 227

12.7 Summary 232
12.8 Development and Bibliographic Notes 233

Chapter 13 Future Directions **235**
13.1 Introduction 235
13.2 Expanding SULFA 235
13.3 Improving Efficiency 236
13.4 Verification of Larger Hardware Modules 237
13.5 Undecidable Domains 237
13.6 Verified SAT-Based Procedure 238

Chapter 14 Conclusion **240**

Bibliography **245**

Vita **258**

Chapter 1

Introduction

Even a single bug in a hardware design, if not detected until late in the design process, can be extremely costly. Intel, for example, estimated the cost of the Pentium® bug in 1994 as 500 million dollars [14]. Thus, a lot of time and energy is spent attempting to eliminate bugs in industrial hardware designs as early as possible.

The primary means for verifying hardware today is simulation. Exhaustive simulation is impossible, since the number of input and state combinations in most designs is enormous. Instead, simulations are carefully engineered to catch as many potential problems as possible. Nevertheless, there is always the potential for a bug to escape all simulated cases. And as hardware designs become ever more complex, the cost of keeping the possibility of a bug acceptably low is always increasing.

An alternative to simulation is formal verification, which proves that a design satisfies its formal specification for all possible input and state combinations. The formal verification of entire processors has been shown to be possible. In 1985, Warren Hunt used formal verification to prove that the FM8501 processor implements a formal specification of its instruction set architecture [27]. While the FM8501 was never fabricated, another processor that was fabricated, the FM9001 processor, was later proven to implement its specification as well, by Bishop and Hunt [29].

While the formal verification of entire processor designs is possible, the cost of performing formal verification has always been too high to make it practical for most industrial designs. The trend, however, because of the increasing complexity of hardware designs, the increasing power of formal verification tools, and the increasing expertise with formal verification tools, has been towards more formal verification. After the Pentium® bug in 1994, for example, the decision was made by both Intel and AMD to use formal methods to verify their floating-point units in future designs[60, 72].

In order to continue the trend, we need to develop new hardware description languages, improve formal verification tools, and build expertise in formal verification strategies for state-of-the-art hardware designs. In this dissertation, we make progress on each of these fronts. We have developed a new hardware description language, DE2, designed to simplify formal verification. We have developed new algorithms, combining theorem proving with decision procedures, to increase the automation and flexibility of formal verification tools. Finally, we have applied these techniques to components of the TRIPS processor, a complex multi-core research processor, with unique features designed to address future challenges in microprocessor design.

The dissertation begins with an overview of relevant formal methods in Chapter 2. Chapter 3 provides a more detailed overview of the ACL2 theorem prover in particular, a powerful interactive formal tool, that has been successfully applied to the verification of industrial hardware designs [72, 21].

One way to improve the ACL2 theorem prover, and similar theorem provers, is to decrease the amount of user guidance required to prove theorems. Chapter 4 describes a relatively small modification which does just that. The modification increases the amount of search in one of the standard proof techniques, called forward chaining, allowing more theorems to be proven automatically at the cost of requiring more time to find a given proof.

Chapter 5 defines the Subclass of Unrollable List Formulas in ACL2 (SULFA). SULFA can be recognized efficiently, verified fully automatically (in principle), and con-

tains a definition principle that allows it to be extended with user-defined functions. Chapter 5 formally defines SULFA and presents an efficient procedure for determining whether an arbitrary ACL2 formula is in SULFA. Chapter 6 then shows that SULFA is decidable, by presenting a relatively simple and inefficient decision procedure for SULFA and a proof sketch of its correctness. Chapter 7 then describes a more efficient SULFA solver based on Boolean satisfiability solvers (SAT solvers).

Chapter 8 applies the SAT-based SULFA solver to the formal verification of a data-tile communication protocol implementation within the TRIPS processor. The data-tile protocol is unique to the TRIPS processor, and is required directly due to the processor's unique method of addressing future architecture challenges. Thus, the verification of the data-tile protocol shows both the feasibility of our approach and helps build formal verification expertise beyond the units verified in industrial microprocessors today.

Chapter 9 describes how we have used our approach to develop a fully-automatic technique for the standard Satisfiability Modulo Theory (SMT) of bit vectors. The SMT theory of bit vectors is part of a set of recently developed standard theories. Every previously created SMT solver uses a special purpose simplifier followed by a fully-automatic technique. Our approach, however, achieves a greater degree of flexibility by using a general purpose theorem prover to simplify the problem before passing the problem to a fully-automated technique. Furthermore, by creating an SMT solver for bit vectors, we were able to apply our approach, combining general purpose theorem proving with a SAT-based solver for SULFA formulas, to a large benchmark suite of problems, mostly derived from hardware verification.

Chapter 10 describes a different hardware verification approach, which combines the ACL2 theorem prover with another powerful verification tool, IBM's SixthSense model checker. These two tools combine synergistically, since ACL2 is scalable and contains powerful arithmetic proof techniques, while SixthSense is fully automatic and contains built-in support for industrial hardware description languages. Chapter 10 describes how

the two tools are combined and described an application of this approach to the verification of an industrial high-performance multiplier.

Both Chapter 7 and Chapter 10 involve extending ACL2 with proof techniques that use external tools. Each of these extensions originally required modifications to the source code of the theorem prover. Chapter 11, describes a new general-purpose mechanism for extending ACL2 with external tools at run time, which is powerful enough to support both of these new proof engines. Furthermore, this mechanism allows users to control and identify exactly which mechanisms are available in their proofs.

Finally, Chapter 12 describes a new hardware description language, DE2. DE2 has a number of features that make it well-suited for formal verification, including a simple two-pass semantics, user-defined primitives, and annotations as first-class objects. The formal semantics of DE2 are specified within the ACL2 logic, allowing the proof of DE2 programs. We have developed an automated procedure for translating DE2 circuits into simplified ACL2 models, which, along with the ACL2 model, produces a proof of equivalence between the ACL2 model and the original DE2 circuit, relative to the DE2 language semantics. The DE2 language is also used as part of the verification of the data-tile protocol implementation described in Chapter 8.

One of the strengths of our work is that lots of future improvements and applications are possible. Chapter 13 describes a few of these future directions. The decidable subclass of the ACL2 logic should be enlarged, and a number of optimizations have the potential to improve the performance of the SAT-based procedure. Also, we feel it would be worthwhile to continue to apply our techniques to hardware designs beyond the scope of what is traditionally formally verified in industry.

Chapter 2

An Overview of Formal Verification

2.1 Introduction

This chapter presents an overview of formal verification and its application to hardware designs. As is common, we divide formal methods into two categories: fully-automatic methods, exemplified by model checking, and interactive methods, exemplified by general-purpose theorem provers.

2.2 Fully-Automatic Methods

Ideally, a hardware design would be proven to satisfy its specification entirely automatically. Thus, the human effort would be limited to formally expressing the design and its specification. In practice, methods capable of performing such automatic hardware verification exist, but suffer from problems with scalability and expressiveness. Often small designs can be verified quickly and automatically, but as the number of input and state bits increase, exponentially more time and/or memory is required. In order to ensure full automation, the expressiveness of the logic is limited, which can make expressing a given design and its specification more difficult or, in some cases, impossible.

This section provides an overview of various fully-automatic techniques, starting with those using the least expressive logic and ending with those using the most expressive logic.

2.2.1 Boolean Satisfiability (SAT) Solvers

Boolean satisfiability solvers, which we refer to as SAT solvers, determine whether a set of variables can be instantiated with Boolean value such that a formula is true. The formula solved must be in conjunctive normal form (CNF), such as the one in the following example:

$$(a \vee b) \wedge (a \vee c) \wedge (\neg b \vee \neg c) \wedge (b \vee c).$$

The above example has two satisfying instances, $[a \mapsto \textbf{true}, b \mapsto \textbf{true}, c \mapsto \textbf{false}]$ and $[a \mapsto \textbf{true}, b \mapsto \textbf{false}, c \mapsto \textbf{true}]$. A SAT solver would return that the above formula is satisfiable, along with one of the two possible satisfying instances.

In the context of a CNF formula, we say that a *literal* is either a variable or its negation. For example, b and $\neg b$ are literals. A *clause* is then a disjunction of literals, such as $(\neg b \vee \neg c)$. A CNF formula is then simply a conjunction of clauses.

SAT solving is an NP-complete problem, and therefore suffers from exponential complexity as the number of Boolean variables increase. Nevertheless, problems from hardware verification are regularly translated into CNF problems with hundreds of thousands of variables and solved by SAT solvers. Chapter 7 describes such an algorithm, which translates ACL2 formulas into CNF to be solved by SAT. Furthermore, Chapter 8 uses SAT solvers, with ACL2, to verify a component of a processor design. SAT solvers are also used within the SixthSense model checker, which was integrated with the ACL2 theorem prover as described in Chapter 10.

There is a standard input format for SAT solvers [74] and an annual competition to compare the performance of SAT solvers [86]. Most of the fastest solvers today use modifications of the Davis, Putnam, Logemann and Lovel (DPLL) algorithm [11], first

described in 1962. Some of the more recent modifications to this algorithm are described by Moskewicz et al. [55]. The SAT solvers used in the implementations discussed in this dissertation are zChaff [88] and minisat [87].

2.2.2 Satisfiability Modulo Theory (SMT) Solvers

Satisfiability Modulo Theory (SMT) solvers are similar to SAT solvers in that, given a formula, an SMT solver determines whether a satisfying substitution exists under which a the formula is true. SMT solvers, however, are less restrictive than SAT solvers with respect to the formulas accepted. Multiple standard SMT theories exist, supporting formulas involving linear arithmetic, arrays, uninterpreted functions, bit vectors, and quantifiers. For example, the following formula can be represented using the quantifier-free SMT theory of uninterpreted functions:

$$\mathsf{f}(a) \wedge \neg \mathsf{f}(b) \wedge (a = b).$$

The above formula is unsatisfiable, since all functions return the same value given the same inputs.

A similar example including linear arithmetic is as follows:

$$\mathsf{f}(a) \wedge \neg \mathsf{f}(b) \wedge (a = b + 1).$$

The above formula has an infinite number of satisfying instances, including when $a \mapsto 1$, $b \mapsto 0$, and

$$\mathsf{f}(x) \triangleq \begin{cases} \textbf{false}, & \text{if } x = 0, \\ \textbf{true}, & \text{otherwise.} \end{cases}$$

Since SMT solvers have a less restrictive input language than SAT solvers, less translation is required to translate problems from hardware verification, and other domains, into SMT problems. Another strength of SMT theories is that solvers for disjoint theories

Table 2.1: LTL Temporal Operators

Operation	Description
$\Diamond E$	E is *eventually* true.
$\Box E$	E is *always* true.
$\bigcirc E$	E is true during the *next* time step.
$E_1 \ \mathcal{U} \ E_2$	E_1 is true *until* E_2 is true (and E_2 must eventually be true).
$E_1 \ \mathcal{R} \ E_2$	E_2 is either true forever or until E_1 is true. Thus, E_1 *releases* E_2.

often can be combined using the techniques developed by Nelson and Oppen [57] and Shostak [82]. Since 2005, an annual competition has existed comparing the performance of SMT solvers in various standard SMT theories. Chapter 9 describes the SMT quantifier-free theory of bit vectors with uninterpreted functions in more detail, as well as a solver we have developed for this theory.

2.2.3 Temporal Logic

Fully-automatic techniques have also been developed to determine whether finite state machines satisfy properties that hold over time using temporal logics [10, 65]. Numerous such logics exists, including, in order from less expressive to more expressive, Linear Temporal Logic (LTL), Computation Tree Logic (CTL), and μ-Calculus.

In every temporal logic, the propositional calculus is combined with temporal operators. The LTL temporal operators are shown in Table 2.1. An example of an LTL formula then is the following:

$$\Box(\mathbf{B} \rightarrow \Box\mathbf{B}).$$

which states that once **B** is high, then **B** continues to be high for all future cycles. The property is satisfied by the circuit in figure 2.1.

CTL expands on LTL by adding operators that reason about program branches. This allows statements such as “For all time, there exists some program branch on which P occurs in the future.” The μ-Calculus is also able to express such properties, through the

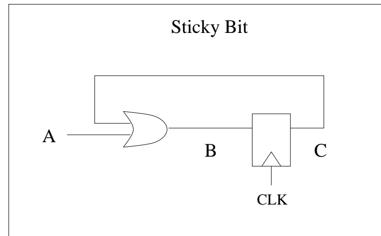


Figure 2.1: A simple example circuit, in which a high bit **A** causes bit **B** and bit **C** to be high for all later times.

concept of least fixpoints. No knowledge, however, of CTL or μ-Calculus is required in this dissertation.

Another fully-automated technique worthy of note is symbolic evaluation, which can be seen to verify a subset of LTL. Using symbolic evaluation, properties of the form $\Box(E)$ can be verified, where E is an LTL term containing no temporal operators other than $\bigcirc$. Thus, $\Box(\mathbf{B} \rightarrow \bigcirc\mathbf{B})$, is an example of a symbolic evaluation formula that is satisfied by the circuit in Figure 2.1. An extension of symbolic evaluation, called Symbolic Trajectory Evaluation (STE), has been used successfully in the verification of significant industrial designs at Intel [60]. STE is similar to symbolic evaluation, but beyond the traditional two values of Boolean logic, STE contains a “do not care” value for wires that are not being driven to any value and an “over constrained” value for wires that are being driven to both true and false simultaneously.

Industrially successful solvers have also been developed for LTL and CTL. IBM’s SixthSense model checker, which is discussed in more detail in Chapter 10, is one example of an industrially successful LTL model checker [52]. The SMV model checker, designed by Carnegie Mellon University and Cadence Berkeley Laboratories, is an example of an industrially successful CTL model checker [47].

Fully-automatic methods for hardware verification generally suffer from scalability problems. While they may work very well on small designs, as the design gets larger the

time required to fully verify the design increases at a much faster rate. Furthermore, certain circuits, such as multipliers are particularly difficult to verify automatically. Thus, no fully-automatic method can completely verify large industrial processor designs.

2.3 Interactive Techniques

To verify large designs formally, some human guidance is required. Traditionally, this takes the form of mathematical proof, checked by computer. A number of automated theorem provers exists that aid in the creation and checking of mathematical proofs. Some examples of theorem provers that have been used for hardware verification include HOL4 [20], HOL-Light [25], Isabelle [59], Coq [15], PVS [62], and ACL2 [41]. The ACL2 theorem prover is described in more detail in Chapter 3.

There is strong evidence that theorem proving can scale to large designs. An entire processor design was verified by a theorem prover in 1985 [27]. While that design was not fabricated, a design that was fabricated was later verified [29]. This design was missing some of the features of modern processors, such as out-of-order execution, but a model of a more complex processor design containing more features, including out-of-order execution, has also been verified [75].

Another advantage of theorem proving is that the logics used by theorem provers are generally more expressive than the logics used by fully-automatic tools. Most automatic tools, for example, cannot formally verify reconfigurable designs, if an infinite number of configurations is possible. Such designs are especially important in the development of hardware design libraries.

The disadvantage of theorem proving is that human guidance is required. Thus, even proofs about small designs often require considerable amount of human effort to develop. Furthermore, it is common for months of training to be required before someone can be a productive user of any given theorem prover.

2.4 Combined Approaches

While we divide formal verification tools into fully automatic and interactive approaches, the truth is that there is a lot of interplay between the two techniques. Effective fully-automatic model checking techniques are often inspired by the methods used during a mathematical proof of correctness of similar designs. Interactive approaches also often make use of techniques originating from fully-automatic methods. Furthermore, considerable research has gone into directly combining fully automatic and interactive formal verification tools.

Too many deductions occur within an automated theorem prover to rely on external decision procedures to perform those deductions [7]. However, external decision procedures can be used to prove user-generated lemmas efficiently that would otherwise require user guidance. Thus, most combinations of fully automatic and interactive techniques are "coarse-grained" solutions, where theorem proving is used to combine user-generated lemmas that are verified by fully-automatic procedures. Examples of such integrations include the integration of the ACL2 theorem prover with UCLID [44] and the integration of the HOL theorem prover with the VOSS model checker [36].

An industrially successful combination of fully automatic and interactive tools includes the integration of an STE solver with a theorem prover for higher order logic (HOL), which was used to verify Intel's Pentium® floating-point unit [60]. Due to the relatively low expressive power of STE, however, significant amount of human guidance is often required to reduce hardware specifications to STE properties. In particular, a target property often must be strengthened until it is an inductive invariant. Jacobi addresses this problem by using a theorem prover to compose more expressive temporal logic properties in the verification of an out-of-order scheduler implementation [35]. Ray addresses the problem a different way, by developing techniques to automatically generate inductive invariants [68].

Other work combining theorem proving and fully-automatic tools includes the μ-calculus model checkers that have been developed within PVS and ACL2 [43, 61]. Also,

Cadence SMV, an industrially successful model checker [50], includes some support for user-guided composition.

A disadvantage with most combined approaches is that it often necessitates working in a logical framework, such as the μ-calculus, that is not ideal for human reasoning. The logic of automated theorem provers are designed to facilitate human reasoning. The logic of model checkers, on the other hand, are usually only well-suited for specification, not for deduction. Our work attempts to address this issue by developing automated techniques within a natural subclass of a logic developed for human reasoning.

Chapter 3

The ACL2 Logic and Theorem Prover

ACL2, or A Computational Logic for Applicative Common Lisp (*ACL*^{2}), is a programming language, a mathematical logic, and an automated theorem prover. As a programming language, it is a subset of Common Lisp; as a logic, it is a form of quantifier-free first-order logic; and as a theorem prover, it is a tool designed to facilitate the creation and checking of proofs. ACL2 has been successfully applied to formal verification projects within industry, including the AMD Athlon ™ floating-point unit [72] and the Rockwell Collins AAMP7 Separation Kernel [21]. For a complete introduction to the theorem prover, with exercises and examples, see *Computer-Aided Reasoning: An Approach* by Kaufmann, Manolios, and Moore [41].

Both the ACL2 logic and the theorem prover are heavily incorporated into the work described in this dissertation. This chapter overviews the key concepts and notation from ACL2 necessary to understand the rest of the dissertation. We begin with an example of an ACL2 definition and proof in Section 3.1. Section 3.2 then describes the ACL2 syntax and the mathematical notation used in most of the dissertation to represent ACL2 definitions and formulas. Section 3.4 and 3.5 discuss the initial set of axioms in ACL2 and ACL2's system

for soundly extending them. Finally, Section 3.6 overviews the ACL2 theorem prover and some of its proof techniques.

3.1 Example

The ACL2 logic is a form of first-order logic, with recursively defined functions. The primary data structure is trees, where the function $\mathsf{cons}(x, y)$ constructs a tree with the left branch x and right branch y, $\mathsf{car}(x)$ returns the left branch of the tree x, $\mathsf{cdr}(x)$ returns the right branch of the tree x, and $\mathsf{consp}(x)$ returns whether the object x is a tree. Lists are usually represented as trees where the first element of the list is the left branch and the list containing the remaining elements is the right branch. For example, the predicate $\mathsf{in}(a, x)$, which checks whether an element a is in a list x, and the function $\mathsf{del}(a, x)$, which removes a from x, can be defined in the ACL2 logic using the following recursive definitions:

$$\mathsf{in}(a, x) \triangleq \begin{cases} \textbf{false}, & \text{if } \neg\mathsf{consp}(x) \\ \textbf{true}, & \text{if } a = \mathsf{car}(x) \\ \mathsf{in}(a, \mathsf{cdr}(x)), & \text{otherwise.} \end{cases}$$

$$\mathsf{del}(a, x) \triangleq \begin{cases} x, & \text{if } \neg\mathsf{consp}(x) \\ \mathsf{cdr}(x), & \text{if } a = \mathsf{car}(x) \\ \mathsf{cons}(\mathsf{car}(x), \mathsf{del}(a, \mathsf{cdr}(x))), & \text{otherwise.} \end{cases}$$

Once $\mathsf{in}(a, x)$ and $\mathsf{del}(a, x)$ are defined, the predicate $\mathsf{perm}(x, y)$, which returns whether x is a permutation of y, can be defined as:

$$\mathsf{perm}(x, y) \triangleq \begin{cases} \neg\mathsf{consp}(y), & \text{if } \neg\mathsf{consp}(x) \\ \textbf{false}, & \text{if } \neg\mathsf{in}(\mathsf{car}(x), y) \\ \mathsf{perm}(\mathsf{cdr}(x), \mathsf{del}(\mathsf{car}(x), y), & \text{otherwise.} \end{cases}$$

A property one might wish to prove about the permutation predicate is that permutation is transitive. This can be written as the following first-order property, expressible in ACL2:

perm-transitivity :

$$\mathsf{perm}(x,y) \land \mathsf{perm}(y,z) \rightarrow \mathsf{perm}(x,z)$$

Note that variables occurring in ACL2 formulas, such as x, y, and z above, are implicitly universally quantified.

The ACL2 theorem prover cannot prove *perm-transitivity* automatically. However, ACL2 does attempt to prove *perm-transitivity* by induction on the size of the list x, which is a good strategy. By induction on x and the definition of $\mathsf{perm}(x,y)$, we can reduce *perm-transitivity* to the following two lemmas:

in-one-but-not-other :

$$\mathsf{in}(a,x) \land \neg\mathsf{in}(a,y) \rightarrow \neg\mathsf{perm}(x,y)$$

perm-implies-perm-minus-a :

$$\mathsf{perm}(x,y) \rightarrow \mathsf{perm}(\mathsf{del}(a,x), \mathsf{del}(a,y))$$

where *in-one-but-not-other* states that if an element is in x but not y, then x is not a permutation of y and *perm-implies-perm-minus-a* states that if x is a permutation of y, then x and y are still permutations after some element a is deleted from both of them.

Neither *in-one-but-not-other* nor *perm-implies-perm-minus-a* can be proven automatically by the theorem prover. However, by induction on the size of x and the definitions of the functions $\mathsf{in}(a,x)$ and $\mathsf{perm}(x,y)$, *in-one-but-not-other* can reduced to the following lemma:

in-minus-a-implies-in :

$$\mathsf{in}(a, \mathsf{del}(b,x)) \rightarrow \mathsf{in}(a,x)$$

which states that if an element a is in the list x minus some element b, then a is in x itself. The lemma *in-minus-a-implies-in* is proven automatically by the theorem prover by induction on the size of x and the definitions of the functions $\mathsf{del}(a, x)$ and $\mathsf{in}(a, x)$.

Thus to prove the transitivity of permutation, only the lemma *perm-implies-perm-minus-a* still needs to be proven. Again, by induction on x and the definitions of $\mathsf{del}(a, x)$ and $\mathsf{perm}(x, y)$, *perm-implies-perm-minus-a* is reduced to the following two lemmas:

in-implies-in-minus-a :

$$\mathsf{in}(a, x) \land a \neq b \rightarrow \mathsf{in}(a, \mathsf{del}(b, x))$$

del-symmetry :

$$\mathsf{del}(a, \mathsf{del}(b, x)) = \mathsf{del}(b, \mathsf{del}(a, x))$$

The lemma *in-implies-in-minus-a* states that if an element is in the list x and then it is in x with any other element removed. The lemma *del-symmetry* states that the order of the removal of two elements is not significant. Both *in-implies-in-minus-a* and *del-symmetry* can be proven automatically by induction on the size of x and the definitions of the functions $\mathsf{del}(a, x)$ and $\mathsf{in}(a, x)$.

Thus, ACL2 can prove *perm-transitivity* with some user guidance in the form of five lemmas. Note that the proof closely follows the proof a human might create to prove the transitivity of the $\mathsf{perm}(x, y)$ predicate.

3.2 Syntax

The examples in the previous section use standard math notation for first-order logic to represent ACL2 formulas, which is possible since ACL2 is a form of first-order logic. The syntax of the ACL2 logic actually, however, uses Lisp syntax to denote terms. A formula is created from terms using the equality predicate =, and propositional logic, which are axiomatized in the standard manner [38]. For example, the following is an ACL2 formula:

1. $\texttt{'T} \neq \texttt{'NIL}$
2. $\texttt{X} = \texttt{Y} \rightarrow \texttt{(EQUAL X Y)} = \texttt{'T}$
3. $\texttt{X} \neq \texttt{Y} \rightarrow \texttt{(EQUAL X Y)} = \texttt{'NIL}$
4. $\texttt{X} = \texttt{'NIL} \rightarrow \texttt{(IF X Y Z)} = Z$
5. $\texttt{X} \neq \texttt{'NIL} \rightarrow \texttt{(IF X Y Z)} = Y$

Figure 3.1: A subset of axioms in ACL2's ground zero theory.

$$\texttt{X} = \texttt{Y} \rightarrow \texttt{(CONS X Y)} = \texttt{(CONS X X)}.$$

Since the axioms shown in Figure 3.1 are a part of every ACL2 theory, every ACL2 formula can be expressed as a formula of the form:

$$\alpha \neq \texttt{'NIL}$$

for some ACL2 term α. For example, the ACL2 formula

$$\texttt{X} = \texttt{Y} \rightarrow \texttt{(CONS X Y)} = \texttt{(CONS X X)}$$

can be written as the equivalent ACL2 formula

$$\texttt{(IF (EQUAL X Y) (EQUAL (CONS X Y) (CONS X X)) 'T)} \neq \texttt{'NIL}.$$

Thus in this dissertation, we sometimes assume that all ACL2 formulas are of the form $\alpha \neq$ 'NIL. In fact, users of the ACL2 theorem prover never create ACL2 formulas, only ACL2 terms. Also, a term α may be used in place of a formula, with the implicit formula being $\alpha \neq$ 'NIL. Note that 'T and 'NIL represent the Boolean constants **true** and **false** respectively.

ACL2 also uses Lisp syntax to specify definitional axioms, such as those in Figure 3.2. These definitions passed directly to the theorem prover in the same manner that they would be passed to a Lisp interpreter. Each block of data passed to the theorem prover,

```
(DEFUN DEL (A X)
  (IF (NOT (CONSP X))
      X
    (IF (EQUAL A (CAR X))
        (CDR X)
      (CONS (CAR X) (DEL A (CDR X))))))

(DEFUN IN (A X)
  (IF (NOT (CONSP X))
      NIL
    (IF (EQUAL A (CAR X))
        T
      (IN A (CDR X)))))

(DEFUN PERM (X Y)
  (IF (NOT (CONSP X))
      (NOT (CONSP Y))
    (IF (NOT (IN (CAR X) Y))
        NIL
      (PERM (CDR X) (DEL (CAR X) Y)))))
```

Figure 3.2: The definitions from the example in Section 3.1 in the actual ACL2 syntax, rather than standard math notation.

such as each definition in Figure 3.2 is called an *event*. As in Lisp, a definition event has the form (*name formals body*), which defines a function named *name* with formal parameters *formals* and body *body*.

Another ACL2 event is the DEFTHM event, which passes a conjecture to the theorem prover. The DEFTHM event has the form (DEFTHM *name expr opt*), where *name* is the name of the theorem, *expr* is a Lisp expression representing the conjecture, and *opt* consists of zero or more optional arguments to the theorem prover. For example, the *perm-transitivity* theorem from the previous section, is written as the following DEFTHM event:

```
(DEFTHM PERM-TRANSITIVITY
  (IMPLIES (AND (PERM X Y) (PERM Y Z))
           (PERM X Z)))
```

Note that the conjecture contains variables X, Y, and Z, which are implicitly universally quantified. The conjecture is said to be valid if there is be no substitution of values in the variables such that it evaluates to 'NIL.

To be more precise, we define the following terminology:

- An *ACL2 symbol* is a string of numbers and characters starting with a character, such as CAR or MY-320463TH-SYMBOL.
- An *ACL2 constant* is of the form (QUOTE *expr*) or '*expr*, where *expr* can be a rational number, symbol, Lisp expression, or various other expressions. The quote may be omitted for numbers and for the symbols T and NIL, which represent the Boolean constants.
- A *lambda expression* is an unnamed function with the same syntax as lambda expressions in Lisp. For example, (LAMBDA (X Y) (+ X Y 4)) represents the function that, given arguments x and y returns $x + y + 4$.

- An *ACL2 term* is either a variable, a constant, or a function application of the form (*fn-name term-list*), where *fn-name* is a symbol corresponding to a function name and *term-list* is a (possibly empty) sequence of terms separated by white space. Thus, `(CAR (FOO 4 X))` is an ACL2 term, which can be written in first-order math notation as $\mathsf{car}(\mathsf{foo}(4, x))$.

- A *subterm* of an ACL2 term *e* is any term contained within *e* including itself. For example, the subterms of `(CAR (FOO 4 X))` are `X`, `4`, `(FOO 4 X)`, and `(CAR (FOO 4 X))`.

- An ACL2 term is *grounded* or a *ground term* if it contains no variable symbols.

- An *ACL2 constant* may also be used more loosely in this dissertation to refer to any grounded ACL2 term containing only the ACL2 core primitive functions, given in Figure 3.1. For example, `(CONS '0 '0)` is referred to as an ACL2 constant. All such constants can be normalized, so it is possible to determine whether any two constants are equal. Technically, this loose definition is necessary even for many innocuous looking constants, such as `'3`, which technically is an abbreviation for

```
(BINARY-+ '1 (BINARY-+ '1 '1)).
```

- `LET*` is used as an abbreviation for nested lambda expressions that bind variables to values. For example,

```
(LET* ((X 4)
       (Y (+ X 8))
       (Z (+ X Y)))
  (* X Y Z))
```

  is the same as

```
((LAMBDA (X)
   ((LAMBDA (Y)
      ((LAMBDA (Z) (* X Y Z))
       (+ X Y)))
    (+ X 8)))
 4)
```

which we write in math notation as

$$
\begin{aligned}
&x * y * z \\
&\quad \text{where} \quad z := x + y \\
&\qquad\qquad\quad\; y := x + 8 \\
&\qquad\qquad\quad\; x := 4.
\end{aligned}
$$

Note that order of the definitions in math notation is the exact opposite of Lisp's `LET*`. In Lisp's `LET*` notation, a variable is used below its definition, whereas in math notation a variable is used above its definition.

3.3 Mixing Math and Lisp Notation

Standard math notation is often used in this dissertation to denote ACL2 functions and terms. Translation of names present some difficulty, since function and variables names are usually more than one character, and often contain symbols, such as "-", which can be confused with operations. To avoid ambiguity, names are sometimes modified. For example, ACL2's `BINARY+` function might be renamed bplus, with a note stating that it implements ACL2's binary addition. Also, since names and variables are often multiple symbols, the $\times$ symbol is never implied. For example, in this dissertation $ab + c$ means add the variable ab to c, not $a \times b + c$.

Actual ACL2 notation is also used in places in this dissertation. For the most part, its use is limited to when functions are defined that input or return ACL2 terms. Such

functions warrant further discussion, since they can easily be the cause of much confusion. Consider, for example, the function $\mathsf{Fn}(E)$, used in Chapter 5. Given an ACL2 term E, denoting a function application, then $\mathsf{Fn}(E)$ is the function being applied. For example,

$$\mathsf{Fn}(\ulcorner\texttt{(CAR (CONS X Y))}\urcorner) = \ulcorner\texttt{CAR}\urcorner.$$

We use the $\ulcorner\urcorner$ delimiters to enclose an ACL2 term, which is a constant in the surrounding mathematical term. Thus, in the above example, the function Fn is applied to the argument $\ulcorner$`(CAR (CONS X Y))`$\urcorner$, which is a constant denoting the ACL2 term `(CAR (CONS X Y))`.

In Lisp, the `QUOTE` (or `'`) symbol has a similar effect as $\ulcorner\urcorner$. For example, given the Lisp definition:

```
(DEFUN FN (X) (CAR X))
```

then

```
(FN '(CAR (CONS X Y))) = 'CAR,
```

whereas

```
(FN (CAR (CONS X Y))) = (FN X) = (CAR X).
```

Similarly, the ACL2 formula

```
(CAR (CONS X Y)) = X
```

is valid, since it is an axiom in ACL2. On the other hand,

$$\ulcorner\texttt{(CAR (CONS X Y))}\urcorner \neq \ulcorner\texttt{X}\urcorner,$$

since $\ulcorner$`(CAR (CONS X Y))`$\urcorner$ is literally a different constant than $\ulcorner$`X`$\urcorner$.

When an ACL2 term occurs by itself, without any enclosing first-order math notation, the $\ulcorner\urcorner$ delimiters may be used or they may be omitted. To emphasize that some

expression is an ACL2 term or ACL2 formula, it is often is enclosed in a box, like the previous ACL2 examples. first-order math notation, on the other hand, is usually presented without an enclosing box.

As another example, consider the function (actually set of functions) $\mathsf{MakeFn}(f, x_0, x_1, ...x_n)$ that, given ACL2 term f and ACL2 terms x_0 through x_n, returns the ACL2 term that applies f to the argument list x_0 through x_n. Thus,

$$\mathsf{Fn}(\mathsf{MakeFn}(f, x_0, x_1, ..., x_n)) = f$$

For example,

$$\mathsf{MakeFn}(\ulcorner\mathtt{CONS}\urcorner, \ulcorner\mathtt{X}\urcorner, \ulcorner\mathtt{Y}\urcorner) = \ulcorner\mathtt{(CONS\ X\ Y)}\urcorner.$$

and

$$\mathsf{MakeFn}(\ulcorner\mathtt{EQUAL}\urcorner, \mathsf{MakeFn}(\ulcorner\mathtt{CAR}\urcorner, \mathsf{MakeFn}(\ulcorner\mathtt{CONS}\urcorner, \ulcorner\mathtt{X}\urcorner, \ulcorner\mathtt{Y}\urcorner)), \ulcorner\mathtt{X}\urcorner).$$

is equal to the ACL2 term

```
(EQUAL (CAR (CONS X Y)) X),
```

which is a valid ACL2 formula (here we implicitly translate the ACL2 term into a formula as described in Section 3.2).

3.4 Primitives

An *ACL2 primitive* is a function symbol present in the ACL2 ground-zero theory, i.e., the theory in which the ACL2 theorem prover begins. We further divide the ACL2 primitives into core primitives and defined primitives. The *core primitives*, or undefined primitives, are the 29 primitives in Table 3.1 [1]. We call the remaining ACL2 primitives *defined primitives*.

Note that while ACL2 is an untyped language, its objects are divided essentially into characters, strings, numbers, symbols, and lists (or trees). ACL2 numbers are the complex

[1] Table 3.1 is copied verbatim from "A Precise Description of the ACL2 Logic" by Kaufmann and Moore [38]

rationals (e.g., $4/11 + 1/2 \times i$), characters are ASCII characters (e.g., 'a'), lists (or trees) are lists ACL2 objects (e.g., $[1, 2, 3]$), strings are essentially lists of characters tagged as a string (e.g., "abc"), and symbols are essentially strings tagged as a symbol (e.g., *abc*). For more information on the constants of ACL2 and how they are constructed, see "A Precise Description of the ACL2 Logic" by Kaufmann and Moore [38].

The core primitives are defined through a series of axioms. For example, the `IF` primitive satisfies the following axioms:

```
X = 'NIL → (IF X Y Z) = Y
X ≠ 'NIL → (IF X Y Z) = Z
```

The defined primitives are each defined using a single Lisp definition, which represents a single axiom in the ground zero theory. Some examples of common primitives and their definitions are shown in Figure 3.3 and described as follows:

- The `IMPLIES` and `IFF` functions implement logical implication and Boolean equivalence respectively.
- The `(NFIX X)` function returns `X` if `X` is a natural number, and zero otherwise.
- The `(ZP X)` function returns true if `X` is zero or is not a natural number.
- The `MIN` and `MAX` functions select the minimum and maximum of their inputs.
- The `NTH` function returns the nth member of a list.
- The `LEN` returns the length of a list.

ACL2 also has a built-in representation for ordinals up to ϵ_0. Since ordinals, such as the natural numbers, cannot decrease forever, they are useful for reasoning about fixpoints. Primitives involving ordinals include `(O-P X)`, which recognizes whether `X` is an ordinal in ACL2's representation and `(O< X Y)`, which determines whether an `X` represents an ordinal less than the ordinal represented by `Y`.

Table 3.1: ACL2 Core Primitives

Function	Arity	Description
BINARY-*	2	multiplies two numbers
BINARY-+	2	adds two numbers
UNARY--	1	negates a number
UNARY-/	1	inverts a number
<	2	less than on the rationals
BOOLEANP	1	recognizes 'T and 'NIL
CAR	1	first element of a list
CDR	1	all but first element of a list
CHAR-CODE	1	maps characters to integers
CHARACTERP	1	recognizes characters
CODE-CHAR	1	maps integers to characters
COMPLEX	2	builds a complex from two rationals
COMPLEX-RATIONALP	1	recognizes a complex number
COERCE	2	maps between character lists and strings
CONS	2	builds a list
CONSP	2	recognizes a non-empty list
DENOMINATOR	1	denominator of a rational
EQUAL	2	equality predicate
IF	3	if-then-else
IMAGPART	1	imaginary part of a complex
INTEGERP	1	recognizes integers
INTERN-IN-PACKAGE-OF-SYMBOL	2	maps a string to a symbol
NUMERATOR	1	numerator of a rational
RATIONALP	1	recognizes rationals
REALPART	1	real part of a complex
STRINGP	1	recognizes strings of characters
SYMBOL-NAME	1	name of a symbol
SYMBOL-PACKAGE-NAME	1	package name of a symbol
SYMBOLP	1	recognizes symbols

```
(DEFUN IMPLIES (P Q) (IF P (IF Q 'T 'NIL) 'T))

(DEFUN IFF (P Q) (IF P (IF Q 'T 'NIL) (IF Q 'NIL 'T)))

(DEFUN NFIX (X)
 (IF (IF (INTEGERP X) (NOT (< X '0)) 'NIL)
     X '0))

(DEFUN ZP (X)
  (IF (INTEGERP X) (NOT (< '0 X)) 'T))

(DEFUN MIN (X Y) (IF (< X Y) X Y))

(DEFUN MAX (X Y) (IF (< Y X) X Y))

(DEFUN NTH (N L)
  (IF (NOT (CONSP L))
      'NIL
      (IF (ZP N)
          (CAR L)
          (NTH (BINARY-+ '-1 N) (CDR L)))))

(DEFUN LEN (X)
  (IF (CONSP X)
      (BINARY-+ '1 (LEN (CDR X)))
     '0))
```

Figure 3.3: Some examples of defined primitives that are used elsewhere in the dissertation.

ACL2 also supports Lisp macros, which are simply abbreviations in the logic. Some primitive macros used later in the dissertation include:

- **Conjunction.** Conjunction, like implication is implemented using the `IF` core primitive. Conjunction, however, is implemented using a macro, `AND`, so that any number of arguments may be conjoined. For example, `(AND X Y)` is an abbreviation for `(IF X Y 'NIL)`, while `(AND X Y Z)` is an abbreviation for `(IF X (IF Y Z 'NIL) 'NIL)`.

- **Disjunction.** Disjunction is analogous to conjunction, and is named `OR`. For example, `(OR X Y Z)` is an abbreviation for `(IF X X (IF Y Y Z))`.

- **Addition.** Addition, named `+`, is another macro that can take multiple arguments. The expression `(+ X Y Z)` is an abbreviation for `(BINARY-+ X (BINARY-+ Y Z))`.

- **Multiplication.** Multiplication, named `*`, is analogous to addition. The expression `(* X Y Z)` is an abbreviation for `(BINARY-* X (BINARY-* Y Z))`.

- **Subtraction.** Subtraction is a macro, named `-`, with two arguments. The expression `(- X Y)` is an abbreviation for `(BINARY-+ X (UNARY-- Y))`.

For a complete list of the axioms in the ground-zero theory and ACL2's standard abbreviations, see "A Precise Description of the ACL2 Logic" by Kaufmann and Moore.

3.5 The Definition Principle

The ACL2 ground-zero theory can be extended using definition events, such as those in Figure 3.2. Each such event adds a new axiom to the ACL2 theory defining the corresponding function symbol. For example, the axiom corresponding to the `IN` definition is:

```
  (IN A X)
=
  (IF (NOT (CONSP X))
     NIL
    (IF (EQUAL A (CAR X))
        T
       (IN A (CDR X))))
```

where `A` and `X` are again implicitly universally quantified.

It is possible to create a syntactically well-formed Lisp definition that leads to an inconsistent theory. For example, if `(F X) = (NOT (F X))` is added to any ACL2 theory, then it becomes possible to prove anything. To avoid such axioms, ACL2 requires that every recursive definition terminate. To do this, for every recursive function f, a *measure function* m_f, with the same formal parameters as f, must be shown to return an ordinal that decreases on each recursive application. In other words, if the definition of $\mathsf{f}(x)$ contains a recursive call $\mathsf{f}(\mathsf{r}(x))$, under condition $\mathsf{p}(x)$, then it must be proven that $\mathsf{p}(x) \rightarrow \mathsf{m}_f(\mathsf{r}(x)) < \mathsf{m}_f(x)$. For example, consider the definition of `IN` again:

```
(DEFUN IN (A X)
  (DECLARE (XARGS :MEASURE (LEN X)))
  (IF (NOT (CONSP X))
      NIL
    (IF (EQUAL A (CAR X))
        T
      (IN A (CDR X)))))
```

Normally, ACL2 chooses a measure function heuristically, and in the case of `IN`, ACL2's heuristics succeed. However, in the above example, the body of the measure function is given explicitly as `(LEN X)` (where `X` corresponds to the formal parameter of `IN` with the same name). Given the measure function corresponding to `IN` and its recursive call, the

proof obligation is:

```
(AND (O-P (LEN X))
     (IMPLIES (AND (CONSP X)
                   (NOT (EQUAL A (CAR X))))
              (O< (LEN (CDR X)) (LEN X))))
```

which states that `(LEN X)` is an ordinal and that it decreases when `IN` is called recursively. The above proof obligation is a theorem that is proven automatically by the ACL2 theorem prover. The proof follows from the fact that `(LEN X)` is a natural number, and all natural numbers are ordinals. Furthermore, if `X` is a list, then `(LEN (CDR X))` is smaller than `(LEN X)` by induction on `X` and the definition of `LEN`.

ACL2 also supports the introduction of mutual-recursive functions, but no knowledge of mutual-recursive functions is needed in this dissertation.

3.5.1 Encapsulation

ACL2 also supports *constrained functions*, which are functions that satisfy a list of properties, rather than a single axiom corresponding to a definition event. To avoid inconsistent axioms, such as constraining `f` to be a function satisfying

$$\mathsf{f}(x) = \neg \mathsf{f}(x),$$

it is required to provide a *witness*—a function defined using the definition principle that satisfies the property. For example, a function `EQUIV` could be introduced satisfying only the constraints in Figure 3.4. The `EQUIV` constraints are the constraints required to be an equivalence relation, and are satisfied by the previously defined `PERM` function, as well as the ACL2 primitives `EQUAL`, `IFF`, and the function that always returns **true**, `(LAMBDA (X Y) 'T)`.

A constrained function may also be introduced with an empty set of constraints, in which case we call it an *uninterpreted function*. Uninterpreted functions only have the

```
(IMPLIES (AND (EQUIV X Y) (EQUIV Y Z)) (EQUIV X Z))

(IMPLIES (EQUIV X Y) (EQUIV Y X))

(EQUIV X X)
```

Figure 3.4: The list of constraints for an arbitrary equivalence relation, `EQUIV`.

implicit constraint that they return the same value given the same arguments. Note that the proof obligation for an uninterpreted function is trivial.

Constrained functions are implemented using ACL2's *encapsulation* feature, which allows definitional axioms to occur only within an encapsulated block of events. Thus the witness function need not introduce any new function symbols into the ACL2 theory produced after the encapsulation block. The encapsulation block also may be used to manage the complexity of large proofs by introducing function symbols and theorems locally to the block and then exporting only the key theorems and definitions needed for some larger proof. The removal of such function symbols is justified in a paper by Kaufmann and Moore [39], which shows that every definitional axiom in ACL2 is a conservative extension of the previous theory, i.e., every theorem provable in the extended theory is either provable in the previous theory or involves the function symbols introduced by the extension.

3.6 The Theorem Prover

The ACL2 theorem prover combines many proof techniques, which are, for the most part, applied automatically using heuristics. This section presents a brief overview of the techniques, beginning with forward chaining and rewriting. For a more complete overview, see *Computer-Aided Reasoning: An Approach*.

3.6.1 Forward Chaining

At any point during a proof, ACL2 has a set of valid assumptions, known as the *context*. For example, ACL2 will attempt to prove $\mathsf{f}(x) \wedge \mathsf{g}(x) \rightarrow \mathsf{h}(x)$ by proving $\mathsf{h}(x)$ within a context where $\mathsf{f}(x)$ and $\mathsf{g}(x)$ are assumed. If that fails, it will attempt to prove $\neg\mathsf{f}(x)$, within a context where $\mathsf{g}(x)$ and $\neg\mathsf{h}(x)$ are assumed (since $\mathsf{f}(x) \wedge \mathsf{g}(x) \rightarrow \mathsf{h}(x)$ is equivalent to $\mathsf{g}(x) \wedge \neg\mathsf{h}(x) \rightarrow \neg\mathsf{f}(x)$). If that fails, it will similarly attempt to prove $\neg\mathsf{g}(x)$, within a context where $\mathsf{f}(x)$ and $\neg\mathsf{h}(x)$ are assumed.

ACL2 also keeps a database of previously proven theorems. The forward chaining technique attempts to use theorems in the database marked as *forward chaining rules* to grow the context. For example, if $\mathsf{f}(x)$ and $\mathsf{g}(x)$ are in the current context, and $\mathsf{f}(x) \rightarrow m(x)$ is a forward chaining rule, then $m(x)$ is added to the context. If $m(x)$ is the proof goal, then forward chaining has succeeded. Otherwise, it still may be useful to know $m(x)$ while attempting to prove the current goal by additional forward chaining, or other techniques.

Since any instantiation of a theorem is also a theorem, the forward chaining technique attempts to find useful instances of forward chaining rules. The primary technique involves unification. We say a term A *unifies* with a term B, if there exists a substitution σ mapping the variables in A to expressions such that $A/\sigma = B$. Therefore, if A is a universally quantified theorem and A unifies with some term B, then B is also a theorem. In the automated theorem proving community, this is sometimes referred to as one-way unification, but within this dissertation all unification is one-way, so we refer to it simply as unification.

Each forward chaining rule has an associated *trigger term*. Before a forward chaining rule is applied the trigger term must unify with some assumption in the context. Then the forward chaining technique attempts to extend the substitution implied by the unification to create an instance of the rule where all hypotheses occur in the context. For example, if $\mathsf{f}(\mathsf{g}(a))$ is in the current context and $\mathsf{f}(x) \rightarrow \mathsf{h}(x)$ is a forward chaining rule with trigger term $\mathsf{f}(x)$, then $\mathsf{h}(\mathsf{g}(a))$ is added to the context, since $\mathsf{f}(x)$ unifies with $\mathsf{f}(\mathsf{g}(a))$ under the

substitution $[x \mapsto \mathsf{g}(a)]$. If a user marks a theorem as a forward chaining rule without specifying a trigger term, then the trigger term is the first hypothesis of the theorem.

Unification on the trigger term may fail to produce a complete substitution mapping the variables in the forward chaining rule to values. In this case, which is discussed in Chapter 4, other heuristics are used to complete the substitution.

3.6.2 Rewriting

The rewriting proof technique attempts to use previously proven theorems, marked as *rewrite rules*, to simplify the current proof goal. For example, if marked as a rewrite rule, a theorem of the form:

My-Rewrite:

$$\mathsf{p}(x) \wedge \mathsf{q}(x) \rightarrow \mathsf{f}(x) = \mathsf{g}(x)$$

is treated as an instruction to rewrite instances of $\mathsf{f}(x)$ to $\mathsf{g}(x)$, when $\mathsf{p}(x)$ and $\mathsf{q}(x)$ are true. If $\mathsf{f}(x)$ unifies with a term in the current proof goal under a substitution σ, then $\mathsf{f}(x)/\sigma$ is replaced by $\mathsf{g}(x)/\sigma$ if $\mathsf{p}(x)/\sigma$ and $\mathsf{q}(x)/\sigma$ can be proven from the current context. For example, using *My-Rewrite*, ACL2 will rewrite $\mathsf{h}(\mathsf{f}(\mathsf{g}(a)))$ to $\mathsf{h}(\mathsf{g}(\mathsf{g}(a)))$ if $\mathsf{p}(\mathsf{g}(a))$ and $\mathsf{q}(\mathsf{g}(a))$ can be proven from the current context.

Rewriting in ACL2 occurs inside out. Thus, using *My-Rewrite* and assuming the hypotheses can be proven, a proof goal $\mathsf{f}(\mathsf{f}(a))$ will first be rewritten to $\mathsf{f}(\mathsf{g}(a))$ and then to $\mathsf{g}(\mathsf{g}(a))$.

Rewrite rules need not conclude with an equality; they can also conclude with any *equivalence relation*, which is a binary function satisfying the theorems in Figure 3.4. For example, if $\mathsf{equiv}(x, y)$ is such an equivalence relation, then a rewrite rule can have the conclusion $\mathsf{equiv}(\mathsf{f}(x), \mathsf{g}(x))$. Such a rule is treated as an instruction to rewrite $\mathsf{f}(x)$ to $\mathsf{g}(x)$ when $\mathsf{equiv}(\mathsf{f}(x), \mathsf{g}(x))$ is sufficient to ensure that the original proof goal is true if and only if the rewritten proof goal is true. For example, if the proof goal is $\mathsf{h}(\mathsf{f}(a))$, then $\mathsf{h}(\mathsf{f}(a))$ can be rewritten to $\mathsf{h}(\mathsf{g}(a))$ if $\mathsf{equiv}(x, y) \rightarrow (\mathsf{h}(x) \leftrightarrow \mathsf{h}(y))$.

ACL2 relies on theorems marked as *congruence rules* to justify rewriting based on equivalence relations other than equality. Congruence rules have the form

$$\mathsf{equiv}_0(x_i, y_i) \rightarrow \mathsf{equiv}_1(\mathsf{f}(x_1, ..., x_n), \mathsf{f}(x_1, ..., x_{i-1}, y_i, x_{i+1}, ..., x_n))$$

where equiv_0 and equiv_1 are arbitrary equivalence relations; x_1 through x_n and y are variables; and f is an arbitrary function. Since "if and only if" is an equivalence relation $\mathsf{equiv}(x, y) \rightarrow (\mathsf{h}(x) \leftrightarrow \mathsf{h}(y))$ is an example of a theorem that can be marked as a congruence rule.

By combining congruence rules, ACL2 can justify equivalence-based rewriting of complex expressions. For example, if $\mathsf{equiv}(\mathsf{f}(x), \mathsf{g}(x))$, then the rewriting of $\mathsf{h}(\mathsf{f}(\mathsf{f}(a)))$ to $\mathsf{h}(\mathsf{g}(\mathsf{g}(a)))$ is justified by the congruence rules $\mathsf{equiv}(x, y) \rightarrow \mathsf{equiv}(\mathsf{f}(x), \mathsf{f}(y))$ and $\mathsf{equiv}(x, y) \rightarrow (\mathsf{h}(x) \leftrightarrow \mathsf{h}(y))$.

By default, all theorems entered into ACL2's theorem database are marked as rewrite rules. Theorems with a conclusion Q that is not an equivalence relation can be considered a rewrite rules with the conclusion $Q \leftrightarrow \mathbf{true}$. For example, $\mathsf{p}(x) \rightarrow \mathsf{q}(x)$ is treated as $\mathsf{p}(x) \rightarrow (\mathsf{q}(x) \leftrightarrow \mathbf{true})$. This leads to a form of backchaining, since ACL2 will attempt to prove $\mathsf{q}(a)$ by rewriting $\mathsf{q}(a)$ to **true** if $\mathsf{p}(a)$ can be proven. This backchaining with the rewriting proof technique is also how the theorem prover is able to use lemmas automatically in the example proofs of Section 3.1.

3.6.3 Overview of other Main Techniques

Some other techniques used within the ACL2 theorem prover are as follows:

- **Evaluation.** All of the core primitives in Table 3.1 are *executable*, meaning that their value can be determined given any constant inputs. Similarly, any function whose body contains only executable functions (or itself) is also executable. Thus, many *ground terms*, i.e., terms containing no variables—can be evaluated to constants. For example, $\mathsf{car}(\mathsf{cons}(4 + 5, 7))$ is automatically reduced to 9. The evaluation proof

technique is used to reduce, when possible, all function calls with constant arguments to constants.

- **Expansion.** Every application of a function introduced by a definition event can be expanded into an instance of its body. Performing such expansions is not always a good idea though, since recursive functions can be expanded infinitely. The expansion proof technique performs such expansions only when ACL2's heuristics determine that such an expansion is likely to lead to a successful proof.

- **Linear Arithmetic.** ACL2 contains a proof technique for proving linear arithmetic theorems. While the mechanism does not prove all theorems within the traditional theory of linear arithmetic, it has the advantage that it can be extended with theorems involving user-created functions.

- **Type Prescription.** ACL2 also contains a mechanism specially designed to prove rules involving the basic types of ACL2, such as lists and Booleans. For example, when a new predicate symbol $\mathsf{p}(x)$ is introduced, ACL2 will likely automatically deduce $\mathsf{booleanp}(\mathsf{p}(x))$, where $\mathsf{booleanp}(x)$ is the function recognizing Booleans. Then, if the hypothesis of some forward chaining rule or rewrite rule requires $\mathsf{booleanp}(\mathsf{p}(x))$, that hypothesis is proven without needing to look at the body of $\mathsf{p}(x)$.

- **Case splitting.** Case splitting is sometimes used to reduce an `IF` into two cases corresponding to its true and false branches.

- **Destructor Elimination.** The destructor elimination proof technique removes calls of the destructors `CAR` and `CDR`.

- **Induction.** ACL2 contains heuristics that automatically attempt to prove theorems by induction. The ACL2 heuristics attempt to find a good induction scheme to prove

a given formula from the proofs of termination of functions occurring in that formula. For example,

$$\begin{array}{l} \textit{in-minus-a-implies-in}: \\ \quad \mathsf{in}\,(a, \mathsf{del}\,(b, x)) \rightarrow \mathsf{in}\,(a, x), \end{array}$$

from Section 3.1, is proven automatically by induction on the size of x; which is also the measure used to prove the termination of del.

- **Generalization.** Often induction only succeeds when given a more general form of a property. ACL2's generalization proof technique attempts to find such generalizations and perform them prior to induction.
- **Functional instantiation.** Any theorem proven about a constrained function is also valid for any function satisfying its constraints. For example, any theorem proven about the `EQUIV` function with the constraints in Figure 3.4, can be automatically deduced about `IFF`, `EQUAL`, and `PERM`. Functional instantiation is not tried automatically by the theorem prover, but is accessible through ACL2's hint mechanism described in Section 3.6.4.

The above techniques are combined with a few techniques not mentioned, to create the ACL2 theorem prover. The most straightforward techniques are generally tried first, such as evaluation and rewriting, and those requiring the most heuristics, such as induction or generalization, are tried last.

3.6.4 User Guidance

Expert users can control the theorem prover through multiple mechanisms. Most of the techniques described in Section 3.6.3 can be completely disabled when they are interfering with other techniques. More commonly though, users guide the theorem prover by proving lemmas, as described in Section 3.1, and marking those lemmas as rules to be associated

with various ACL2 proof techniques. Rules are also commonly disabled or enabled based on whether they are likely to be useful in a given proof.

Users can affect a proof more directly by using ACL2's hint mechanism. ACL2 typically divides a proof into many user-accessible goals and the hint mechanism allows users to provide assistance at a particular goal. The assistance may involve, among other things, the disabling and enabling of particular rules; the use of a particular strategy, such as expanding a certain function application; or instantiating a particular theorem. In Chapter 11, a new form of assistance, accessible through the hint mechanism, is described.

3.6.5 Meta Reasoning

Meta reasoning is a proof technique that allows the use of ACL2 functions that directly manipulate ACL2 terms. This section provides an introduction to meta reasoning, which is needed for the discussion of a new technique similar to meta reasoning in chapter 11. A more complete discussion of meta reasoning can be found in Hunt et al. [30].

A simple, but powerful, form of meta reasoning is available through the $\mathsf{syntaxp}\,(x)$ primitive. Within the ACL2 logic, $\mathsf{syntaxp}\,(x) = \mathbf{true}$, so $\mathsf{syntaxp}\,(x) \rightarrow \mathsf{p}\,(x)$ reduces to proving $\mathsf{p}\,(x)$. "Under the hood", however, $\mathsf{syntaxp}\,(x)$ can prevent proof techniques from being used automatically. A theorem $\mathsf{syntaxp}\,(x) \rightarrow \mathsf{p}\,(x)$ will only be applied automatically when x/σ evaluates to true, where σ is a substitution mapping the variables in the current theorem to constants corresponding to their value after unification. For example, the rewrite rule

$$\mathsf{syntaxp}\,(\mathsf{p}\,(x)) \rightarrow \mathsf{f}\,(x) = \mathsf{g}\,(x)$$

will rewrite $\mathsf{f}\,(\mathsf{h}\,(a))$ to $\mathsf{g}\,(\mathsf{h}\,(a))$ only when $\mathsf{p}\,(\ulcorner\texttt{(H A)}\urcorner)$ is true, where $\ulcorner\texttt{(H A)}\urcorner$ is the ACL2 term representing $\mathsf{h}\,(a)$.

More sophisticated forms of meta reasoning require that an ACL2 *evaluator* be defined. Evaluators are constrained functions, with constraints that are satisfied by the actual evaluation of ACL2 terms. An evaluator $\mathsf{ev}\,(e, \sigma)$ inputs an ACL2 term e and a

substitution σ and is either equal to e/σ, as defined by the current ACL2 theory, or is undefined over the given term e. For example, if the definitional axiom for some function is $\mathsf{f}(x) = x + 1$, then

$$\mathsf{ev}(\ulcorner\texttt{(F X)}\urcorner, \sigma) = \mathsf{ev}(\ulcorner\texttt{(+ X 1)}\urcorner, \sigma)$$

may be one of the constraints defining $\mathsf{ev}(e, \sigma)$. Thus, it is possible to prove theorems about the evaluation of $\mathsf{f}(x)$, such as

$$\mathsf{ev}(\ulcorner\texttt{F 10}\urcorner, \sigma) = 11,$$

which states that $\mathsf{f}(10)$ evaluates to 11. Another example is

$$\mathsf{ev}(\ulcorner\texttt{(F (+ X 1))}\urcorner, \sigma) = \mathsf{ev}(\ulcorner\texttt{(+ (F X) 1)}\urcorner, \sigma),$$

which states that $\mathsf{f}(x + 1)$ evaluates to the same value as $\mathsf{f}(x) + 1$.

Once an evaluator $\mathsf{ev}(e, \sigma)$ is defined, a *meta rule* can be associated with theorems such as:

$$\begin{aligned} &\mathsf{termp}(x) \wedge \mathsf{alistp}(\sigma) \wedge \mathsf{ev}(\mathsf{hyp}(x), \sigma) \\ &\rightarrow \\ &\mathsf{equiv}(\mathsf{ev}(x, \sigma), \mathsf{ev}(\mathsf{trans}(x), \sigma)), \end{aligned}$$

where $\mathsf{termp}(x)$ recognizes a well-formed ACL2 term x, $\mathsf{alistp}(\sigma)$ recognizes a well formed substitution σ, $\mathsf{equiv}(x, y)$ is any equivalence relation, $\mathsf{hyp}(x)$ is any predicate and $\mathsf{trans}(x)$ is any function.

A trigger term, provided by the user, is also associated with the meta rule. The meta rule is then applied as part of the rewriting proof technique described in Section 3.6.2. Whenever the trigger term unifies with a term x in the current proof goal, then x is rewritten by applying the trans function on it if the hyp predicate returns true, and equivalence-based rewriting using $\mathsf{equiv}(x, y)$ is justified as described in Section 3.6.2. For example, if $\mathsf{hyp}(\ulcorner\texttt{(F (G A))}\urcorner) = \textbf{true}$ and $\mathsf{trans}(\ulcorner\texttt{(F (G A))}\urcorner) = \ulcorner\texttt{(G (F A))}\urcorner$, then the proof goal $\mathsf{h}(\mathsf{f}(\mathsf{g}(a)))$ is rewritten by meta-reasoning to $\mathsf{h}(\mathsf{g}(\mathsf{f}(a)))$, assuming that the equivalence-based rewriting is justified.

Meta rules therefore are a more direct way to manipulate terms within the ACL2 theorem prover. Meta rules can, at times, be more efficient than traditional rewriting. They are also capable of encoding more complex heuristics.

Chapter 4

Increasing Free Variable Instantiation During Forward Chaining

4.1 Introduction

One way to make large-scale formal verification more practical is to increase the automation in a tool already capable of large-scale formal verification, such as ACL2. This chapter describes such a modification to ACL2's forward chaining proof technique. The modification essentially requires more time to prove theorems, but can prove more theorems automatically. The modification became part of ACL2's default forward chaining technique in version 2.7, and has been the default since then.

Section 4.2 presents an example of a theorem that used to require user interaction to prove, but now can be proven automatically. Section 4.3 describes the modification in detail. Finally, Section 4.4 analyzes the performance effects of the modification on the theorem prover.

4.2 Example

Consider the predicate $\mathsf{len\text{-}equal}(x,y)$, which returns whether two lists have the same length. In ACL2, $\mathsf{len\text{-}equal}(x,y)$ can be defined as follows:

$$\mathsf{len\text{-}equal}(x,y) \triangleq \begin{cases} \mathsf{atom}(x) \land \mathsf{atom}(y), & \text{if } \mathsf{atom}(x) \lor \mathsf{atom}(y) \\ \mathsf{len\text{-}equal}(\mathsf{cdr}(x), \mathsf{cdr}(y)), & \text{otherwise.} \end{cases}$$

Furthermore, let $\mathsf{p}(x,y)$ be a constrained function satisfying the following axiom:

len-equal-implies-p:

$$\mathsf{len\text{-}equal}(x,y) \rightarrow \mathsf{p}(x,y)$$

Given the definition of $\mathsf{len\text{-}equal}(x,y)$, the ACL2 theorem prover can prove the following theorem automatically by induction on x:

len-equal-transitive:

$$\mathsf{len\text{-}equal}(x,y) \land \mathsf{len\text{-}equal}(y,z) \rightarrow \mathsf{len\text{-}equal}(x,z)$$

If *len-equal-transitive* and *len-equal-implies-p* are marked as forward chaining rules, then the following theorem can be proven automatically:

len-equal-implies-p-2:

$$\mathsf{len\text{-}equal}(a,b) \land \mathsf{len\text{-}equal}(b,c) \rightarrow \mathsf{p}(a,c)$$

During the above proof attempt, the ACL2 theorem prover keeps a list of facts, called the *context*, that follow from the hypotheses of the theorem being proven (as well as some other sources). Among the context are the hypotheses themselves, $\mathsf{len\text{-}equal}(a,b)$ and $\mathsf{len\text{-}equal}(b,c)$. The forward chaining technique then adds $\mathsf{len\text{-}equal}(a,c)$ to the context, since it follows from the facts in the context and *len-equal-transitive* under the substitution $[x \mapsto a, y \mapsto b, z \mapsto c]$. The forward chaining technique next proves the conclusion, since $\mathsf{p}(a,c)$ follows from $\mathsf{len\text{-}equal}(a,c)$ and *len-equal-implies-p* under the substitution $[x \mapsto a, y \mapsto c]$.

Now consider the following two theorems, which are the same as *len-equal-implies-p-2*, but each contain an extra (unnecessary) hypothesis.

len-equal-implies-p-3:

$$\text{len-equal}\,(a,b) \wedge \text{len-equal}\,(b,c) \wedge \text{len-equal}\,(b,d) \rightarrow \text{p}\,(a,c)$$

len-equal-implies-p-4:

$$\text{len-equal}\,(a,b) \wedge \text{len-equal}\,(b,d) \wedge \text{len-equal}\,(b,c) \rightarrow \text{p}\,(a,c)$$

The theorem *len-equal-implies-p-4* is proven automatically by the forward chaining technique in ACL2 version 2.6, but the theorem *len-equal-implies-p-3* is not.

To explain the problem, first we need to explain a little about how the forward chaining proof technique works. The forward chaining proof technique relies on unification with a single term, known as the *trigger term*, to find an instance of a forward chaining rule that is applicable to a given proof. The unification provides an instance of variables to values for all variables occurring in the trigger term. Another method must be used to instantiate variables occurring outside the trigger term, which are called *free variables*. By default, the trigger term is the first hypotheses of a forward chaining rule. Thus, $len-equal-transitive$ trigger term is the $\text{len-equal}\,(x,y)$. The variable z in $len-equal-transitive$ is a free variable.

In ACL2 version 2.6, only the first instance found for a free variable is used. Thus, when attempting to prove *len-equal-implies-p-3* and *len-equal-implies-p-4*, the forward chaining technique uses *len-equal-transitivity* either under the substitution $[x \mapsto a, y \mapsto b, z \mapsto c]$ or under the substitution $[x \mapsto a, y \mapsto b, z \mapsto d]$, but not both. The chosen substitution depends on the order of the hypotheses in the conjecture being proven. In theorem *len-equal-implies-p-3*, $[x \mapsto a, y \mapsto b, z \mapsto d]$ is chosen, and $\text{len-equal}\,(a,d)$ is added to the context. In theorem *len-equal-implies-p-4*, on the other hand, $[x \mapsto a, y \mapsto b, z \mapsto c]$ is chosen, and $\text{len-equal}\,(a,c)$ is added to the context. Therefore, *len-equal-implies-p-4* is proven by forward chaining, but *len-equal-implies-p-3* is not.

The modification described in this chapter is such that all possible instances of free variables are added to the context. Thus, both len-equal (a, c) and len-equal (a, d) are added to the context during the proof attempts of *len-equal-implies-p-3* and *len-equal-implies-p-4*, and both theorems are proven automatically.

4.3 Forward Chaining Implementation

This section describes the modification to the forward chaining proof technique in detail. We refer to the unmodified forward chaining technique as the *match once* approach and the modified forward chaining technique as the *match all* approach.

First, given two substitutions σ and σ', we say that σ' *extends* σ, or $\sigma \sqsubseteq \sigma'$, if σ' contains every element in σ. For example,

$$[a \mapsto 1, b \mapsto 2] \sqsubseteq [a \mapsto 1, b \mapsto 2, c \mapsto 3] = \mathbf{true}$$

whereas

$$[a \mapsto 1, b \mapsto 2] \sqsubseteq [a \mapsto 1, c \mapsto 3] = \mathbf{false}.$$

The match once approach begins with a substitution σ_0 resulting from unification with the forward chaining rule's trigger term. Then, given hypothesis H_i, σ_i is defined, if possible, as a substitution satisfying $\sigma_{i-1} \sqsubseteq \sigma_i \wedge H_i/\sigma_i = C$ for some term C in the current context. If, for any hypothesis, no such σ_i exists, then the forward chaining technique fails to use that rule. Otherwise, Q/σ_n is added to the context, where n is the number of hypotheses in the forward chaining rule and Q is its conclusion. The addition of Q/σ_n is justified since each hypothesis of the forward chaining rule under the substitution σ_n is in the context. We illustrate the match once approach, alongside the match all approach, on an example in Figure 4.1.

The match all approach begins with a list of substitutions $\Sigma_0 = [\sigma_0]$, where σ_0 is again the substitution resulting from unification with the trigger term. Then, given hypothesis H_i, a list of substitutions Σ_i is constructed from Σ_{i-1}. A substitution Σ_i contains a

Rule: $\mathsf{f}(w,x) \wedge \mathsf{f}(x,y) \wedge \mathsf{f}(y,z) \rightarrow \mathsf{f}(w,z)$

Initial Context: $\mathsf{f}(a,b)$, $\mathsf{f}(b,c)$, $\mathsf{f}(b,d)$, $\mathsf{f}(c,e)$, $\mathsf{f}(c,f)$, $\mathsf{f}(d,g)$
Initial Substitution: $\sigma_0 = [w \mapsto a, x \mapsto b]$

Match Once	
Hypothesis	σ
$\mathsf{f}(w,x)$:	$[w \mapsto a, x \mapsto b]$
$\mathsf{f}(x,y)$:	$[w \mapsto a, x \mapsto b, y \mapsto c]$
$\mathsf{f}(y,z)$:	$[w \mapsto a, x \mapsto b, y \mapsto c, z \mapsto e]$
Conclusion:	$\mathsf{f}(a,e)$

Match All	
Hypothesis	Σ
$\mathsf{f}(w,x)$:	$[[w \mapsto a, x \mapsto b]]$
$\mathsf{f}(x,y)$:	$[[w \mapsto a, x \mapsto b, y \mapsto c], [w \mapsto a, x \mapsto b, y \mapsto d]]$
$\mathsf{f}(y,z)$:	$[[w \mapsto a, x \mapsto b, y \mapsto c, z \mapsto e], [w \mapsto a, x \mapsto b, y \mapsto c, z \mapsto f],$ $[w \mapsto a, x \mapsto b, y \mapsto d, z \mapsto g]]$
Conclusions:	$\mathsf{f}(a,e)$, $\mathsf{f}(a,f)$, $\mathsf{f}(a,g)$

Figure 4.1: The above figure illustrates the match once and match all approaches on an example. The example rule is a weak form of transitivity for some predicate $\mathsf{f}(x,y)$. It has two free variables, y and z. An initial substitution and context are given, which are the same for both approaches. In the match once approach, a substitution σ is then developed from each hypothesis, leading to a single conclusion to add to the context. In the match all approach, a list of substitutions Σ is developed from each hypothesis, leading to multiple conclusions to add to the context.

substitution σ_i if and only if there exists a $\sigma_{i-1} \in \Sigma_{i-1}$ and a term C in the current context such that $\sigma_{i-1} \sqsubseteq \sigma_i \wedge H_i/\sigma_i = C$. The match all approach then grows the context by adding Q/σ_n for each σ_n in Σ_n, where n is the number of hypotheses in the forward chaining rule and Q is its conclusion. The addition of Q/σ is justified by the validity of the rule, since σ maps each hypothesis to an element in the context. The match all approach is illustrated alongside the match once approach with an example in Figure 4.1.

The number of additions to the context can be exponential in the number of free variables. For instance, x^n conclusions can be drawn from a forward chaining rule of the form:

$$\mathsf{f}(x_1) \wedge \mathsf{f}(x_2) \wedge ... \wedge \mathsf{f}(x_n) \rightarrow \mathsf{g}(x_1, x_2, ..., x_n)$$

when the initial context $[f(a), f(b)]$. If n is 2, for example, $\mathsf{g}(a, a)$, $\mathsf{g}(a, b)$, $\mathsf{g}(b, a)$, and $\mathsf{g}(b, b)$, are derived from $[f(a), f(b)]$ and the rule

$$\mathsf{f}(x_1) \wedge \mathsf{f}(x_2) \rightarrow \mathsf{g}(x_1, x_2).$$

To avoid this exponential and other performance issues caused by the match all technique, forward chaining rules can be tagged as match once, in which case the original match once approach is still used to instantiate them. In practice, however, the match all approach rarely leads to a large increase in proof time, because of the low number of free variables in most rules. Since version 2.7, the ACL2 theorem prover uses the match all technique by default.

4.4 Results

The match all technique has now been part of the prover for over four years. Figure 4.2 compares the time taken to verify ACL2 version 3.1's regression suite using a prover with only the match once forward chaining technique versus a prover with only the match all forward chaining technique. Four files contained theorems that were not provable with the match once prover, whereas every file could be proven with the match all prover. While

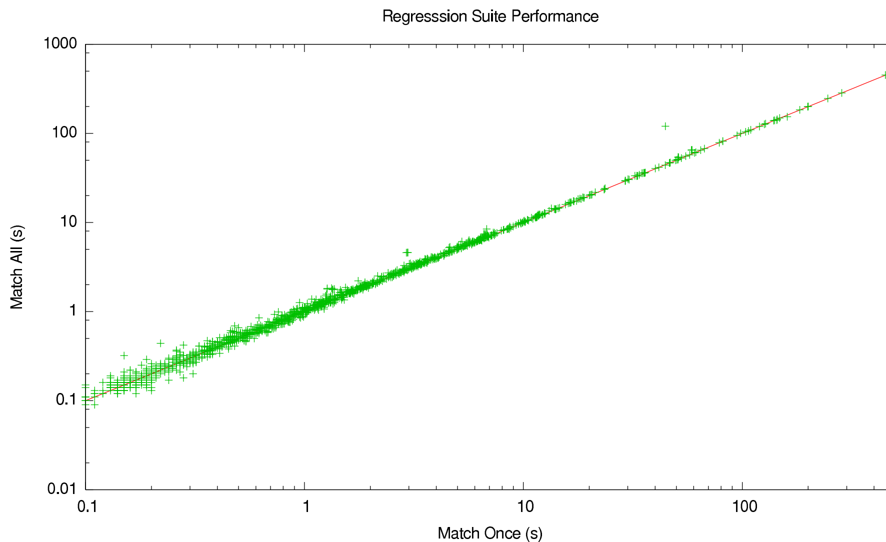


Figure 4.2: A graph comparing the performance of the theorem prover before and after increasing free variable instantiation of forward chaining rules. Each point represents a file in the ACL2 regression suite. The x-axis is the time required when using the old, match once, approach. The y-axis is the time required when using the new, match all, approach. A point is on the line if the same time is required by both techniques.

the match all technique can cause an exponential slow-down, Figure 4.2 shows that any slow-down at all is rare. The total time to prove all the files in the regression suite except the four files that cannot be proven by the match once approach, was 2 hours, 16 minutes, and 7 seconds for the match once prover, and 2 hours, 19 minutes, and 2 seconds for the match all prover—a difference of 2.1 percent [1].

In ACL2 version 2.6, the rewriting and linear arithmetic proof techniques also used an approach analogous to the match once approach in the forward chaining proof technique. However, an approach analogous to the match all approach is now the default for all three techniques. Figure 4.3 extends the comparison to the analogous rewriting and linear arithmetic proof techniques. Here, the match once prover uses a match once free variable approach for rewriting, linear arithmetic, and forward chaining, whereas the match all prover

[1]These results were obtained on a Pentium® 4, 3.0 GHz, dual processor with 2 gigabytes of random access memory. ACL2 version 3.1 was used, running under GNU Common Lisp version 2.6.7.

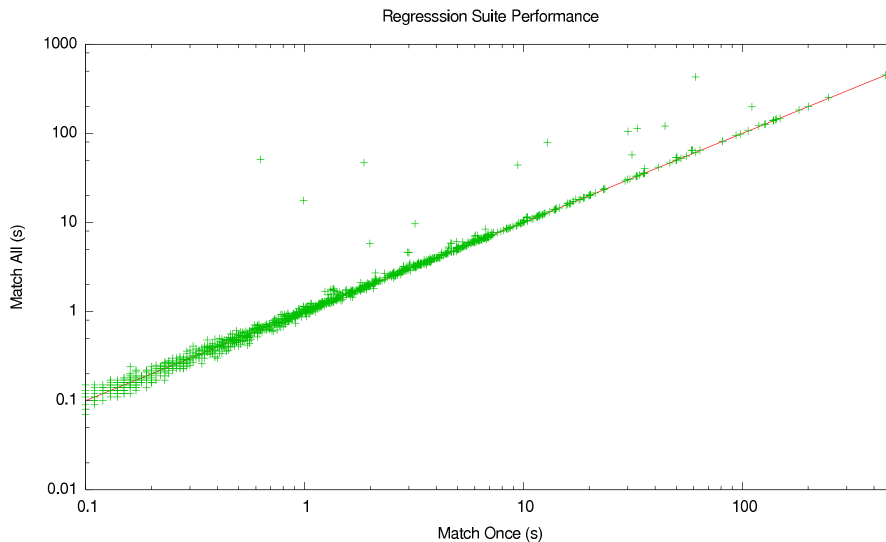


Figure 4.3: A graph comparing the performance of the theorem prover before and after increasing free variable instantiation in the rewriting, linear arithmetic, and forward chaining proof technique. Each point represents a file in the ACL2 regression suite. The x-axis is the time required when using the old, match once, approach. The y-axis is the time required when using the new, match all, approach. A point is on the line if the same time is required by both techniques.

uses a match all free variable approach for all of these techniques. One file contained proofs that could not be completed by the match all prover, due to the slow-down in performance. Twenty-seven files contained proofs that could not be completed by the match once prover. Omitting these twenty-eight files, the total time to prove the regression suite for the match once prover was 1 hour, 53 minutes, 33 seconds versus 2 hours, 10 minutes, and 48 seconds for the match all prover—a difference of 15.2 percent.

4.5 Summary

ACL2's forward chaining proof technique has been modified to allow it to prove more theorem automatically. Furthermore, the modification does not significantly reduce performance on any of the existing proofs in ACL2's regression suite. Based on these results, the

modified technique is now ACL2's default forward chaining technique. Similar improvements to the rewriting and linear arithmetic proof techniques were later implemented and added to the theorem prover as well.

The modification to the forward chaining technique highlights the potential for future improvements to the theorem prover. A general purpose theorem prover must balance a trade-off between proving difficult theorems automatically and proving easy theorems quickly. The modification to the forward chaining technique adjusts this trade-off by proving more theorems automatically and the expense of requiring more time to prove other theorems.

4.6 Development and Bibliographic Notes

This chapter describes joint work with J Moore and Matt Kaufmann. J Moore helped develop the new forward chaining algorithm, I implemented the initial forward chaining modification, and Matt Kaufmann modified it somewhat before adding it to the theorem prover. Matt Kaufmann went on to implement similar modification to the rewriting and linear arithmetic proof techniques.

Automated instantiation of free variables is not always built into general-purpose theorem provers. PVS [62], for example, has no built-in technique for automating free variable instantiation, though such a technique may be implemented through user-defined "tactics." A contextual rewriting technique that instantiates free variables, called here *existential variables*, is part of the Terminating Functional Programs (TFL) environment [83], implemented for HOL [20] and Isabelle [59]. Techniques for using unification to match free variables, called here *holes*, can also be found in the field of logic programming [45, 51].

Chapter 5

The Subclass of Unrollable List Formulas in ACL2 (SULFA)

5.1 Introduction

The previous chapter describes a method to decrease the amount of user guidance required in theorem proving by modifying the ACL2 theorem prover's existing proof techniques. Another way to decrease the amount of user guidance is to identify ACL2 formulas that can be verified fully automatically and then use fully-automated procedures to prove or disprove such formulas. Such a technique not only can significantly reduce the amount of human effort required to prove valid formulas, but also can inform users when a conjecture is not provable from the current ACL2 theory.

This chapter defines the decidable Subclass of Unrollable List Formulas in ACL2 (SULFA). Here we say that SULFA is *a decidable subclass* of ACL2 because:

1. A terminating procedure exists that recognizes whether any ACL2 formula is in SULFA.
2. A terminating procedure exists that decides whether any formula in SULFA is prov-

able in ACL2.

This chapter addresses the first requirement, by showing that there exists a terminating procedure that recognizes whether any ACL2 formula is in SULFA. Chapter 6 addresses the second requirement, by presenting a terminating procedure that decides any SULFA formula.

Note that "decidable subclass" is a pretty weak term. Any terminating proof procedure is a decision procedure for a decidable subclass, namely the subclass of formulas that it proves. The SULFA subclass is a meaningful subclass in that it is significantly easier to recognize than it is to solve. We have successfully applied the SULFA recognizer to all the theorems in ACL2's regression suite. On the other hand, solving SULFA formulas is an NP-hard problem.

Furthermore, SULFA is distinguished from most other meaningful decidable subclasses in that it is not restricted to a finite set of functions, but, like ACL2, can be extended by a function definitional principle. Furthermore, SULFA is defined entirely using the primitives of ACL2, along with a definitional principle to support additional functions. Thus sometimes ACL2 formulas are in SULFA even when created by users unaware of it, including 3.2% of the formulas in ACL2's regression suite.

This chapter begins by describing the intuition behind SULFA and why it is decidable in Section 5. Section 5.3 then explains some simplifications we make to the ACL2 logic. Section 5.4 rigorously defines SULFA and Section 5.5 shows how to create an efficient recognizer for SULFA. Section 5.6 presents the results of applying the efficient SULFA recognizer to ACL2's regression suite of theorems.

5.2 Intuition Behind SULFA

SULFA is based primarily on the theory of list structures, which, as defined by Nelson and Oppen [58], corresponds to ACL2 formulas formed by terms in the following grammar:

E ::= `(CAR` E`)` | `(CDR` E`)` | `(CONS` E E`)` | `(CONSP` E`)` | `'NIL` | `'T` | **var**

where **var** is any symbol, denoting a universally quantified variable.

The axioms of the theory of list structures, aside from the standard axioms of equality and conjunction, are shown in Figure 5.1. Nelson and Oppen show not only that this theory is decidable, but that, if limited to formulas involving conjunctions of equalities and negated equalities, it can be decided with complexity $O(N \times \mathsf{Log}(N))$, where N is the size of the formula. The size of a formula here is equal to the sum of the sizes of all terms in the formula, where a term's size is the number of leaves in its parse tree (e.g., `A` has size 1, `(CONS A B)` has size 3, and `(CONS (CONS A B) (CAR X))` has size 6).

Nelson and Oppen also show, in a separate paper, that decision procedures for any two theories sharing only equality and propositional symbols can be combined into a decision procedure for the combined theory. Thus, a decidable theory axiomatizing ACL2's `EQUAL` and `IF` primitives can be combined with the theory of lists to form a decidable theory including `CAR`, `CDR`, `CONS`, `CONSP`, `IF`, and `EQUAL`. The decidable theory can further be combined with a decision procedure for uninterpreted functions. Also unrollable function applications, by definition, can be unrolled into expressions involving only previously defined functions and primitives. The SULFA subclass is defined to be exactly this decidable subclass, including function applications that can be unrolled into `CAR`, `CDR`, `CONS`, `CONSP`, `IF`, `EQUAL`, and uninterpreted functions.

As suggested by the intuition, the theory of list structures within ACL2 is decidable and can be combined with many other decidable theories. In practice, however, some complications arise:

- The axioms in Figure 5.1 are all theorems provable from ACL2's ground-zero theory. Thus, any theorem proven by Nelson and Oppen's procedure is a theorem in the ACL2 logic. However, the converse is not true. ACL2 formulas in the language of the theory of lists may be provable from the ground zero ACL2 theory but not provable from the axioms in the theory of list structures. For example, `(CAR 'NIL) = 'NIL`

1. 'T ≠ 'NIL
2. (CONSP X) = 'T ∨ (CONSP X) = 'NIL
3. (CONSP (CONS X Y)) = 'T
4. (CONSP X) ⊢ (CONS (CAR X) (CDR X)) = X
5. (CAR (CONS X Y)) = X
6. (CDR (CONS X Y)) = Y

Figure 5.1: Axioms in the traditional theory of list structures.

and (CAR 'T) = 'NIL are provable from the ACL2 ground zero theory, but not provable from the axioms in Figure 5.1.

- We do not want to limit ourselves to the primitives CAR, CDR, CONS, CONSP, IF, and EQUAL. Since any ACL2 formula involving primitives applied to constants can be decided with ACL2's evaluation proof technique, we wish to include all primitives when applied to constants.

Due to the above complications, Chapter 6 defines a decision procedure for SULFA from scratch, rather than relying on a combination of previous, well-known decision procedures.

5.3 Simplified ACL2 Logic

This chapter defines SULFA with respect to a simplified version of the ACL2 logic. The simplified version contains no lambda expressions, contains no mutually-recursive functions, and assumes that all ACL2 formulas are of the form $\alpha \neq$ 'NIL, for some ACL2 term α. For example,

```
(F X) ≠ 'NIL
```

is in the simplified ACL2, but not the equivalent formula

$(F\ X) \neq 'NIL \wedge (F\ X) \neq 'NIL.$

None of these removals affect the expressiveness of the ACL2 logic or the decidability of SULFA. ACL2 lambda expressions can be substituted for named functions or removed by β reduction; mutually-recursive set of functions can be defined as a single function with a flag; and all ACL2 formulas can be expressed as equivalent formulas of the form $\alpha \neq$ 'NIL. For example,

$(P\ X) \rightarrow (F\ X) = (G\ X)$

is equivalent to

(IMPLIES (P X) (EQUAL (F X) (G X))) $\neq$ 'NIL.

The implementation of the SULFA recognizer and SULFA solver available with the ACL2 distribution does support lambda functions and mutually recursive functions. An ACL2 formula with lambda functions and mutually recursive functions is in SULFA if it would be in SULFA were each lambda function replaced with a named function and each mutually recursive function defined with a flag.

Users of the ACL2 theorem prover only write terms, not formulas. When a user proves an ACL2 term α, they actually are proving the formula $\alpha \neq$ 'NIL. Thus there is no need, in practice, to consider more general ACL2 formulas than $\alpha \neq$ 'NIL.

5.4 SULFA Recognizer

This chapter provides a procedure that determines whether any ACL2 formula is in SULFA. The primary primitives of SULFA are ACL2's list, equality, and if-then-else primitives. We refer to these primitives (EQUAL, IF, CONS, CAR, CDR, and CONSP) as the *SULFA core primitives*. SULFA terms may also include uninterpreted functions, which in ACL2, as defined in Section 3.5.1, are constrained functions with minimal constraints. Terms involving other

functions may also occur, under the condition that some of the arguments to these functions are reducible to constants.

The following terminology is necessary before defining SULFA rigorously:

- Define $\mathbb{E}$ as the set of all ACL2 terms. Recall that an ACL2 term is either a constant, a variable, or the application of a function to an argument list composed of ACL2 terms. We denote terms as lisp expressions surrounded by $\ulcorner\urcorner$, though the $\ulcorner\urcorner$ may be removed when its removal does not cause ambiguity. For example, $\mathsf{f}(\ulcorner\texttt{(IF A B C)}\urcorner)$ is the value of some function f, given as input the ACL2 term `(IF A B C)`.

- Define $\mathbb{S} \subset \mathbb{E}$ as the set of all ACL2 symbols, representing either functions and variables in terms. For example, the term `(+ 4 X)` has a function symbol + and a variable symbol `X`.

- An *ACL2 function* is an ACL2 symbol denoting the name of the function.

- The *SULFA core primitives* are the ACL2 functions `IF`, `CONS`, `CAR`, `CDR`, `CONSP`, and `EQUAL`.

- $\mathsf{Fn}(E)$, given a function application $E \in \mathbb{E}$, is the ACL2 function being applied in E. For example, $\mathsf{Fn}(\ulcorner\texttt{(IF A B C)}\urcorner) = \ulcorner\texttt{IF}\urcorner$.

- $\mathsf{NA}(E)$, given a function application $E \in \mathbb{E}$, is its number of arguments. For example, $\mathsf{NA}(\ulcorner\texttt{(IF A B C)}\urcorner) = 3$.

- $\mathsf{Arg}(i, E)$, given a natural number $i \in \mathbb{N}$ and function application term $E \in \mathbb{E}$, is the ith argument of E. For example, $\mathsf{Arg}(1, \ulcorner\texttt{(IF A B C)}\urcorner) = \ulcorner\texttt{A}\urcorner$.

- $|E|$, given any $E \in \mathbb{E}$, is its size, defined as:

$$|E| \triangleq \begin{cases} 0, & \text{if } E \text{ is a constant} \\ 1, & \text{if } E \text{ is a symbol} \\ 2 + \sum_{i=1}^{\mathsf{NA}(E)} |\mathsf{Arg}(i, E)|, & \text{otherwise.} \end{cases}$$

where all grounded ACL2 terms involving only applications of ACL2 primitives are considered constant. For example, $|\ulcorner\texttt{A}\urcorner| = 1$ (the size of the symbol A), $|\ulcorner\texttt{'A}\urcorner| = 0$ (the size of the ACL2 constant 'A), $|\ulcorner\texttt{(CONS '4 '5)}\urcorner| = 0$ ($\ulcorner\texttt{(CONS '4 '5)}\urcorner$ is also a constant), and $|\ulcorner\texttt{(IF A B)}\urcorner| = 4$.

- Define an *ACL2 history* as a sequence of ACL2 events, including events defining functions and introducing constrained functions. We say that an ACL2 formula is valid in an ACL2 history, if it is valid in the theory corresponding to that history, i.e., the ACL2 ground zero theory plus the axioms corresponding to the ACL2 events in the history.

- $\mathsf{Body}(H, f)$, given an ACL2 history H and ACL2 function f, is the term denoting the body of the function f. For example,

  $$\mathsf{Body}(H, \ulcorner\texttt{IFF}\urcorner) = \ulcorner\texttt{(IF P (IF Q 'T 'NIL) (IF Q 'NIL 'T))}\urcorner,$$

  which corresponds to the definition of the IFF primitive:

  ```
  (DEFUN IFF (P Q)
    (IF P (IF Q 'T 'NIL) (IF Q 'NIL 'T))).
  ```

  Note that when f has no body in H, then $\mathsf{Body}(H, f)$ is undefined. In particular, $\mathsf{Body}(H, f)$ is undefined if f is a constrained function or an ACL2 core primitive.

- $\mathsf{Formals}(H, f)$, given an ACL2 history H and ACL2 function f, is the set of formal parameters of f. For example, $\mathsf{Formals}(H, \ulcorner\texttt{IFF}\urcorner) = \{\ulcorner\texttt{P}\urcorner, \ulcorner\texttt{Q}\urcorner\}$.

- $\mathsf{Meas}(H, f)$, given an ACL2 history H and an ACL2 function f, is the body of the measure function of f, if one exists. For example, given that the following definition is in H

```
(DEFUN IN (A X)
  (DECLARE (XARGS :MEASURE (LEN X)))
  (IF (NOT (CONSP X))
      NIL
    (IF (EQUAL A (CAR X))
        T
      (IN A (CDR X)))))
```

then $\mathsf{Meas}(H, \ulcorner\texttt{IN}\urcorner) = \ulcorner\texttt{(LEN X)}\urcorner$.

- Given an ACL2 history H, a set $X \subset \mathbb{S}$, $E \in \mathbb{E}$, then the predicate $\mathsf{Evblp}(H, X, E)$ is defined as:

$$\mathsf{Evblp}(H, X, E) \triangleq \begin{cases} \textbf{true}, & \text{if } |E| = 0 \\ E \in X, & \text{if } |E| = 1 \\ \textbf{false}, & \text{if } \mathsf{evalFn}(H, \mathsf{Fn}(E)) \\ \bigwedge_{i \in \mathbb{N}, 1 \le i \le \mathsf{NA}(E)} \mathsf{Evblp}(H, X, \mathsf{Arg}(i, E)), & \text{otherwise.} \end{cases}$$

  where $\mathsf{evalFnp}(H, f)$ returns whether f is an evaluatable function in H, i.e., whether f is a function that can be evaluated for any constant inputs into a constant. All ACL2 primitives are evaluatable, as are all functions defined from evaluatable functions (including recursive functions). Constrained functions are **not** evaluatable.

  Intuitively, $\mathsf{Evblp}(H, X, E)$ returns whether mapping the variables in X to constants is sufficient to ensure that E can be reduced to a constant by evaluation. For example, $\mathsf{Evblp}(H, X, \ulcorner\texttt{(+ 1 A B)}\urcorner)$ is true if `A` and `B` are in X.

- $\mathsf{ConstForm}(H, X, E)$, given an ACL2 history H, a function application $E \in \mathbb{E}$, and a set of symbols $X \subset \mathbb{S}$, is a subset of the formal parameters of $\mathsf{Fn}(E)$ including the ith formal if $\mathsf{Evblp}(H, X, \mathsf{Arg}(i, E))$.

Given an ACL2 history H, a set of symbols $X \subset \mathbb{S}$, a symbol $f \in \mathbb{S} \cup \{\clubsuit\}$, and a term $E \in \mathbb{E}$, then the predicate $\mathsf{SULFAp}(H, X, f, E)$ is defined as follows (where $G \triangleq \mathsf{ConstForm}(H, X, E)$, if E is a function application):

1. **true**, if $|E| \leq 1$.
2. **false**, if $\bigvee_{1 \leq \mathsf{NA}(E)} \neg\mathsf{SULFAp}(H, X, f, \mathsf{Arg}(i, E))$.
3. **true**, if $\mathsf{Fn}(E)$ is a SULFA core primitive or an uninterpreted function in H.
4. **false**, if $\mathsf{Fn}(E)$ is a constrained function in H.
5. $\mathsf{Formals}(H, \mathsf{Fn}(E)) \subseteq G$, if $\mathsf{Fn}(E)$ is an ACL2 core primitive or a recursive function without a valid measure.
6. $\mathsf{SULFAp}(H, G, \mathsf{Fn}(E), \mathsf{Body}(H, \mathsf{Fn}(E)))$, if $\mathsf{Fn}(E)$ is not recursive.
7. $\mathsf{Evblp}(H, G, \mathsf{Meas}(H, \mathsf{Fn}(E))) \wedge \mathsf{SULFAp}(H, G, \mathsf{Fn}(E), \mathsf{Body}(H, \mathsf{Fn}(E)))$, if $\mathsf{Fn}(E) \neq f$.
8. **true**, if $X \subseteq G$.
9. $\mathsf{Evblp}(H, X \cap G, \mathsf{Meas}(H, \mathsf{Fn}(E))) \wedge \mathsf{SULFAp}(H, X \cap G, f, \mathsf{Body}(H, \mathsf{Fn}(E)))$, otherwise.

Figure 5.2: The definition of the predicate $\mathsf{SULFAp}(H, X, f, E)$. If E is a formula in H, then it is defined to be in SULFA, if and only if $\mathsf{SULFAp}(H, \emptyset, \clubsuit, E)$. We use $\clubsuit$ as a constant not equal to any ACL2 function name.

$\mathsf{ConstForm}(H, X, E)$ thus returns the formals corresponding to the constant arguments of E. For example,

$$\mathsf{ConstForm}(H, \{\ulcorner\texttt{A}\urcorner, \ulcorner\texttt{B}\urcorner\}, \ulcorner\texttt{(IFF (< (+ A B) 0) (< A C))}\urcorner) = \{\ulcorner\texttt{P}\urcorner\}.$$

since the formals of `IFF`, as previously defined, are `P` and `Q`; the argument corresponding to `P` is `(< (+ A B) 0)`, which is reducible to a constant, given that `A` and `B` are constants; and the argument corresponding to `Q` is `(< A C)`, which is not reducible to a constant.

Given the above terminology, a formula E in an ACL2 history H is in SULFA if $\mathsf{SULFAp}(H, \emptyset, \clubsuit, E)$, where the predicate $\mathsf{SULFAp}(H, X, f, E)$ is defined in Figure 5.2.

Note that ♣ is used as a constant that is not equal to (or easily confused with) an ACL2 function. Section 5.4.1 shows that SULFAp (H, X, f, E) terminates.

Intuitively, in the definition of SULFAp (H, X, f, E), $f = ♣$ if E is a top-level formula; otherwise, f is a function symbol, E is a subterm of the body of f, and X is a set of variables symbols assumed to be constant. It is probably easiest to understand the definition of SULFA by considering examples.

By condition 1 in Figure 5.2, any formula that is simply a variable or constant is in SULFA. Thus, `X`, `Y`, `'T` and `'4` are all in SULFA.

For a formula to be in SULFA all the terms inside the formula must be in SULFA. For example, if `(F (G X))` is in SULFA, then so is `(G X)`. Thus, there is no reason to make a distinction between a formula and a term.

By conditions 1, 2, and 3, any formula composed of SULFA core primitives is in SULFA. Thus,

`(CONSP (CAR X))`,

`(EQUAL (CAR (CONS X Y)) (CDR (CONS Y X)))`, and

`(EQUAL (CAR X) (CDR X))`

are in SULFA.

Condition 3 also allows uninterpreted functions (uninterpreted functions, as defined in Section 3.5.1, are constrained functions with minimal constraints). Thus, if `F` and `G` are uninterpreted functions, then terms such as `(F (G X))` and `(CAR (F (G (CONS X '5))))` are in SULFA.

Condition 4, however, rules out constrained functions that are not uninterpreted. If `F` is introduced using ACL2's constrained function feature, then not even grounded terms involving it, such as `(F '4)`, are in SULFA.

Condition 5 ensures if a term involves applications of ACL2 core primitives other than the SULFA core primitives, then those applications must be grounded. For example,

```
(EQUAL (BINARY-* '2 '4) (BINARY-+ '4 '4))
```

is in SULFA, but not

```
(EQUAL (BINARY-* 2 X) (BINARY-+ X X)),
```

because `BINARY-*` and `BINARY-+` are core primitives. Condition 5 similarly restricts recursive functions whose measure cannot be justified by the normal definition principle. Generally, this occurs when the measure has been removed. This also is the case for a couple ACL2 primitives.

Condition 6 essentially restricts non-recursive function applications to those which would be in SULFA if the application is expanded. For example, consider the following definitions:

```
(DEFUN ZP (N)
  (IF (INTEGERP N)
      (NOT (< 0 X))
    'T))

(DEFUN IMPLIES (P Q)
  (IF P (IF Q 'T 'NIL) 'T))

(DEFUN REDUCE (N X)
  (IF (< N 0) (CAR X) (CDR X)))
```

Given the above definitions, the following are SULFA terms:

`(ZP '4)`,

`(IMPLIES A (EQUAL (CAR A) (CDR B)))`,

`(REDUCE '4 A)`, and

`(REDUCE (+ '1 '2) (IF A (CONS X Y) (CONS Y Z)))`,

whereas the following are **not** SULFA terms:

`(ZP N)`,

`(REDUCE A '4)`, and

`(REDUCE N (CONS '4 '5))`.

Since `(ZP N)` expands into an expression of ACL2 core primitives other than the SULFA core primitives, only grounded applications of `ZP` can be permitted. On the other hand, since `(IMPLIES A B)` expands into a nested `IF`, any expression involving just `IMPLIES` is in SULFA. In fact, since all propositional formulas in ACL2 expand into `IF` expressions all propositional formulas are in SULFA.

The function `(REDUCE N X)` is perhaps more interesting because it expands into an expression in which `N` is inside <, but `X` is only inside `IF`, `CAR`, and `CDR`. The result is that `N` must be constant for `(REDUCE N X)` to be in SULFA, but `X` is unrestricted.

Applications of recursive functions are handled by conditions 7, 8, and 9 of the definition of SULFAp (H, X, f, E). Intuitively, recursive applications are permitted in SULFA terms, if those applications can be unrolled and that unrolling produces an expression that is in SULFA. Consider the function `(BV-NOT N X)`, which inverts a bit vector `X`, represented as a list of Booleans of length `N`.

```
(DEFUN BV-NOT (N X)
  (DECLARE (XARGS :MEASURE (IF (ZP N) 0 N)))
  (IF (ZP N)
      NIL
    (CONS (EQUAL (CAR X) NIL)
          (BV-NOT (BINARY-+ N -1) (CDR X))))).
```

Given the above definition, some examples of SULFA terms are

`(BV-NOT 8 X)`,

`(BV-NOT (+ '1 '1) (CONS X0 (CONS X1 'NIL)))`, and

`(EQUAL (BV-NOT 8 (BV-NOT 8 (BV-NOT 8 X))) (BV-NOT 8 X))`.

Whereas, the following are **not**:

`(BV-NOT N X)`,

`(BV-NOT (+ '1 N) (CONS X0 (CONS X1 'NIL)))`, and

`(EQUAL (BV-NOT N (BV-NOT N (BV-NOT N X))) (BV-NOT N X))`.

The first part of condition 7, $\mathsf{Evblp}\,(H, X, \mathsf{Meas}\,(H, \mathsf{Fn}\,(E)))$, ensures that the measure is constant. The second part $\mathsf{SULFAp}\,(H, G, \mathsf{Fn}\,(E), \mathsf{Body}\,(H, \mathsf{Fn}\,(E)))$ ensures that the body is also in SULFA.

Condition 8 and 9 represent the case where E is a recursive application. This case must be handled carefully to ensure termination of the SULFA recognizer. If condition 8 holds, then the recursive call requires fewer or the same formals to be constant as the original call. In this case, the previous, or ongoing, check is sufficient to ensure that the expansion of E is in SULFA. If condition 8 does not hold, then a new check must be undertaken with fewer formals assumed constant. For example, consider the following definition of `BV-AND`, a function that computes the conjunction of two bit vectors:

```
(DEFUN BV-AND (N X Y)
  (DECLARE (XARGS :MEASURE (IF (ZP N) 0 N)))
  (IF (ZP N)
      NIL
    (CONS (AND (CAR X) (CAR Y))
          (BV-AND (BINARY-+ N -1) (CDR Y) (CDR X)))))
```

The following are SULFA terms:

`(BV-AND 8 X Y)`,

`(BV-AND (+ '4 '4) (CONS X Y) (CONS A B))`,

`(EQUAL (BV-AND 4 (BV-NOT 4 X) Y) Z)`

`(BV-AND 2 (CONS 'T (CONS 'NIL 'NIL)) X)`.

Whereas, the following are **not**:

`(BV-AND N X Y)`, and

`(BV-AND N (BV-NOT 4 X) Y).`

Considering the last example term that was in SULFA

$$\mathsf{SULFAp}\,(H, G, \emptyset, \ulcorner\texttt{NIL}\urcorner, \ulcorner\texttt{(BV-AND 2 (CONS 'T (CONS 'NIL 'NIL)) X)}\urcorner),$$

by condition 7, reduces to

$$\mathsf{SULFAp}\,(H, \{\ulcorner\texttt{N}\urcorner, \ulcorner\texttt{X}\urcorner\}, \ulcorner\texttt{BV-AND}\urcorner, \mathsf{Body}\,(H, \ulcorner\texttt{BV-AND}\urcorner)).$$

We have defined `BV-AND` somewhat oddly in that the arguments on each recursive call are swapped. Since

$$\mathsf{ConstForm}\,(H, \{\ulcorner\texttt{N}\urcorner, \ulcorner\texttt{X}\urcorner\}, \ulcorner\texttt{(BV-AND (BINARY-+ N -1) (CDR Y) (CDR X))}\urcorner)$$

$$=$$

$$\{\ulcorner\texttt{N}\urcorner\}$$

condition 9 requires that

$$\mathsf{SULFAp}\,(H, \{\ulcorner\texttt{N}\urcorner\}, \ulcorner\texttt{BV-AND}\urcorner, \mathsf{Body}\,(H, \ulcorner\texttt{BV-AND}\urcorner)),$$

which is true. Essentially, an attempt is made to show that the body of `BV-AND` is in SULFA while assuming `X` is constant, but then it is found that in the recursive call `X` is not constant. Therefore, the body is checked again without the assumption that `X` is constant. In this case, the check succeeds because `X` did not need to be constant.

For an example where condition 9 leads to an ACL2 term not being in SULFA, consider the definition of `BV-ODD`:

```
(DEFUN BV-ODD (N X Y)
  (DECLARE (XARGS :MEASURE (IF (ZP N) 0 N)))
  (IF (ZP N)
      NIL
    (IF (< (LEN X) N)
        X
      (CONS (AND (CAR X) (CAR Y))
            (BV-NOT (BINARY-+ N -1) (CDR Y) (CDR X))))))
```

The following term is **not** in SULFA:

```
(BV-ODD 2 (CONS 'T (CONS 'NIL 'NIL)) X).
```

The evaluation of the above term follows the same pattern as in `BV-ODD`, but the `(LEN X)` test ensures that `X` really must be constant. Thus,

$$\mathsf{SULFAp}\,(H, \{\ulcorner\texttt{N}\urcorner\}, \ulcorner\texttt{BV-AND}\urcorner, \mathsf{Body}\,(H, \ulcorner\texttt{BV-AND}\urcorner)),$$

is false.

5.4.1 Termination

We prove that $\mathsf{SULFAp}\,(H, X, f, E)$ terminates by defining an ordinal that decreases on each recursive call in its definition.

All functions defined in an ACL2 history have an associated unique natural number, such that functions are defined in terms of functions with smaller unique numbers. Define $\mathsf{FnNum}\,(H, f)$, given an ACL2 history H and $f \in (\mathbb{S} \cup \{\clubsuit\})$, as:

- the unique natural number associated with f, if f is a function symbol in history H;
- 0, otherwise.

Thus, every non-recursive function application $E' \in \mathbb{E}$ such that E' is a subterm of $\mathsf{Body}(H, f)$ satisfies $\mathsf{FnNum}(H, \mathsf{Fn}(E')) < \mathsf{FnNum}(H, f)$.

Given an ACL2 history H, a set of symbols $X \subset \mathbb{S}$, a symbol $f \in \mathbb{S} \cup \{\clubsuit\}$, and a term $E \in \mathbb{E}$, the ordinal that decreases is:

$$\begin{aligned} \mathsf{ord}(H, X, f, E) \triangleq\ & \omega^3 \times \mathsf{badFn}(H, f, E) + \\ & \omega^2 \times \mathsf{FnNum}(H, f) + \\ & \omega \times |X| + \\ & |E|. \end{aligned}$$

where

- $\mathsf{badFn}(H, f, E)$ is defined to be 1, if f is not an ACL2 function; 1, if f does not have a definition in H; 1, if there exists a function application $E' \in \mathbb{E}$ that is a subterm of E satisfying $\mathsf{FnNum}(H, \mathsf{Fn}(E')) > \mathsf{FnNum}(H, f)$; and 0, otherwise.

- $|X|$ is the number of elements in the set X.

- $|E|$ is the size of the term E, as defined previously.

We now prove that the ordinal decreases on each recursive call in $\mathsf{SULFAp}(H, X, f, E)$:

- In condition 2, the ordinal decreases since the size of the term decreases, $|\mathsf{Arg}(i, E)| < |E|$, and everything else remains the same,

- In condition 6 and 7, first note that since functions are defined in terms of smaller unique numbers $\mathsf{badFn}(H, \mathsf{Fn}(E), \mathsf{Body}(H, \mathsf{Fn}(E))) = 0$. Thus, if $\mathsf{badFn}(H, f, E) = 1$ then the ordinal decreases on the recursive call. Otherwise, the unique number decreases, i.e., $\mathsf{FnNum}(H, \mathsf{Fn}(E)) < \mathsf{FnNum}(H, f)$.

- In condition 9, the size of the set of formals decreases $|X \cap G| < |X|$, while the function and history remain the same.

5.5 Efficient SULFA Recognizer

The SULFA recognizer defined by $\mathsf{SULFAp}(H, X, f, E)$ essentially has worst case cubic complexity in the size of the bodies and measure terms in H. If no functions were recursive, then it would have quadratic complexity in the size of the bodies and measures in H, because determining whether a given function application E_f is in SULFA requires, in the worst case, considering all previous definitions. Recursive functions add another level of complexity, because their bodies need to be considered, in the worst case, once for each of their formal parameters.

While cubic complexity is better than the worst-case complexity required to solve a SULFA formula, it is still too inefficient to be used on many ACL2 formulas. It is possible though to define a more efficient recognizer. The intuition behind such a recognizer comes from the following theorem, which is provable by induction on the ordinal used to prove the termination of SULFAp:

$$X \subset Y \wedge \mathsf{SULFAp}(H, X, f, E) \rightarrow \mathsf{SULFAp}(H, Y, f, E).$$

As suggested by the above theorem, it is possible for any function f to find a minimal set of formals, called the *ground formal set*, which must be constant in order for an application of f to be in SULFA (assuming the arguments of the application are all in SULFA). For example, given the definition of `IFF`:

```
(DEFUN IFF (P Q)
  (IF P (IF Q 'T 'NIL) (IF Q 'NIL 'T))).
```

Then for the following application to be in SULFA

```
(IFF (+ '4 '5) (+ X '5))
```

it is required that

$$\mathsf{SULFAp}(H, \{\ulcorner\texttt{Q}\urcorner\}, \ulcorner\texttt{IFF}\urcorner, \ulcorner\texttt{(IF P (IF Q 'T 'NIL) (IF Q 'NIL 'T))}\urcorner),$$

which is true since nested `IF` terms are in SULFA. Note that the assumption that `Q` is constant is unnecessary. The following ACL2 term is also in SULFA:

```
(IFF (IFF A B) (IFF B C)).
```

In fact, any ACL2 term composed of only `IFF` applications is in SULFA. Thus, we say that the ground formal set of `IFF` is empty.

On the opposite end of the spectrum, consider the constrained functions other than uninterpreted functions. By case 4 in the definition of SULFAp illustrated in Figure 5.2, no ACL2 term involving such functions can occur in a SULFA formula. We say that such functions have *no valid ground formal set*.

Thus the most restricted functions, in terms of their occurrence in SULFA formulas, are those with no valid ground formal set, whereas the least restricted functions are those with an empty ground formal set. For example, `(F X Y)` is in SULFA only if the ground formal set of `F` is empty. On the other hand, if the ground formal set of `F` is equal to its formals, then `F` can only occur in SULFA if its arguments are constant, such as in `(F (+ '1 '4) '5)`. Finally, if `F` has no valid ground formal set, then it cannot occur in any SULFA formula.

The ACL2 core primitives all have valid sets of ground formals. If an ACL2 core primitive is a SULFA core primitive (i.e., the tree primitives, the if-then-else primitive, or the equality primitive), then its ground formal set is the empty set; otherwise the ground formal set of any ACL2 core primitive is equal to its full set of formals. For example, the ground formal set of `(CONS X Y)` is the empty set, whereas the ground formal set of `(BINARY-+ X Y)` is the full set of formals, $\{\ulcorner \texttt{X} \urcorner, \ulcorner \texttt{Y} \urcorner\}$.

For a constrained function f to have a valid set of ground formals, it must be an uninterpreted function (as defined in Section 3.5.1), i.e., it must have minimal constraints. Uninterpreted functions have a ground formal set equal to the empty set.

For any function f with a definition in an ACL2 history H, its ground formal set can be determined from its *ground formal dependency graph*, which is defined as follows:

- Each node in the graph is either ♠ or a formal parameter of f. We use ♠ because it is not equal to (and should not be confused with) any ACL2 symbol.

  Intuitively, ♠ represents the empty set of ground formals and each edge from v_s to v_t in the ground formal dependency graph denotes a dependence that v_t be in the ground formal set if v_s is in the set.

- A subset of the nodes are marked as *failure nodes*. A failure node denotes a node that cannot be part of the valid set of ground formals. If ♠ is a failure node, or if some dependency leads to a failure node, then there is no valid set of ground formals.

  A node v is marked as a failure node if **any** of the following conditions apply:

  1. $v = ♠$ and there exists some term $E \in \mathbb{E}$ such that E is a subterm of the body of f and $\mathsf{Fn}\,(E)$ has no valid set of ground formals.

     Intuitively, f cannot be in any SULFA formula because it is defined from a function that cannot be in any SULFA formula.

  2. $v = ♠$ and there exists a term $E \in \mathbb{E}$ and a natural number $i \in \mathbb{N}$ such that E is a subterm of the body of f, the ith formal of E is in the ground formal set of $\mathsf{Fn}\,(E)$, and $\neg\mathsf{Evblp}\,(H, \mathsf{Formals}\,(H, f), \mathsf{Arg}\,(i, E))$.

     Intuitively, f cannot be in any SULFA formula, because a function occurring in the body of f requires an argument be constant that, in the body of f, cannot be evaluated. For example, if `(G (H X))` is in the body of f, `H` is not executable, and the ground formal set of `G` includes its formal argument, then ♠ is a failure node.

  3. If v is the ith formal of f and there exists a recursive application $E \in \mathbb{E}$ in the body of f (i.e. E is a subterm of the body and $\mathsf{Fn}\,(E) = f$), and

     $$\neg\mathsf{Evblp}\,(H, \mathsf{Formals}\,(H, f), \mathsf{Arg}\,(i, E)).$$

     Intuitively, v cannot be part of the ground formal set because the body of f contains a recursive call in which v cannot be evaluated. For example, if

(F (G X)) is a recursive application in the body of F and G cannot be executed, then X is a failure node.

- An edge occurs from v_s (the source vertex) to v_t in the ground formal dependency graph of f if **any** of the following conditions hold:

  1. $v_s = \spadesuit$, v_t is the ith formal of f, and there exists an $E \in \mathbb{E}$ and $i \in \mathbb{N}$ such that E is a subterm of the body of f, $\mathsf{Fn}(E) \neq f$, the ith formal of $\mathsf{Fn}(E)$ is in the ground formal set of $\mathsf{Fn}(E)$, and v_t is a subterm of $\mathsf{Arg}(i, E)$.

     Intuitively, v_t must be in the ground formal set of f if v_t is part of an expression that must be mapped to a constant in order for some function application in the body of f to be in SULFA. For example, if the body of the definition of F is

     ```
     (IF (INTEGERP (+ N M))
        (F (1- N) M (CDR X) B (CDR X) (CDR Y))
       Y)
     ```

     then an edge goes from ♠ to N and from ♠ to M, since INTEGERP and BINARY-+ are ACL2 core primitives, but not SULFA core primitives. N and M need to be constant so that INTEGERP and BINARY-+ can be removed by evaluation.

  2. $v_s = \spadesuit$ and f is a recursive function without an evaluatable measure.

     Intuitively, such a function cannot be unrolled, but it may be evaluated, so its ground formal set is its complete set of formals.

  3. $v_s = \spadesuit$, f is a recursive function, and v_t is a subterm of the measure of f.

     Intuitively, the measure must be constant in order for the function to be unrolled. For example, if the measure of f is (NFIX X), then an edge goes from ♠ to X, denoting that X must be constant in order for f to be unrolled.

  4. v_s is the ith formal of f and there exists a recursive application E (a subterm of the body of f such that $\mathsf{Fn}(E) = f$) such that v_t is a subterm of $\mathsf{Arg}(i, E)$.

For example, if `(F (1- A) (CONS B C) 0)` is a recursive call in the body of `F`, and `F` has formal parameters `A`, `B`, and `C` respectively, then an edge goes from `A` to `A`, from `B` to `B`, and `C` to `B`. Intuitively, this is because for `B` to be constant in the recursive call, `(CONS B C)` must evaluate to a constant.

Figure 5.3 shows an example function definition and its associated ground formal dependency graph. Since `N` occurs in the measure of f and as an argument of `ZP`, there is a node from ♠ to `N`. There is also a node from ♠ to `A`, since `A` occurs as an argument of <. The other edges follow the dependencies implied by the two recursive calls in the body of `EXAMPLE`. Furthermore, the node `D`, corresponding to the fifth formal of `EXAMPLE`, is marked as a failure node because `F`, which occurs in the fifth argument of a recursive call, is not executable.

The ground formal set of a defined function f can be determined by computing the set of nodes R_f reachable from node ♠ in f's ground formal dependency graph. The function f has no valid ground formal set if R_f contains a failure node. Otherwise, the ground formal set of f is the set of formals contained in R_f.

The set of nodes reachable from ♠ in Figure 5.3 is $\{♠, \ulcorner\texttt{N}\urcorner, \ulcorner\texttt{A}\urcorner, \ulcorner\texttt{B}\urcorner, \ulcorner\texttt{E}\urcorner\}$. Since none of these are failure nodes, the set of ground formals of `EXAMPLE` is $\{\ulcorner\texttt{N}\urcorner, \ulcorner\texttt{A}\urcorner, \ulcorner\texttt{B}\urcorner, \ulcorner\texttt{E}\urcorner\}$. Thus, an application of `EXAMPLE` is in SULFA if its first, second, and sixth arguments are constant.

Note that if we assume that the ground formal sets for all previously defined functions have been determined, then the ground formal dependency graph can be constructed for a newly defined function in linear time with respect to the size of its body and its measure function's body. Also, computing the set of reachable nodes is linear in the number of edges in the graph. Thus the ground formal sets for all functions in a history can be determined in linear time with respect to the size of the events in that history. Once the ground formal sets have been computed for all functions occurring in a formula, it requires only a single traversal through an ACL2 formula to determine whether that formula is in SULFA.

```
(DEFUN EXAMPLE (N A B C D E)
  (DECLARE (XARGS :MEASURE (NFIX N)))
  (IF (ZP N)
      (EQUAL (< 0 A) E)
    (IF (CAR C)
        (EXAMPLE (1- N) B B (EQUAL C D) (F B) E)
      (EXAMPLE (1- N) E E E E E))))
```

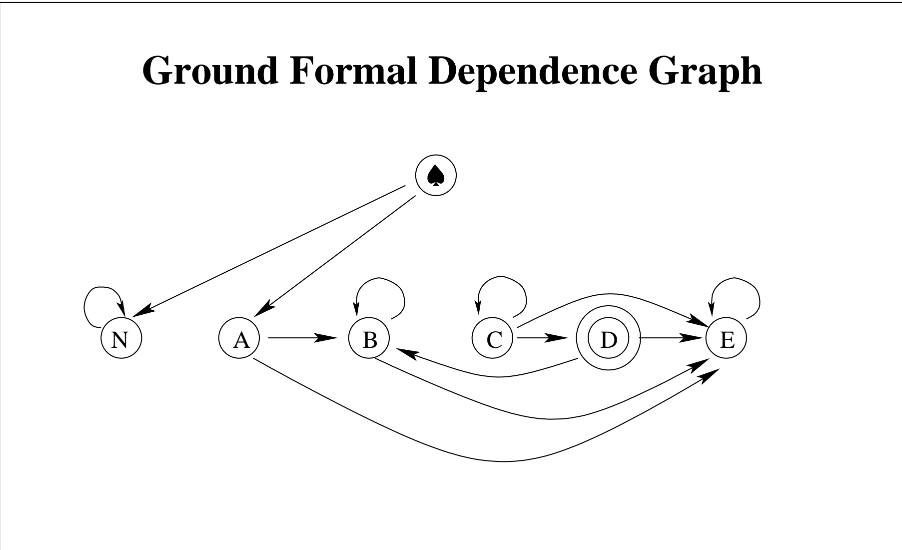


Figure 5.3: An example definition and its ground formal dependency graph. In the above definition, the ACL2 primitive `ZP`, defined in Figure 3.3, checks whether a number is equal to 0 and the function `F` is uninterpreted. The primitive `NFIX`, also defined in Figure 3.3, is the identity function over natural numbers, and returns 0 for any other inputs. A double circle in the dependency graph represents a failure node. In order for an application of `EXAMPLE` to be unrollable all formals reachable from ♠ (i.e., `N`, `A`, `B`, and `E`) must be mapped to constants.

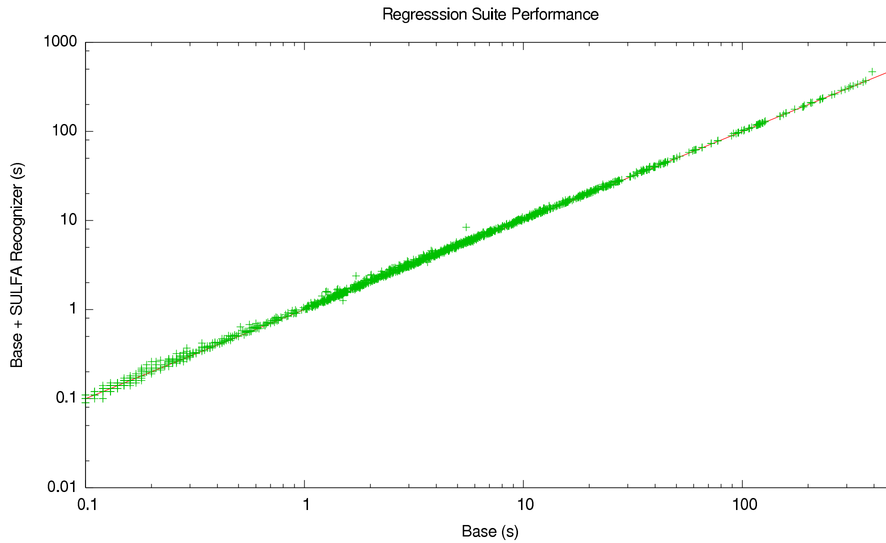


Figure 5.4: A plot comparing the time required to check whether formulas are in SULFA compared to the time required to prove them using the ACL2 theorem prover. Each point represents a file in the ACL2 regression suite. The x-axis is the total proof time to prove all formulas in a given file and the y-axis is the total proof time in addition to the time required to check whether all formulas in the same file are in SULFA. A point is on the line if no measurable time is required to check whether the formulas in its file are in SULFA.

5.6 Results

Since the SULFA recognizer described in Section 5.5 has linear complexity (with respect to the size of the SULFA formula plus the sizes of the bodies of its relevant functions and their measures), it is not surprising that it is reasonably efficient. Figure 5.4 compares the time needed to recognize whether a formula is in SULFA to the time needed by ACL2 to prove the formula. 1,249 formulas were found to be in SULFA and 37,397 were found not to be in SULFA (3.2% are in SULFA). The total proof time without the SULFA check was 5 hours, 12 minutes. An additional 32 minutes were required to determine whether each formula was in SULFA (10.1% of the proof time). [1]

[1] These results were obtained on a Pentium® 4, 3.0 GHz, dual processor with 2 gigabytes of random access memory. ACL2 version 3.1 was used, running under GNU Common Lisp version 2.6.7.

5.7 Summary

This chapter identifies a subclass of ACL2 formulas, SULFA, for which a decision procedure is presented in Chapter 6. Unlike most previously identified decidable subclasses of first-order logic, SULFA is not a single decidable theory, but instead an infinite set of decidable theories, since it has a principle for sound extension with new definitional axioms while maintaining decidability.

Furthermore, SULFA is tightly integrated with ACL2, allowing procedures that determine the validity of SULFA formulas (SULFA solvers) to be used as proof techniques within the ACL2 theorem prover. Chapter 7 describes one such SULFA solver based, which makes use of Boolean SAT solvers.

We believe developing decision procedures that operate on primitives and can solve properties about user-defined functions is the key to developing a tight integration with a general-purpose theorem prover. Furthermore, such work may promote a deeper understanding of the decidable space of formulas within the ACL2 logic, and, perhaps, first-order logic itself.

5.8 Development and Bibliographic Notes

SULFA was first described at the 3rd International Joint Conference on Automated Reasoning (IJCAR 2006) [70]. This chapter differs, however, from the description in IJCAR 2006 in that it is more rigorous and that uninterpreted functions and equality have been added.

Sophisticated decision procedures have been previously integrated with general-purpose theorem provers. The PVS general-purpose theorem prover contains decision procedures for the μ-calculus, linear arithmetic, and tree structures [61, 81, 12]. One way in which SULFA differs from such decidable domains is that SULFA can be extended with new function symbols and axioms. Similarly decision procedures in the HOL theorem prover, such as the SAT-based decision procedure for propositional logic [34], are usually

restricted to a finite set of functions. One exception is the HOL-Voss System, which supports model checking within the HOL theorem prover [36]. The HOL-Voss system also makes use of the HOL type system to restrict formulas into a finite domain unique to the HOL-Voss system. This differs from SULFA in that SULFA, like ACL2, is untyped, and universally quantifies over the entire, infinite domain of ACL2 objects. Also, since HOL-Voss uses its own type system, no formulas created by users unaware of the HOL-Voss system can be verified by it.

The ACL2 theorem prover also has been integrated with a μ-Calculus model checker [43], the SMV model checker [69], the SixthSense model checker [77], and UCLID [44]. The distinguishing feature of SULFA is that it is defined directly from ACL2's ground-zero primitives (in fact, SULFA is built on top of ACL2's undefined, core primitives) and can be extended with new function symbols. In that since, SULFA is more closely related to ACL2's built-in BDD proof technique, written by Matt Kaufmann and based on work by J Moore [53]. SULFA differs from the BDD system in that the BDD system uses a less automatic approach for function unrolling, which relies on user-generated rewrite rules, and the BDD system requires hypotheses mapping variables directly into the Boolean domain.

Chapter 6

A SULFA Decision Procedure

6.1 Introduction

Recall that we say that SULFA is a *decidable subclass* of ACL2 because it meets the following two conditions:

1. A terminating procedure exists that recognizes whether any ACL2 formula is in SULFA.
2. A terminating procedure exists that decides whether any formula in SULFA is provable in ACL2.

Chapter 5 presented a procedure that recognizes whether an ACL2 formula is in SULFA. This chapter completes the description of SULFA as a decidable subclass by presenting a decision procedure for all SULFA formulas. Note that for SULFA to be a decidable subclass there is no need for the decision procedure to be efficient. The one presented in this chapter is not efficient, but instead is intended to be as simple as possible. Chapter 7 shows how to create a more efficient SULFA solver using SAT solvers.

This chapter reuses the terminology and simplified ACL2 logic from Chapter 5 to present a decision procedure for SULFA. First, Section 6.2 shows how a SULFA formula

may be reduced to a formula involving only SULFA core primitives, uninterpreted functions, variables, and constants. Section 6.3 then shows how uninterpreted functions can be removed. Finally, Section 6.4 presents a decision procedure for any ACL2 formula involving only SULFA core primitives, variables, and ACL2 constants. Section 6.5 then discusses the problem of generating counterexamples to invalid SULFA formulas.

6.2 Unrolling SULFA Formulas

This section presents a method for reducing a SULFA formula into a formula involving only uninterpreted functions and the SULFA core primitives. We begin by introducing the following terminology:

- $\mathsf{MakeFn}(f, E_1, E_2, ...E_n)$, given n ACL2 terms $E_1, E_2, ...E_n$ and a $f \in \mathbb{S}$, is the ACL2 term representing the application of the function f to arguments $E_1, E_2, ...E_n$. For example,

  $\mathsf{MakeFn}(\ulcorner\texttt{CAR}\urcorner, \ulcorner\texttt{X}\urcorner) = \ulcorner\texttt{(CAR X)}\urcorner$, and

  $\mathsf{MakeFn}(\ulcorner\texttt{F}\urcorner, \ulcorner\texttt{(G X)}\urcorner, \ulcorner\texttt{Y}\urcorner) = \ulcorner\texttt{(F (G X) Y)}\urcorner$.

- $\mathsf{MakeEq}(X, Y) \triangleq \mathsf{MakeFn}(\ulcorner\texttt{EQUAL}\urcorner, X, Y)$.

- $\mathsf{MakeIf}(X, Y) \triangleq \mathsf{MakeFn}(\ulcorner\texttt{IF}\urcorner, X, Y)$.

- $\mathsf{MakeCar}(X, Y) \triangleq \mathsf{MakeFn}(\ulcorner\texttt{CAR}\urcorner, X, Y)$.

- $\mathsf{MakeCdr}(X, Y) \triangleq \mathsf{MakeFn}(\ulcorner\texttt{CDR}\urcorner, X, Y)$.

- $\mathsf{MakeCons}(X, Y) \triangleq \mathsf{MakeFn}(\ulcorner\texttt{CONS}\urcorner, X, Y)$.

- $\mathsf{MakeConsp}(X, Y) \triangleq \mathsf{MakeFn}(\ulcorner\texttt{CONSP}\urcorner, X, Y)$.

- $\mathsf{MakeImp}(X, Y) \triangleq \mathsf{MakeFn}(\ulcorner\texttt{IMPLIES}\urcorner, X, Y)$.

- $\mathsf{MakeImpEq}(X, Y, Z) \triangleq \mathsf{MakeImp}(X, \mathsf{MakeEq}(Y, Z))$.

- $\mathsf{EvPrf}(H, E)$, given an ACL2 history H and an ACL2 term E, is the result of applying ACL2's evaluation proof strategy to E. As described in Section 3.6.5, the evaluation proof strategy is a terminating technique that reduces evaluatable terms to constants by evaluating executable ACL2 functions. For example,

$$\mathsf{EvPrf}(H, \ulcorner\texttt{(+ 4 5)}\urcorner) = \ulcorner\texttt{9}\urcorner,$$

  whereas

$$\mathsf{EvPrf}(H, \ulcorner\texttt{(+ X 4)}\urcorner) = \ulcorner\texttt{(+ X 4)}\urcorner.$$

- $\mathsf{EvPrfp}(H, E)$, given an ACL2 history H and an ACL2 term E, is the predicate that returns true if the evaluation proof strategy successfully proves that E is not `NIL`. Thus,

$$\mathsf{EvPrfp}(H, E) \triangleq |\mathsf{EvPrf}(H, E)| = 0 \wedge \mathsf{EvPrf}(H, E) \neq \ulcorner\texttt{NIL}\urcorner.$$

  where $|\mathsf{EvPrf}(H, E)| = 0$ is whether $\mathsf{EvPrf}(H, E)$ returns a constant.

- $\mathsf{Expd}(H, E)$, given a history H and a function application $E \in \mathbb{E}$, is the *expansion* of the top-level function application in E. To be more precise,

$$\mathsf{Expd}(H, E) \triangleq \mathsf{Body}(H, \mathsf{Fn}(E))/\sigma,$$

  where σ is the substitution mapping the formals of $\mathsf{Fn}(E)$ to the corresponding arguments of E. For example, given the definition of `IFF`

```
(DEFUN IFF (P Q)
  (IF P (IF Q 'T 'NIL) (IF Q 'NIL 'T))).
```

  Then $\mathsf{Expd}(H, \ulcorner\texttt{(IFF (IFF X Y) Z)}\urcorner)$ is

```
(IF (IFF X Y) (IF Z 'T 'NIL) (IF Z 'NIL 'T)).
```

- $\mathsf{ExpdM}(H, E)$, given a history H and a function application $E \in \mathbb{E}$, is the *expansion of the measure* of E. To be more precise,

$$\mathsf{ExpdM}(H, E) \triangleq \mathsf{Meas}(H, \mathsf{Fn}(E))/\sigma,$$

where σ is the substitution mapping the formals of $\mathsf{Fn}(E)$ to the corresponding arguments of E. For example, given the definition of `BV-NOT`

```
(DEFUN BV-NOT (N X)
  (DECLARE (XARGS :MEASURE (IF (ZP N) 0 N)))
  (IF (ZP N)
      NIL
    (CONS (EQUAL (CAR X) NIL)
          (BV-NOT (BINARY-+ N -1) (CDR X))))).
```

Then $\mathsf{ExpdM}(H, \ulcorner$`(BV-NOT 8 (F X))`$\urcorner)$ is `(IF (ZP 8) 0 8)`.

The function $\mathsf{UnRoll}(H, f, o, E)$ is defined in Figure 6.1. By Theorem 1, UnRoll terminates for all valid inputs. Furthermore, given a SULFA formula F in an ACL2 history H, by Theorem 3, F is valid if and only if $\mathsf{UnRoll}(H, \clubsuit, \ulcorner 0 \urcorner, F)$ is valid and, by Theorem 2, $\mathsf{UnRoll}(H, \clubsuit, \ulcorner 0 \urcorner, F)$ contains only SULFA core primitives and uninterpreted functions. Therefore, any SULFA formula can be automatically reduced to a formula involving only SULFA core primitives and uninterpreted functions. For example, consider the following SULFA formula, which we name *cadrBvNot*

```
(IMP (CAR (CDR (BV-NOT 2 X))) (EQUAL (CAR (CDR X)) NIL))
```

where the `BV-NOT` function is the same function defined previously and `IMP` is a simpler version of `IMPLIES`, defined as

```
(DEFUN IMP (X Y) (IF X Y 'T))
```

Given an ACL2 history H, $f \in (\mathbb{S} \cup \{\clubsuit\})$, $o \in \mathbb{E}$, and $E \in \mathbb{E}$, define $\mathsf{UnRoll}(H, f, o, E)$ as

1. E, if $|E| < 2$.
2. $\ulcorner$`'ERROR`$\urcorner$, if $\neg\mathsf{Evblp}(H, \emptyset, o) \vee ((f \neq \clubsuit) \wedge (\mathsf{FnNum}(H, f) \leq \mathsf{FnNum}(H, \mathsf{Fn}(E))))$.
3. $\mathsf{EvPrf}(H, E)$, if $\mathsf{Evblp}(H, \emptyset, E)$.
4. E_{rec}, if $\mathsf{Fn}(E)$ has no definition in H, i.e., it is a constrained function or a core primitive.
5. $\mathsf{UnRoll}(H, \mathsf{Fn}(E), \clubsuit, \mathsf{Expd}(H, E_{rec}))$, if $\mathsf{Fn}(E)$ is not recursive.
6. $\mathsf{UnRoll}(H, \mathsf{Fn}(E), o_{rec}, \mathsf{Expd}(H, E_{rec}))$, if $\mathsf{Fn}(E) \neq f$.
7. $\mathsf{UnRoll}(H, f, o_{rec}, \mathsf{Expd}(H, E_{rec}))$, if $\mathsf{EvPrfp}(H, \mathsf{MakeFn}(\ulcorner \mathsf{0<} \urcorner, o_{rec}, o))$.
8. $\ulcorner$`'IRRELEVANT`$\urcorner$, otherwise.

where

- $o_{rec} \triangleq \mathsf{ExpdM}(H, E_{rec})$
- E_{rec} is the application of $\mathsf{Fn}(E)$ such that the ith argument of E_{rec} is $\mathsf{UnRoll}(H, f, o, \mathsf{Arg}(i, E))$.

Figure 6.1: Definition of $\mathsf{UnRoll}(H, f, o, E)$, which is used to unroll the user-defined functions in a SULFA formula. Note that $\mathsf{MakeFn}(\ulcorner \mathsf{0<} \urcorner, x, y)$ is an ACL2 term representing whether the term x represents a smaller ordinal than the term y.

UnRoll $(H, \clubsuit, \ulcorner \mathtt{0} \urcorner, \mathit{cadrBvNot})$ is therefore equal to

```
(if (car
     (cdr
      (if nil nil
        (cons (equal (car x) nil)
              (if nil nil
                (cons (equal (car (cdr x)) nil)
                      (if t nil
                        (cons (equal (car (cdr (cdr x))) nil)
                              'irrelevant))))))))
    (equal (car (cdr x)) nil)
  t)
```

Note that (BV-NOT 0 (CDR (CDR X))) is unrolled into

```
(IF T NIL (CONS (IF (CAR (CDR (CDR X))) NIL T) 'IRRELEVANT)).
```

The other applications of BV-NOT, and IMPLIES are expanded directly.

The term (BV-NOT -1 (CDR (CDR (CDR X)))) is replaced with 'IRRELEVANT to prevent the recursion in BV-NOT from continuing without end. In this case, as is common in practice, the replacement can be justified directly by simplifying IF terms with constant conditions.

In general, however, more sophisticated reasoning may be required. For example, consider the following ACL2 definition:

```
(DEFUN F (X)
  (DECLARE (XARGS :MEASURE 1))
  (IF (EQUAL X X)
      T
      (F X)))
```

Since the `IF` condition is a valid theorem in ACL2, the theorem prover can reduce `(F X)` to `T` and thus allow the introduction of `F` with the above definition.

The SULFA formula `(F X)` unrolls into `(IF (EQUAL X X) T 'IRRELEVANT)`. Thus, an axiom regarding `EQUAL` is needed to prove that the unrolled term is equivalent to the original formula. For any E, the task of proving $\mathsf{UnRoll}(H, \clubsuit, \ulcorner 0 \urcorner, E)$ equivalent to E, on a case by case basis, can be as difficult as determining the validity of an arbitrary SULFA formula. However, instead of proving the equivalence on a case by case basis, we rely on the proof of termination, which implies that any recursive call in which a measure fails to decrease is in an impossible `IF` branch.

6.2.1 Correctness

Given an ACL2 history H, $f \in (\mathbb{S} \cup \{\clubsuit\})$, $o \in \mathbb{E}$, and $E \in \mathbb{E}$, define the ordinal $\mathsf{UnRollOrd}(H, f, o, E)$ as

$$\mathsf{UnRollOrd}(H, f, o, E) \triangleq \begin{cases} \epsilon_0^3, & \text{if } f = \clubsuit \\ \epsilon_0^2 \times \mathsf{FnNum}(H, f) + \epsilon_0 \times \mathsf{ord}(H, o) + |E|, & \text{otherwise.} \end{cases}$$

where

- $\mathsf{ord}(H, o)$, is the ordinal represented by $\mathsf{EvPrf}(H, o)$, if $\mathsf{EvPrf}(H, o)$ returns an ACL2 ordinal constant; otherwise, $\mathsf{ord}(H, o) \triangleq 0$. Recall that an ACL2 ordinal constant represents an ordinal less than ϵ_0.
- $\mathsf{FnNum}(H, f)$ has the same definition as in Section 5.4.1.

Theorem 1. $\mathsf{UnRoll}(H, f, o, E)$ *terminates for all valid inputs.*

Proof Sketch: The ordinal $\mathsf{UnRollOrd}(H, f, o, E)$ decreases on each recursive call in Figure 6.1. □

Lemma 1. $\mathsf{SULFAp}(H, X, f, E) \rightarrow \mathsf{SULFAp}(H, X, f, \mathsf{Arg}(i, E))$*, where H is an ACL2 history, X a set of symbols, $f \in \mathbb{S} \cup \{\clubsuit\}$, $E \in \mathbb{E}$ is a function application, and i is a natural number less than* $\mathsf{NA}(E)$.

Proof Sketch: Trivial from the definition of SULFAp. □

Lemma 2. $\mathsf{SULFAp}(H, X, f, E) \rightarrow \mathsf{SULFAp}(H, X, f, \mathsf{Expd}(H, E))$*, where H is an ACL2 history, X a set of symbols,* $f \in \mathbb{S} \cup \{\clubsuit\}$*, and* $E \in \mathbb{E}$ *is a function application.*

Proof Sketch: First note, by induction on $|E|$

$$\mathsf{Evblp}(H, X, E) \rightarrow \mathsf{Evblp}(H, X, \mathsf{Expd}(H, E)).$$

It follows that

$$\mathsf{ConstForm}(H, X, E) = \mathsf{ConstForm}(H, X, \mathsf{Expd}(H, E)).$$

The theorem thus follows from the definition of $\mathsf{SULFAp}(H, X, f, E)$ and induction on the ordinal used to prove termination of $\mathsf{SULFAp}(H, X, f, E)$. □

Lemma 3. $\mathsf{SULFAp}(H, \emptyset, \clubsuit, E) \rightarrow \mathsf{SULFAp}(H, \emptyset, \clubsuit, \mathsf{UnRoll}(H, f, o, E))$.

Proof Sketch: The theorem follows from induction on $\mathsf{UnRollOrd}(H, f, o, E)$, the definition of UnRoll, Lemma 1, and Lemma 2. □

Theorem 2. *If* $\mathsf{SULFAp}(H, \emptyset, \clubsuit, E)$ *and f is a function applied in* $\mathsf{UnRoll}(H, g, o, E)$*, then either f is an uninterpreted function or a SULFA core primitive.*

Proof Sketch: By, induction on $\mathsf{UnRollOrd}(H, f, o, E)$, the theorem reduces to the case when $\mathsf{Fn}(E)$ is a constrained function or a core primitive. Thus, the theorem follows from the definition of SULFAp given in Figure 5.2. □

Given three ACL2 terms E, A, and C, define

$$\mathsf{SubConj}(E, A, C) \triangleq \begin{cases} \{C\} & \text{if } E = A \\ \emptyset, & \text{if } |E| < 2 \\ \bigcup_{i \in \mathbb{N}, 1 \leq i \leq 3} \mathsf{SubConj}(\mathsf{Arg}(i, E), A, \mathsf{ifConj}(i, C)), & \text{if } \mathsf{Fn}(E) = \ulcorner \mathtt{IF} \urcorner \\ \bigcup_{i \in \mathbb{N}, 1 \leq i \leq \mathsf{NA}(E)} \mathsf{SubConj}(\mathsf{Arg}(i, E), A, C), & \text{otherwise.} \end{cases}$$

where

$$\mathsf{ifConj}\,(i, C) \triangleq \begin{cases} C, & \text{if } i = 1 \\ \mathsf{MakeIf}\,(\mathsf{Arg}\,(1, E), C, \ulcorner\texttt{NIL}\urcorner), & \text{if } i = 2 \\ \mathsf{MakeIf}\,(\mathsf{Arg}\,(1, E), \ulcorner\texttt{NIL}\urcorner, C), & \text{otherwise.} \end{cases}$$

Intuitively, A is a subterm of E and $\mathsf{SubConj}\,(E, A, \mathtt{H})$ is the set of assumptions (encoded as ACL2 terms) under which A is relevant to E.

Lemma 4. *If* $C \in \mathsf{SubConj}\,(\mathsf{Body}\,(H, f), E, \ulcorner\texttt{T}\urcorner)$*, then* $\mathsf{MakeImp}\,(C, \mathsf{MakeFn}\,(\ulcorner\mathsf{0<}\urcorner, \mathsf{ExpdM}\,(H, E), \mathsf{Meas}\,(H, f)))$ *is a valid ACL2 formula; where H is an ACL2 history, E is an ACL2 term, and f is a recursively defined function in H.*

Proof Sketch: The theorem follows directly from the (user-guided) proof of termination of f in H. □

Lemma 5. $\mathsf{EvbIp}\,(H, \emptyset, E) \rightarrow |\mathsf{EvPrf}\,(H, E)| = 0$.

Proof Sketch: Follows directly from our knowledge of the ACL2 evaluation proof technique. The ACL2 evaluation proof technique reduces any grounded ACL2 term containing only executable functions to a constant. □

Theorem 3. *If* $\mathsf{SULFAp}\,(H, \emptyset, \clubsuit, E)$ *then* $\mathsf{UnRoll}\,(H, \clubsuit, \ulcorner\mathtt{0}\urcorner, E)$ *is a valid ACL2 formula if and only if E is a valid ACL2 formula.*

Proof Sketch: First, we generalize the theorem, proving that if $\mathsf{SULFAp}\,(H, \emptyset, \ulcorner\mathtt{0}\urcorner, E)$, then $\mathsf{MakeImpEq}\,(C, E, \mathsf{UnRoll}\,(H, f, o, E))$ is a valid ACL2 formula, where

1. C is $\ulcorner\texttt{T}\urcorner$ if $f = \clubsuit$; otherwise, C is some ACL2 term such that $C \in \mathsf{SubConj}\,(\mathsf{Body}\,(H, f))/\sigma, E, \ulcorner\texttt{T}\urcorner)$.

2. o is $\ulcorner\mathtt{0}\urcorner$, if $f = \clubsuit$; otherwise, o is $\mathsf{Meas}\,(H, f)/\sigma$.

3. f is either $\clubsuit$ or the name of a function defined in H such that E is a subterm of $\mathsf{Body}\,(H, f)/\sigma$.

4. σ is any substitution mapping ACL2 variable symbols to ACL2 terms.

We now prove the generalization by induction on $\mathsf{UnRollOrd}\,(H, f, o, E)$. Consider each case in the definition of UnRoll illustrated in Figure 6.1:

1. Trivial, since $\mathsf{MakeEq}\,(E, E)$ is valid.

2. This case cannot occur under the theorem's assumptions.

3. Trivial, since $\mathsf{MakeEq}\,(E, \mathsf{EvPrf}\,(H, E))$ is valid.

4. $\mathsf{MakeImpEq}\,(C, E, E_{rec})$, where E_{rec} has the definition given in Figure 6.1, follows from induction. Note that unless $\mathsf{Fn}\,(E) = \ulcorner\mathtt{IF}\urcorner$ the assumptions under which E occurs are the same as the assumptions under any argument of E, i.e., $\mathsf{Fn}\,(E) \neq \ulcorner\mathtt{IF}\urcorner$ implies that for every ith argument of E

   $$\mathsf{SubConj}\,(\mathsf{Body}\,(H, f)/\sigma, E, \ulcorner\mathtt{T}\urcorner)$$

   is equal to

   $$\mathsf{SubConj}\,(\mathsf{Body}\,(H, f)/\sigma, \mathsf{Arg}\,(i, E), \ulcorner\mathtt{T}\urcorner).$$

   When, $\mathsf{Fn}\,(E) = \ulcorner\mathtt{IF}\urcorner$, this case reduces to the following ACL2 formula, which is a theorem:

```
(IMPLIES (AND (IMPLIES A (EQUAL X B))
              (IMPLIES (NOT A) (EQUAL X C)))
         (EQUAL X (IF A B C)))
```

5. By induction, and Lemma 3, the theorem reduces to

   $$\mathsf{MakeImpEq}\,(C, E, \mathsf{Expd}\,(H, E_{rec})).$$

   This is valid by the definition of $\mathsf{Fn}\,(E)$, Lemma 2, and the induction hypothesis.

6. Same reasoning as previous case, with the addition that, $\mathsf{SULFAp}(H, \emptyset, \clubsuit, E)$, by the definition of SULFAp, implies $\mathsf{EvbIp}(H, \mathsf{ConstForm}(H, \emptyset, E), E)$, which implies $\mathsf{EvbIp}(H, \emptyset, o_{rec})$.

7. Same reasoning as previous case.

8. By Lemma 4, $\mathsf{MakeImp}(C, \mathsf{MakeFn}(\ulcorner \texttt{0<} \urcorner, \mathsf{ExpdM}(H, E), o))$ is valid, and by the same reasoning as in case 4, $\mathsf{MakeImp}(C, E, E_{rec})$. Thus,

$$\mathsf{MakeImp}(C, \mathsf{MakeFn}(\ulcorner \texttt{0<} \urcorner, o_{rec}, o))$$

is a valid ACL2 formula. This simplifies, by the case assumptions and Lemma 5, to $\mathsf{MakeImp}(C, \ulcorner \texttt{NIL} \urcorner)$. Therefore, $\mathsf{MakeImpEq}(C, E, \mathsf{UnRoll}(H, f, o, E))$ is vacuously true.

□

6.3 Removing Uninterpreted Functions

Section 6.2 shows that defined functions in a SULFA formula can be removed by unrolling, leaving a formula involving only uninterpreted functions and SULFA core primitives. Next, we show that the uninterpreted functions can also be removed, leaving a formula involving only SULFA core primitives. Before explaining how to remove uninterpreted functions though, the following terminology is introduced:

- S/σ, given a set $S \subset \mathbb{E}$, is the set satisfying for all $E \in \mathbb{E}$:

$$(E \in S) \rightarrow (E/\sigma \in S/\sigma).$$

- $\mathsf{FreshV}(F)$, given $F \in \mathbb{E}$, produces a symbol not used as a variable in F. Such a function is possible since ACL2 contains an infinite namespace of variable names.

- It is possible to define a *lexicographic* ordering on ACL2 terms, and such an ordering is provided with the ACL2 theorem prover. The details of this ordering are not significant to this dissertation, except that if A is a subterm of B, then A is lexicographically smaller than (or equal to) B.

- Let F be a formula containing applications of uninterpreted functions. Then, let $\mathsf{PickUF}(F)$ be the lexicographically smallest uninterpreted function application in F.

- $\mathsf{MakeEqArgs}(X, Y)$, given two applications $X \in \mathbb{E}$ and $Y \in \mathbb{E}$ of functions of the same arity, is the ACL2 term that representing

$$\bigwedge_{i \in \mathbb{N}, 1 \leq i \leq \mathsf{NA}(X)} Arg(i, X) = Arg(i, Y).$$

  For example, $\mathsf{MakeEqArgs}(\ulcorner\texttt{(F X Y Z)}\urcorner, \ulcorner\texttt{(G A B C)}\urcorner) =$

$$\ulcorner\texttt{(AND (EQUAL X A) (EQUAL Y B) (EQUAL Z C))}\urcorner$$

- Given $A \in \mathbb{E}$, $X \in \mathbb{E}$, and $E \in \mathbb{E}$:

$$\mathsf{SubUF}(A, X, E) \triangleq \begin{cases} X, & \text{if } E = A \\ E, & \text{if } (|E| < 2) \vee (|A| < 2) \\ E_{rec}, & \text{if } \mathsf{Fn}(E) \neq \mathsf{Fn}(A) \\ \mathsf{MakeIf}(\mathsf{MakeEqArgs}(A, E_{rec}), X, E_{rec}), & \text{otherwise.} \end{cases}$$

  where E_{rec} is an application of $\mathsf{Fn}(E)$ such that the ith argument of E_{rec} is $\mathsf{SubUF}(A, X, \mathsf{Arg}(i, E))$.

  Intuitively, A is an application of an uninterpreted function, X is a variable, and E is a term in which X does not occur. In that case, $\mathsf{SubUF}(A, X, E)$ produces a term equivalent to E, with A replaced by X.

Given a formula F containing only applications of uninterpreted functions and core SULFA primitives, from Theorem 4 and Theorem 5, $\mathsf{SubUF}(\mathsf{PickUF}(F), \mathsf{FreshV}(F), F)$ is

an equivalent formula with fewer uninterpreted function applications. Thus, by repeatedly applying SUBUF, all the uninterpreted functions from F can be removed, leaving a formula with only SULFA core primitives. For example, let E be the following ACL2 term, where F and G are uninterpreted functions:

```
(IF (EQUAL (F Y) (F X))
    (EQUAL (G (F X)) (G (F Y)))
  (EQUAL (EQUAL X Y) NIL)).
```

Note that E is a valid ACL2 formula. Then SubUF(⌜(F Y)⌝, ⌜V0⌝, E) is

```
(IF (EQUAL V0 (IF (EQUAL X Y) V0 (F X)))
    (EQUAL (G (IF (EQUAL X Y) V0 (F X))) (G V0))
  (EQUAL (EQUAL X Y) NIL)).
```

Label the above formula E'. Note that E has four distinct applications of uninterpreted functions, (F Y), (F X), (G (F X)), and (G (F Y)). On the other hand, E' has only three distinct applications of uninterpreted functions, (F X), (G (IF (EQUAL X Y) V0 (F X))), and (G V0). Furthermore, we can prove E from E' by instantiation and we can prove E' from E by functional instantiation (substitute ((LAMBDA (A) (IF (EQUAL A Y) V0 (F A)))) for F).

SubUF(⌜(F X)⌝, ⌜V1⌝, E') is:

```
(IF (EQUAL V0 (IF (EQUAL X Y) V0 V1))
    (EQUAL (G (IF (EQUAL X Y) V0 V1)) (G V0))
  (EQUAL (EQUAL X Y) NIL)).
```

Label the above formula E''. E'' has only two uninterpreted function applications, and E'' is equivalent to E' by a similar justification as that used to prove E' equivalent to E.

Now, let E''' be SubUF(⌜(F X)⌝, ⌜V2⌝, E''), which is equal to:

```
(IF (EQUAL V0 (IF (EQUAL X Y) V0 V1))
    (EQUAL (IF (EQUAL (IF (EQUAL X Y) V0 V1) V0)
               V2
             (G (IF (EQUAL X Y) V0 V1)))
           V2)
  (EQUAL (EQUAL X Y) NIL)).
```

E''' has only one distinct uninterpreted function application, and E''' is equivalent to E'' by a similar justification as that used to prove E' equivalent to E.

Now, let E^{IV} be SubUF (⌜`(G (IF (EQUAL X Y) V0 V2))`⌝, ⌜`V3`⌝, E'''), which is equal to:

```
(IF (EQUAL V0 (IF (EQUAL X Y) V0 V1))
    (EQUAL (IF (EQUAL (IF (EQUAL X Y) V0 V1) V0)
               V2
             V3)
           V2)
  (EQUAL (EQUAL X Y) NIL)).
```

E^{IV} has no uninterpreted function applications, and E^{IV} is equivalent to E''' by a similar justification as that used to prove E' equivalent to E. Thus, E has been reduced to a term with no uninterpreted functions.

6.3.1 Correctness

Lemma 6. *For any ACL2 formula F and ACL2 term x, F is a valid ACL2 formula if and only if* SubUF (x, x, F) *is a valid ACL2 formula.*

Proof Sketch: The theorem follows from the definition of SubUF , by induction on $|F|$. □

Theorem 4. *Let F be a formula in history H containing only applications of SULFA core primitives and uninterpreted functions. Further assume that F contains at least one uninterpreted function application. Then, F is a valid ACL2 formula if and only if* $\mathsf{SubUF}(\mathsf{PickUF}(F), \mathsf{FreshV}(F), F)$ *is valid.*

Proof Sketch: First, we assume that $\mathsf{SubUF}(\mathsf{PickUF}(F), \mathsf{FreshV}(F), F)$ is valid and show that F is valid. Let

$$G \triangleq \mathsf{SubUF}(\mathsf{PickUF}(F), \mathsf{FreshV}(F), F)/[\mathsf{FreshV}(F) \mapsto \mathsf{PickUF}(F)].$$

By instantiation, G is valid. Furthermore,

$$G = \mathsf{SubUF}(\mathsf{PickUF}(F), \mathsf{PickUF}(F), F).$$

F then follows from $\mathsf{SubUF}(\mathsf{PickUF}(F), \mathsf{PickUF}(F), F)$ by Lemma 6.

Next, we assume F and show that $\mathsf{SubUF}(\mathsf{PickUF}(F), \mathsf{FreshV}(F), F)$ is valid. Let g be an ACL2 lambda function with the same arity as $\mathsf{Fn}(\mathsf{PickUF}(F))$ such that its body is

$$\mathsf{MakeIf}(\mathsf{MakeEqArgs}(\mathsf{PickUF}(F), A), \mathsf{FreshV}(F), A),$$

where A is the application of $\mathsf{Fn}(\mathsf{PickUF}(F))$ such that the ith argument of A is equal to the ith formal parameter of g. Let G be F except that all applications of $\mathsf{Fn}(\mathsf{PickUF}(F))$ are replaced with applications of a function g. G follows from F by functional instantiation, justified in ACL2 by Kaufmann and Moore [39]. $\mathsf{SubUF}(\mathsf{PickUF}(F), \mathsf{FreshV}(F), F)$ then follows from G by expanding applications of g and reducing

$$\mathsf{MakeIf}(\mathsf{MakeEqArgs}(\mathsf{PickUF}(F), \mathsf{PickUF}(F)), \mathsf{FreshV}(F), A)$$

to $\mathsf{FreshV}(F)$. □

Theorem 5. *Let $E \in \mathbb{E}$, $x \in \mathbb{S}$, and $A \in \mathbb{E}$ such that A contains exactly one uninterpreted function application, which occurs at the top level of A. Then,* $\mathsf{SubUF}(A, x, E)$ *contains at most the same number of distinct uninterpreted function applications as E. Furthermore,*

if A is a subterm of E, then $\mathsf{SubUF}(A, x, E)$ *contains fewer distinct uninterpreted function applications than E.*

Proof Sketch: The theorem follows from induction on $|E|$ and the definition of SubUF, since $\mathsf{MakeEqArgs}(A, E_{rec})$ only contains terms that occur as arguments to A or E_{rec}. The arguments of A have no uninterpreted function applications, and any applications added from E_{rec} are repetitious. □

6.4 A Decision Procedure for SULFA Core Primitives

This section presents a decision procedure for any ACL2 formula that contains only ACL2 constants, variables, and applications of SULFA core primitives. Before presenting the procedure, we introduce the following terminology:

- $\mathsf{SymTermp}(X, Y)$, given $X \in \mathbb{E}$ and $Y \in \mathbb{E}$, is a predicate returning true if and only if $X \in \mathbb{S}$ and X occurs as a variable in Y. For example,
  $\mathsf{SymTermp}(\ulcorner\texttt{X}\urcorner, \ulcorner\texttt{(CAR (CDR (CONS A X)))}\urcorner) = \textbf{true}$,
  $\mathsf{SymTermp}(\ulcorner\texttt{A}\urcorner, \ulcorner\texttt{(CAR (CDR (CONS A X)))}\urcorner) = \textbf{true}$,
  $\mathsf{SymTermp}(\ulcorner\texttt{Y}\urcorner, \ulcorner\texttt{(CAR (CDR (CONS A X)))}\urcorner) = \textbf{false}$, and
  $\mathsf{SymTermp}(\ulcorner\texttt{(CONS A X)}\urcorner, \ulcorner\texttt{(CAR (CDR (CONS A X)))}\urcorner) = \textbf{false}$.

- $\mathsf{MakeNotConsp}(X) \triangleq \mathsf{MakeEq}(\mathsf{MakeConsp}(X), \ulcorner\texttt{NIL}\urcorner)$. For example, $\mathsf{MakeNotConsp}(\ulcorner\texttt{(F 4)}\urcorner) = \ulcorner\texttt{(EQUAL (CONSP (F 4)) NIL)}\urcorner$.

- $\mathsf{MakeNotEq}(X, Y) = \mathsf{MakeEq}(\mathsf{MakeEq}(X, Y), \ulcorner\texttt{NIL}\urcorner)$. For example, $\mathsf{MakeNotEq}(\ulcorner\texttt{(F A)}\urcorner, \ulcorner\texttt{(G A)}\urcorner) = \ulcorner\texttt{(EQUAL (EQUAL (F A) (G A)) NIL)}\urcorner$.

- Given a set $C \subset \mathbb{E}$, and a term $E \in \mathbb{E}$, define

$$\mathsf{MakeSetImp}(C, E) \triangleq \begin{cases} E, & \text{if } C = \emptyset \\ \mathsf{MakeIf}(C_0, \mathsf{MakeSetImp}(C \setminus \{C_0\}, E), \ulcorner\texttt{T}\urcorner), & \text{otherwise.} \end{cases}$$

where C_0 is the lexicographically smallest element of C (actually any element will do, we only choose the lexicographically smallest element so that $\mathsf{MakeSetImp}$ has a precise definition). Also, note that \ is set subtraction, e.g.,

$$\{a, b, c, d\} \setminus \{a, c\} = \{a, b\}.$$

Intuitively, $\mathsf{MakeSetImp}(C, E)$ is the ACL2 representation of the predicate which returns true when the conjunction of the elements of C imply E. For example, $\mathsf{MakeSetImp}(\{\ulcorner\texttt{A}\urcorner\}, \ulcorner\texttt{(CAR B)}\urcorner\}, \ulcorner\texttt{(EQUAL X Y)}\urcorner)$ is

$$\ulcorner\texttt{(IF A (IF (CAR B) (EQUAL X Y) 'T) 'T)}\urcorner.$$

- $\mathbb{E}_{SC}$ is the subset of ACL2 terms containing only applications of SULFA core primitives.

- $\mathbb{E}_T$ is the set of ACL2 terms containing only variables, constants (which may be any ground terms involving ACL2 primitives), and applications of `CONS`. For example, `(CONS X Y)`, `(CAR (CONS '4 '5))` and `X` are in $\mathbb{E}_T$, whereas `(CAR X)` is not.

- $\mathbb{E}_{NT}$ is the set of ACL2 terms recognized by the grammar

$$\begin{aligned} \textit{negTree} &::= \texttt{(EQUAL (EQUAL } \textit{tree}\ \textit{tree}\texttt{) 'NIL)} \mid \\ &\quad\ \ \texttt{(EQUAL (CONSP } \textit{tree}\texttt{) 'NIL)} \\ \textit{tree} &::= \textbf{var} \mid \textbf{constant} \mid \texttt{(CONS } \textit{tree}\ \textit{tree}\texttt{)} \end{aligned}$$

where **var** is a variable symbol and **constant** is an ACL2 constant (which may be a ground term involving ACL2 primitives). For example,

$\ulcorner$`(EQUAL (EQUAL X Y) 'NIL)`$\urcorner \in \mathbb{E}_{NT}$,

$\ulcorner$`(EQUAL (CONSP X) 'NIL)`$\urcorner \in \mathbb{E}_{NT}$,

$\ulcorner$`(EQUAL (EQUAL X (CONS X Y)) 'NIL)`$\urcorner \in \mathbb{E}_{NT}$,

$\ulcorner$`(EQUAL X Y)`$\urcorner \notin \mathbb{E}_{NT}$,

$\ulcorner$`(EQUAL (EQUAL X Y) 'T)`$\urcorner \notin \mathbb{E}_{NT}$, and

$\ulcorner$`(EQUAL (CONSP (CAR X)) 'NIL)`$\urcorner \notin \mathbb{E}_{NT}$.

- H_{GZ} is the empty history, representing ACL2's ground zero theory.

- $\mathsf{NonConsFA}(E)$, given an $E \in \mathbb{E}$ such that $E \notin \mathbb{E}_T$, returns the lexicographically smallest subterm E' of E, such that $\mathsf{Fn}(E') \neq \ulcorner\texttt{CONS}\urcorner$ and for each ith argument of E', $\mathsf{Arg}(i, E') \in \mathbb{E}_T$.

  Intuitively, $\mathsf{NonConsFA}(E)$ picks one of the inner-most subterms of E that is not an application of `CONS`. For example,

  $\mathsf{NonConsFA}(\ulcorner$`(EQUAL (CONS (CAR (CONS A B)) (EQUAL A C)) X)`$\urcorner)$

  is `'(CAR (CONS A B))`.

- Given an $E \in \mathbb{E}$ define

$$\mathsf{GetCar}(E) \triangleq \begin{cases} \mathsf{EvPrf}(H_{GZ}, \mathsf{MakeCar}(E)), & \text{if } |E| = 0 \\ \mathsf{MakeCar}(E), & \text{if } (|E| = 1) \vee (\mathsf{Fn}(E) \neq \ulcorner\texttt{CONS}\urcorner) \\ \mathsf{Arg}(1, E), & \text{otherwise.} \end{cases}$$

  Intuitively, $\mathsf{GetCar}(E)$ is equal to the `CAR` of E, with some simplifications. For example,

  $\mathsf{GetCar}(\ulcorner$`(CONS A B)`$\urcorner) = \ulcorner$`A`$\urcorner$,

  $\mathsf{GetCar}(\ulcorner$`(+ '4 '5)`$\urcorner) = \ulcorner$`NIL`$\urcorner$,

  $\mathsf{GetCar}(\ulcorner$`A`$\urcorner) = \ulcorner$`(CAR A)`$\urcorner$, and

  $\mathsf{GetCar}(\ulcorner$`(CAR A)`$\urcorner) = \ulcorner$`(CAR (CAR A))`$\urcorner$.

- Given an $E \in \mathbb{E}$ define

$$\mathsf{GetCdr}(E) \triangleq \begin{cases} \mathsf{EvPrf}(H_{GZ}, \mathsf{MakeCdr}(E)), & \text{if } |E| = 0 \\ \mathsf{MakeCdr}(E), & \text{if } (|E| = 1) \vee (\mathsf{Fn}(E) \neq \ulcorner\texttt{CONS}\urcorner) \\ \mathsf{Arg}(2, E), & \text{otherwise.} \end{cases}$$

Intuitively, $\mathsf{GetCdr}(E)$ is equal to the CDR of E, in the same way that $\mathsf{GetCar}(E)$ is equal to the CAR of E.

- Given an $E \in \mathbb{E}$ define

$$\mathsf{GetConsp}(E) \triangleq \begin{cases} \mathsf{EvPrf}(H_{GZ}, \mathsf{MakeConsp}(E)), & \text{if } |E| = 0 \\ \mathsf{MakeConsp}(E), & \text{if } (|E| = 1) \vee (\mathsf{Fn}(E) \neq \ulcorner\texttt{CONS}\urcorner) \\ \ulcorner\texttt{T}\urcorner, & \text{otherwise.} \end{cases}$$

Intuitively, $\mathsf{GetConsp}(E)$ is equal to the CONSP of E, in the same way that $\mathsf{GetCar}(E)$ is equal to the CAR of E.

The decision procedure is broken into two components: first, the formula is simplified into a set of formulas such that all the formulas in the set are valid if and only if the original formula is valid; next, the validity of each simplified formula is determined. During simplification, each element in the set of formulas to be conjoined is maintained as a pair (C, E), representing the formula $\mathsf{MakeSetImp}(C, E)$, where $C \subset \mathbb{E}_T$ and $E \in \mathbb{E}$. The pair $(\emptyset, F)$ is the initial formula. Simplification proceeds as a series of steps of the function $\mathsf{Round}(C, E)$ simplifies a formula into a set of "simpler" formula until each formula is represented by a pair (C, E) such that $E \in \mathbb{E}_T$.

Given a set $C \subset \mathbb{E}_{NT}$ and a term $E \in \mathbb{E}_{SC}$, such that $E \notin \mathbb{E}_T$, let $A \triangleq \mathsf{NonConsFA}(E)$, $A_f \triangleq \mathsf{Fn}(A)$, $A_1 \triangleq \mathsf{Arg}(1, A)$, $A_2 \triangleq \mathsf{Arg}(2, A)$, $A_3 \triangleq \mathsf{Arg}(3, A)$, and $\mathsf{subA}(X)$ be the term formed from E by replacing all occurrences of A with X. Also, let v_0 and v_1 be unique symbols not used as variables in E or any member of C. Then, $\mathsf{Round}(C, E) \triangleq$

1. $\{(C, \mathsf{subA}(X))\}$, if $(A_f \in \{\ulcorner\texttt{CAR}\urcorner, \ulcorner\texttt{CDR}\urcorner, \ulcorner\texttt{CONSP}\urcorner\}) \wedge \mathsf{Fnp}(A_1)$, where X is $\mathsf{Arg}(1, A_1)$, $\mathsf{Arg}(2, A_1)$, or $\ulcorner\texttt{T}\urcorner$, if A_f is $\ulcorner\texttt{CAR}\urcorner$, $\ulcorner\texttt{CDR}\urcorner$, or $\ulcorner\texttt{CONSP}\urcorner$ respectively.

   Intuitively, this case simplifies `(CAR (CONS X Y))`, `(CDR (CONS X Y))`, and `(CONSP (CONS X Y))` into `X`, `Y`, and `T` respectively. For example,

   $\mathsf{Round}(\emptyset, \ulcorner\texttt{(IF (EQUAL (CAR (CONS X Y)) (CAR X)) A B)}\urcorner)$

   $=$

   $\{(\emptyset, \ulcorner\texttt{(IF (EQUAL X (CAR X)) A B)}\urcorner)\}$,

   which simplifies the ACL2 formula

   ```
   (IF (EQUAL (CAR (CONS X Y)) (CAR X)) A B)
   ```

   to

   ```
   (IF (EQUAL X (CAR X)) A B).
   ```

2. $\{(C, \mathsf{subA}(X))\}$, if $(A_f = \ulcorner\texttt{IF}\urcorner) \wedge \neg(A_1 \in \mathbb{S})$, where X is A_3 or A_2, if $A_1 = \ulcorner\texttt{NIL}\urcorner$ or $A_1 \neq \ulcorner\texttt{NIL}\urcorner$ respectively.

   Intuitively, this case simplifies `(IF NIL X Y)` to `Y` and `(IF` α `Y Z)` to `Y`, if α is a non-`NIL` constant or an application of `CONS`. For example,

   $\mathsf{Round}(\emptyset, \ulcorner\texttt{(EQUAL NIL (IF NIL Y Z))}\urcorner)$

   $=$

   $\{(\{\texttt{X}\}, \ulcorner\texttt{(EQUAL NIL Z)}\urcorner)\}$,

   which simplifies the ACL2 formula

   ```
   (EQUAL NIL (IF NIL Y Z)).
   ```

   to

   ```
   (EQUAL NIL Z).
   ```

As a second example,

$$\mathsf{Round}\,(\{\ulcorner\texttt{(EQUAL (EQUAL X NIL) NIL)}\urcorner\},\ulcorner\texttt{(IF (CONS A B) T NIL)}\urcorner)$$

$$=$$

$$\{(\{\ulcorner\texttt{(EQUAL (EQUAL X NIL) NIL)}\urcorner\},\ulcorner\texttt{T}\urcorner)\},$$

which simplifies the ACL2 formula

(EQUAL (EQUAL X NIL) NIL) $\rightarrow$ (IF (CONS A B) T NIL).

to

(EQUAL (EQUAL X NIL) NIL) $\rightarrow$ T.

3. $\{(C, \mathsf{subA}\,(X))\}$, if $(A_f = \ulcorner\texttt{EQUAL}\urcorner) \wedge (\mathsf{SymTermp}\,(A_1, A_2) \vee \mathsf{SymTermp}\,(A_2, A_1))$, where X is $\ulcorner\texttt{T}\urcorner$ or $\ulcorner\texttt{NIL}\urcorner$, if $A_1 = A_2$ or $A_1 \neq A_2$ respectively.

   Intuitively, this case simplifies `(EQUAL X X)` to `T` and `(EQUAL X` α`)` to `NIL`, if α is a `CONS` tree with `X` as one of its leaves. For example,

$$\mathsf{Round}\,(\emptyset,\ulcorner\texttt{(EQUAL (CONS (EQUAL X X) Y) Z)}\urcorner)$$

$$=$$

$$\{(\emptyset,\ulcorner\texttt{(EQUAL (CONS T Y) Z)}\urcorner)\},$$

   which simplifies the ACL2 formula

(EQUAL (CONS (EQUAL X X) Y) Z)

   to

(EQUAL (CONS T Y) Z).

   As a second example,

$\mathsf{Round}(\emptyset, \ulcorner\texttt{(IF (EQUAL X (CONS (CONS X Y) Z)) T NIL))}\urcorner)$

$=$

$\{(\emptyset, \ulcorner\texttt{(IF NIL T NIL)}\urcorner)\}$,

which simplifies the ACL2 formula

```
(IF (EQUAL X (CONS (CONS X Y) Z) T NIL))
```

to

```
(IF NIL T NIL).
```

4. $\{(C, \mathsf{subA}(X))\}$, if $A_f = \ulcorner\texttt{EQUAL}\urcorner \wedge A_1 \notin \mathbb{S} \wedge A_2 \notin \mathbb{S}$, where X is $\ulcorner\texttt{NIL}\urcorner$ or $\mathsf{MakeIf}(E_{car}, E_{cdr}, \ulcorner\texttt{NIL}\urcorner)$, if $\mathsf{GetConsp}(A_1) = \ulcorner\texttt{NIL}\urcorner \vee \mathsf{GetConsp}(A_2) = \ulcorner\texttt{NIL}\urcorner$ or $\mathsf{GetConsp}(A_1) \neq \ulcorner\texttt{NIL}\urcorner \wedge \mathsf{GetConsp}(A_2) \neq \ulcorner\texttt{NIL}\urcorner$ respectively; $E_{car} \triangleq \mathsf{MakeEq}(\mathsf{GetCar}(A_1), \mathsf{GetCar}(A_2))$; $E_{cdr} \triangleq \mathsf{MakeEq}(\mathsf{GetCdr}(A_1), \mathsf{GetCdr}(A_2))$.

   Intuitively, when the arguments to `EQUAL` are not symbols, they must be applications of `CONS`. This case then simplifies `(EQUAL (CONS X0 Y0) (CONS X1 Y1))` to

   ```
   (IF (EQUAL X0 X1) (EQUAL Y0 Y1) NIL)
   ```

   and `(EQUAL` α `(CONS Y Z))` to `NIL` if α is a non-tree constant. For example,

   $\mathsf{Round}(\emptyset, \ulcorner\texttt{(IF (EQUAL (CONS X Y) (CONS A B)) T NIL)}\urcorner)$

   $=$

   $\{(\emptyset, \ulcorner\texttt{(IF (IF (EQUAL X A) (EQUAL Y B) NIL) T NIL)}\urcorner)\}$,

   which simplifies the ACL2 formula

   ```
   (IF (EQUAL (CONS X Y) (CONS A B)) T NIL)
   ```

   to

```
(IF (IF (EQUAL X A) (EQUAL Y B) NIL) T NIL).
```

As a second example,

$\mathsf{Round}\,(\emptyset, \ulcorner\texttt{(CONS (EQUAL (CONS X Y) '4) A)}\urcorner)$

$=$

$\{(\emptyset, \ulcorner\texttt{(CONS NIL A)}\urcorner)\}$,

which simplifies the ACL2 formula

```
(CONS (EQUAL (CONS X Y) '4) A)
```

to

```
(CONS NIL A).
```

5. $\{(C \cup C_{NC}, \mathsf{subA}\,(\ulcorner\texttt{NIL}\urcorner)), (C/\sigma, \mathsf{subA}\,(X)/\sigma)\}$, if $(A_f \in \{\ulcorner\texttt{CAR}\urcorner, \ulcorner\texttt{CDR}\urcorner, \ulcorner\texttt{CONSP}\urcorner\})$; where $\sigma \triangleq [A_1 \mapsto \mathsf{MakeCons}\,(v_0, v_1)]$; $C_{NC} \triangleq \{\mathsf{MakeNotConsp}\,(A_1)\}$; and X is v_0, v_1, or `T`, if A_f is $\ulcorner\texttt{CAR}\urcorner$, $\ulcorner\texttt{CDR}\urcorner$, or $\ulcorner\texttt{CONSP}\urcorner$ respectively.

   Intuitively, this case breaks up a formula involving `(CAR X)`, `(CDR X)`, and `(CONSP X)` into two cases: one when `X` is a variable satisfying `(EQUAL (CONSP X) NIL)` and another `X` is a variable satisfying `(CONSP X)`. In the second case, `(CONS V0 V1)` is substituted for `X`, where `V0` and `V1` are variables not occurring in the original term. A simpler formula is produced in the sense that the conclusion has fewer applications of `CAR`, `CDR`, and `CONSP`. For example,

   $\mathsf{Round}\,(\emptyset, \ulcorner\texttt{(IF (CAR X) A (CDR X))}\urcorner)$

   $=$

   $\{(\{\ulcorner\texttt{(EQUAL (CONSP X) NIL)}\urcorner\}, \ulcorner\texttt{(IF NIL A (CDR X))}\urcorner),$
   $(\emptyset, \ulcorner\texttt{(IF V0 A V1)}\urcorner)\}$,

which simplifies the ACL2 formula

```
(IF (CAR X) A (CDR X))
```

to the conjunction of the validity of

```
((EQUAL (CONSP X) NIL) → (IF NIL A (CDR X)))
```

and

```
(IF V0 A V1).
```

As a second example,

Round ({⌜(EQUAL (EQUAL (CONS A X) Y) NIL)⌝}, ⌜(CAR X)⌝)

=

{({⌜(EQUAL (EQUAL (CONS A X) Y) NIL)⌝,

⌜(EQUAL (CONSP X) NIL)⌝},

⌜NIL⌝),

({⌜(EQUAL (EQUAL (CONS A (CONS V0 V1)) Y) NIL)⌝}, ⌜V0⌝)},

which simplifies the ACL2 formula

```
(EQUAL (EQUAL (CONS A X) Y) NIL) → (CAR X)
```

to the conjunction of the validity of

```
(EQUAL (EQUAL (CONS A X) Y) NIL) ∧ (EQUAL (CONSP X) NIL)
→
NIL
```

and

```
(EQUAL (EQUAL (CONS A (CONS V0 V1)) Y) NIL) → V0.
```

6. $\{(C \cup C_{NEQ}, \mathsf{subA}(A_2)), (C/\sigma, \mathsf{subA}(A_3)/\sigma)\}$, if $A_f = \ulcorner\texttt{IF}\urcorner$, where $\sigma \triangleq [A_1 \mapsto \ulcorner\texttt{NIL}\urcorner]$ and $C_{NEQ} \triangleq \{\mathsf{MakeNotEq}(\ulcorner\texttt{NIL}\urcorner, A_1)\}$.

   Intuitively, this case breaks `(IF X Y Z)` (where `X` is a variable) into two cases, one where `X` is true and the other where `X` is false. For example,

   $\mathsf{Round}(\emptyset, \ulcorner\texttt{(IF X Y Z)}\urcorner)$
   $=$
   $\{(\{\ulcorner\texttt{(EQUAL (EQUAL X NIL) NIL)}\urcorner\}, \ulcorner\texttt{Y}\urcorner),$
   $(\emptyset, \ulcorner\texttt{Z}\urcorner)\},$

   which simplifies the ACL2 formula

   ```
   (IF X Y Z)
   ```

   to the conjunction of the validity of

   ```
   (EQUAL (EQUAL X NIL) NIL) → Y
   ```

   and

   ```
   Z.
   ```

   As a second example,

   $\mathsf{Round}(\{\ulcorner\texttt{(EQUAL (EQUAL (CONS A X) Y) NIL)}\urcorner\},$
   $\ulcorner\texttt{(CAR (IF X Y (CONS X Y)))}\urcorner)$
   $=$
   $\{(\{\ulcorner\texttt{(EQUAL (EQUAL (CONS A X) Y) NIL)}\urcorner,$
   $\ulcorner\texttt{(EQUAL (EQUAL X NIL) NIL)}\urcorner\},$
   $\ulcorner\texttt{(CAR Y)}\urcorner),$
   $(\{\ulcorner\texttt{(EQUAL (EQUAL (CONS A NIL) Y) NIL)}\urcorner\},$
   $\ulcorner\texttt{(CAR (CONS NIL Y))}\urcorner)\},$

which simplifies the ACL2 formula

```
(EQUAL (EQUAL (CONS A X) Y) NIL)
→
(CAR (IF X Y (CONS X Y)))
```

to the conjunction of the validity of

```
(EQUAL (EQUAL (CONS A X) Y) NIL) ∧ (EQUAL (EQUAL X NIL) NIL)
→
(CAR Y)
```

and

```
(EQUAL (EQUAL (CONS A NIL) Y) NIL) → (CAR (CONS NIL Y)).
```

7. $\{(C \cup C_{NEQ}, \mathsf{subA}(\ulcorner\texttt{NIL}\urcorner)), (C/\sigma, \mathsf{subA}(\texttt{T})/\sigma)\}$, otherwise, where A_S is the argument of A that is a symbol ($A_S \in \mathbb{S}$), A_O is the other argument of A, $\sigma \triangleq [A_S \mapsto A_O]$, and $C_{NEQ} \triangleq \{\mathsf{MakeNotEq}(A_S, A_O)\}$.

   Note that at this point A_f must be `EQUAL` and either A_1 or A_2 is a symbol. Intuitively, this case breaks `(EQUAL X Y)` into two cases, one in which `(EQUAL X Y)` is false and the other in which it is true. In the false case, `(EQUAL (EQUAL X Y) NIL)` is added to the set of hypotheses. In the true case, `X` is replaced everywhere with `Y`. For example,

   $\mathsf{Round}(\emptyset, \ulcorner\texttt{(EQUAL X (CONS A B))}\urcorner)$
   $=$
   $\{(\{\ulcorner\texttt{(EQUAL (EQUAL X (CONS A B)) NIL)}\urcorner\}, \ulcorner\texttt{NIL}\urcorner),$
   $(\emptyset, \ulcorner\texttt{T}\urcorner)\},$

   which simplifies the ACL2 formula

```
(EQUAL X (CONS A B))
```

to the conjunction of the validity of

```
(EQUAL (EQUAL X (CONS A B)) NIL) → NIL
```

and

```
T.
```

As a second example,

```
Round({⌜(EQUAL (EQUAL (CONS A X) Y) NIL)⌝},
      ⌜(CONS (EQUAL X Y) X)⌝)
=
{({⌜(EQUAL (EQUAL (CONS A X) Y) NIL)⌝,
   ⌜(EQUAL NIL NIL)⌝},
  ⌜(CONS NIL X)⌝),
 ({⌜(EQUAL (EQUAL (CONS A Y) Y) NIL)⌝},
  ⌜(CONS T Y)⌝)},
```

which simplifies the ACL2 formula

```
(EQUAL (EQUAL (CONS A X) Y) NIL) → (CONS (EQUAL X Y) X)
```

to the conjunction of the validity of

```
(EQUAL (EQUAL (CONS A X) Y) NIL) ∧ (EQUAL (EQUAL X Y) NIL)
→
(CONS NIL X)
```

and

```
(EQUAL (EQUAL (CONS A Y) Y) NIL) → (CONS T Y).
```

Thus through a sequence of calls Round can reduce a set of formulas represented as pairs (C, E), where $C \subset \mathbb{E}_{NT}$ and $E \in \mathbb{E}_{SC}$, into simplified pairs (C', E'), where $C \subset \mathbb{E}_{NT}$ and $E \in \mathbb{E}_T$. This sequence is formalized in with following function.

Given $S \in (\mathcal{P}(\mathbb{E}_{NT}) \times \mathbb{E}_{SC})$, define

$$\mathsf{SimplifyCore}(S) \triangleq \begin{cases} S, & \text{if } \forall (C, E) \in S : E \in \mathbb{E}_T \\ \mathsf{SimplifyCore}((S \setminus \{(X_C, X_E)\}) \cup \mathsf{Round}(X_C, X_E)), & \text{otherwise.} \end{cases}$$

where (X_C, X_E) is some element of S such that $X_E \notin \mathbb{E}_T$ (to be precise we can choose the element such that $\mathsf{MakeSetImp}(C, E)$ is lexicographically smallest).

The $\mathsf{SimplifyCore}$ function terminates by Theorem 6 and by Theorem 7 it produces a set of formula that are valid if and only if its input formula is valid.

The decision procedure can then be completed with a procedure that checks the validity of each formula returned by $\mathsf{SimplifyCore}$.

Given a set $C \subset \mathbb{E}_{NT}$ and a term $E \in \mathbb{E}_T$, define

$$\mathsf{ValidFFp}(C, E) \triangleq \begin{cases} \mathsf{EvPrfp}(H_{GZ}, \mathsf{MakeSetImp}(C, E)/\sigma) = \ulcorner \mathtt{NIL} \urcorner, & \text{if } E = \ulcorner \mathtt{NIL} \urcorner \\ \mathsf{ValidFFp}(C/[E \mapsto \ulcorner \mathtt{NIL} \urcorner], \ulcorner \mathtt{NIL} \urcorner), & \text{if } E \in \mathbb{S} \\ \textbf{true}, & \text{otherwise.} \end{cases}$$

where σ is the substitution that maps each v_i to the natural number $i + M$, $v_1 \in \mathbb{S}$ through $v_n \in \mathbb{S}$ are the n variables that occur in $\mathsf{MakeSetImp}(C_G, E)$, and M is a natural number at least as large as any constant that occurs in $\mathsf{MakeSetImp}(C, E)$.

From Theorem 8 it follows that $\mathsf{ValidFFp}(C, E)$ is true if and only if $\mathsf{MakeSetImp}(C, E)$ is a valid ACL2 formula. Thus, $F \in \mathbb{E}_{SC}$ is valid if and only if $\mathsf{ValidFFp}(C, E)$ is true for all $(C, E) \in \mathsf{SimplifyCore}(\emptyset, F)$. We therefore have a decision procedure for any formula that contains only SULFA core primitives.

Intuitively, ValidFFp (C, E) determines whether MakeSetImp (C, E) is valid by breaking up the problem into three cases: 1) E is NIL, 2) E is a symbol, 3) E is an application of CONS (the only other possibility since $E \in \mathbb{E}_T$). Case 3 is trivially valid, since

```
(CONS X Y) ≠ NIL.
```

Case 2 can be reduced to case 1, since if the formula has a counterexample, that counterexample occurs when E is NIL. For example,

```
(EQUAL (EQUAL (CONS A Y) Y) NIL) → Y.
```

is reduced to:

```
(EQUAL (EQUAL (CONS A NIL) NIL) NIL) → NIL.
```

Case 1 then determines whether there is a substitution σ mapping variables to values such that every hypothesis is true. Since each hypothesis is the negation of a CONSP application or the negation of an EQUAL, this can be determined easily.

- Every CONSP negation that can be true will be true when all variables are mapped to non-tree values. For example, (EQUAL (CONSP X) NIL) is true when X is a non-tree value, (EQUAL (CONSP (CONS X Y)) NIL) must be false, and (EQUAL (CONSP '8) NIL) must be true.

- Every EQUAL negation that can be true will be true when all variables are given values distinct from other variables and from values occurring in the original term. For example, (EQUAL (EQUAL X Y) NIL) is true when X and Y are different and

  ```
  (EQUAL (EQUAL (CONS X Y) (CONS A B)) NIL)
  ```

  is true when X, Y, A, and B are all different from each other. Similarly,

  ```
  (EQUAL (EQUAL (CONS X '4) (CONS A '4)) NIL)
  ```

is true when `X` and `A` are different. On the other hand,

```
(EQUAL (EQUAL (CONS X Y) (CONS X Y)) NIL)
```

will be false regardless of which values are given to `X` and `Y`.

Thus, given a $C \subset \mathbb{E}_{NT}$, a simple way of determining whether a formula MakeSetImp$(C, \ulcorner\texttt{NIL}\urcorner)$ is valid is to see if MakeSetImp$(C/\sigma, \ulcorner\texttt{NIL}\urcorner)$ evaluates to `NIL` when σ is a substitution mapping the variables in C to non-tree constants (such as natural numbers) different from one another and from the constants in C. For example, the validity of

```
(EQUAL (EQUAL (CONS X Y) (CONS X Y)) NIL)
∧
(EQUAL (CONSP X) NIL)
∧
(EQUAL (CONSP Y) NIL)
→
NIL
```

is determined by evaluating

```
(EQUAL (EQUAL (CONS 1 2) (CONS 1 2)) NIL)
∧
(EQUAL (CONSP 1) NIL)
∧
(EQUAL (CONSP 2) NIL)
→
NIL.
```

On the other hand, a counterexample to

```
(EQUAL (EQUAL (CONS X 4) (CONS X Y)) NIL)
∧
(EQUAL (CONSP X) NIL)
∧
(EQUAL (CONSP Y) NIL)
→
NIL
```

is determined by evaluating

```
(EQUAL (EQUAL (CONS 5 4) (CONS 5 6)) NIL)
∧
(EQUAL (CONSP 5) NIL)
∧
(EQUAL (CONSP 6) NIL)
→
NIL.
```

6.4.1 Correctness

Given a set of symbols S and term E, define $\mathsf{Count}(S, E)$ to be the number of times that symbols in S appear as functions in E. Furthermore, define $\mathsf{VarCount}(E)$ to be the number of distinct variable symbols that occur in E. For example,

$$\mathsf{Count}(\{\ulcorner\texttt{CAR}\urcorner, \ulcorner\texttt{CDR}\urcorner\}, \ulcorner\texttt{(CONSP (CAR (CDR (CDR X))))}\urcorner) = 3,$$

$$\mathsf{Count}(\{\ulcorner\texttt{CDR}\urcorner\}, \ulcorner\texttt{(CONSP (CAR (CDR (CDR X))))}\urcorner) = 2,$$

$$\mathsf{Count}(\{\ulcorner\texttt{CONSP}\urcorner\}, \ulcorner\texttt{(CONSP (CAR (CDR (CDR X))))}\urcorner) = 1,$$

$$\mathsf{VarCount}(\ulcorner\texttt{(CONS X (CONS X Z))}\urcorner) = 3, \text{ and}$$

$$\mathsf{VarCount}(\ulcorner\texttt{(CONS X (CONS X Y))}\urcorner) = 2.$$

Also, given $E \in \mathbb{E}$, define

$$\begin{aligned}\mathsf{RoundOrd}(E) \triangleq\ & \omega^3 \times \mathsf{Count}(\{\ulcorner\mathtt{CAR}\urcorner, \ulcorner\mathtt{CDR}\urcorner, \ulcorner\mathtt{CONSP}\urcorner\}, E) + \\ & \omega^2 \times \mathsf{VarCount}(E) + \\ & \omega \times \mathsf{Count}(\{\ulcorner\mathtt{CONS}\urcorner\}, E) + \mathsf{Count}(\{\ulcorner\mathtt{EQUAL}\urcorner, \ulcorner\mathtt{IF}\urcorner\}, E).\end{aligned}$$

Lemma 7. *Given $C \subset \mathbb{E}_{\mathrm{NT}}$ and $E \in \mathbb{E}_{\mathrm{SC}}$ such that $E \notin \mathbb{E}_{\mathrm{T}}$, let $S \triangleq \mathsf{Round}(C, E)$. Then, for each element $(S_C, S_E) \in S$ it follows that $S_C \in E_{\mathrm{NT}}$, $S_E \in \mathbb{E}_{\mathrm{SC}}$, and $\mathsf{RoundOrd}(S_E) <$ $\mathsf{RoundOrd}(E)$.*

Proof Sketch: Note that from the definition of NonConsFA it follows that A is a subterm of E such that $\mathsf{Fn}(A) \in \{\mathtt{CAR}, \mathtt{CDR}, \mathtt{CONSP}, \mathtt{IF}, \mathtt{EQUAL}\}$.

For all $(S_C, S_E) \in S$, it follows trivially from the definition of Round that $S_C \in E_{NT}$ and $S_E \in mathbbE_{SC}$. To prove that $\mathsf{RoundOrd}(S_E) < \mathsf{RoundOrd}(E)$, we consider each case within the definition of Round.

- In cases 1 through 3, each conclusion S_E returned by $\mathsf{Round}(C, E)$ is trivially smaller than $\mathsf{RoundOrd}(E)$, since S_E is formed by replacing A with an argument of A.

- In case 4, either A_1 or A_2 is an application of $\ulcorner\mathtt{CONS}\urcorner$. Thus, by the definition of GetCar and GetCdr, A contains at least one fewer application of $\ulcorner\mathtt{CONS}\urcorner$. Furthermore, since $A_1 \notin \mathbb{S}$ and $A_2 \notin \mathbb{S}$ no new applications of $\ulcorner\mathtt{CAR}\urcorner$ or $\ulcorner\mathtt{CDR}\urcorner$ are added.

- In case 5, the substitution σ may add applications of $\ulcorner\mathtt{CONS}\urcorner$, but not applications of any other functions. Therefore, each conclusion S_E returned by $\mathsf{Round}(C, E)$ has one fewer application of $\ulcorner\mathtt{CAR}\urcorner$, $\ulcorner\mathtt{CDR}\urcorner$, or $\ulcorner\mathtt{CONSP}\urcorner$ than E.

- In case 6, both $\mathsf{subA}(A_2)$ and $\mathsf{subA}(A_3)/\sigma$ have one fewer application of $\ulcorner\mathtt{IF}\urcorner$, since σ just maps variables to $\ulcorner\mathtt{NIL}\urcorner$.

- In case 7, $\mathsf{subA}(\ulcorner\mathtt{NIL}\urcorner)$ has at least one fewer application of EQUAL then E and $\mathsf{subA}(\mathtt{T})/\sigma$, while it may have more CONS applications than E, has fewer variables.

□

Theorem 6. *For all* $S \in (\mathcal{P}(\mathbb{E}_{\text{NT}}) \times \mathbb{E})$ $\mathsf{SimplifyCore}\,(S)$ *terminates.*

Proof Sketch: From Lemma 7 it follows that the ordinal $\sum_{(C,E)\in S} \omega^{\mathsf{RoundOrd}\,(E)}$ decreases on each recursive call in the definition of $\mathsf{SimplifyCore}\,(S)$. □

Lemma 8. *Let* $X \in \mathbb{E}_{\text{T}}$ *and* $Y \in \mathbb{E}_{\text{T}}$ *satisfy* $\mathsf{SymTermp}\,(X, Y)$ *and* $X \neq Y$*. Then,* $\mathsf{MakeEq}\,(\mathsf{MakeEq}\,(X, Y), \ulcorner\mathtt{NIL}\urcorner)$ *is a valid ACL2 formula in* H_{GZ}*.*

Proof Sketch: Trivial by induction on $|Y|$ and ACL2's axioms defining the primitive `ACL2-COUNT`, from which it follows that:

```
(AND (INTEGERP (ACL2-COUNT X))

     (<= 0 (ACL2-COUNT X))

     (IMPLIES (CONSP X)
              (< (ACL2-COUNT (CAR X)) (ACL2-COUNT X)))

     (IMPLIES (CONSP X)
              (< (ACL2-COUNT (CDR X)) (ACL2-COUNT X)))).
```

□

Lemma 9. *Given* $E \in \mathbb{E}$*,* $X \in \mathbb{S}$*, an ACL2 history* H*, let* $v_0 \in \mathbb{S}$ *and* $v_0 \in \mathbb{S}$ *be symbols not occurring in* E *such that* $v_0 \neq v_1$*. Then,* $\mathsf{MakeImp}\,(\mathsf{MakeConsp}\,(X), E)$ *is valid in* H *if and only if* $E/[X \mapsto \mathsf{MakeCons}\,(v_0, v_1)]$ *is valid in* H*.*

Proof Sketch: If $\mathsf{MakeImp}\,(\mathsf{MakeConsp}\,(X), E)$ is valid, then $E/[X \mapsto \mathsf{MakeCons}\,(v_0, v_1)]$ follows by instantiation.

If $F = E/[X \mapsto \mathsf{MakeCons}\,(v_0, v_1)]$ is valid, then

$$F/[v_0 \mapsto \mathsf{MakeCar}\,(X), v_1 \mapsto \mathsf{MakeCdr}\,(X)]$$

is valid. Therefore, $\mathsf{MakeImp}(\mathsf{MakeConsp}(X), E)$ follows from the validity of

```
(IMPLIES (CONSP X) (EQUAL (CONS (CAR X) (CDR X)) X)).
```

□

Lemma 10. *Given $E \in \mathbb{E}$, $X \in \mathbb{S}$, $Y \in \mathbb{E}$, and an ACL2 history H, the formula* $\mathsf{MakeImp}(\mathsf{MakeEq}(X, Y), E)$ *is valid in H if and only if $E/[X \mapsto Y]$ is valid in H.*

Proof Sketch: Trivial from the inference rules of the ACL2 logic. □

Theorem 7. *Given a finite set $C \subset \mathbb{E}_{\mathrm{NT}}$ and an ACL2 term $E \in \mathbb{E}_{\mathrm{SC}}$ such that $E \notin \mathbb{E}_{\mathrm{T}}$, let $S \triangleq \mathsf{Round}(C, E)$. Then, for any ACL2 history H,* $\mathsf{MakeSetImp}(C, E)$ *is a valid ACL2 formula in H if and only if for all $(S_C, S_E) \in S$,* $\mathsf{MakeSetImp}(S_C, S_E)$ *is a valid ACL2 formula.*

Proof Sketch: We prove the theorem for each of the cases in the definition of Round, and using the same definitions of A_1, A_2, and A_f as in the definition Round. Note that, since $A_1 \in E_T$, $\mathsf{Fnp}(A_1) \to \mathsf{Fn}(A_1) = \ulcorner\texttt{CONS}\urcorner$ and the same is true for A_2.

1. Follows directly from the validity of

   `(EQUAL (CAR (CONS X Y)) X)`,

   `(EQUAL (CDR (CONS X Y)) X)`, and

   `(EQUAL (CONSP (CONS X Y)) T)`.

2. Either A_1 is a constant or an application of `CONS`. Therefore, the equivalence of $\mathsf{MakeSetImp}(C, \mathsf{subA}(X))$ and $\mathsf{MakeSetImp}(C, E)$ follows from the validity of

   `(IMPLIES (NOT (EQUAL X NIL)) (EQUAL (IF X Y Z) Y))`,

   `(EQUAL (IF NIL Y Z) Z))`, and

   `(EQUAL (IF (CONS A B) Y Z) Y)`.

3. If $A_1 = A_2$, then (EQUAL (EQUAL X X) T). Otherwise, the theorem follows from Lemma 8.

4. One of A_1 and A_2 is an application of CONS and the other is either a constant or an application of CONS. Therefore, if either GetConsp$(A_1) = \ulcorner$NIL$\urcorner$ or GetConsp$(A_2) = \ulcorner$NIL$\urcorner$ then the theorem follows from the validity of

   (IMPLIES (NOT (CONSP X)) (EQUAL (CAR X) NIL)),

   (IMPLIES (NOT (CONSP X)) (EQUAL (CDR X) NIL)), and

   (IMPLIES (NOT (CONSP X)) (EQUAL (CONSP X) NIL)).

   Otherwise, the theorem follows from

```
(IMPLIES (AND (CONSP A1) (CONSP A2))
         (EQUAL (EQUAL A1 A2)
                (IF (EQUAL (CAR A1) (CAR A2))
                    (EQUAL (CDR A1) (CDR A2))
                  NIL))).
```

5. Note that $A_1 \in \mathbb{S}$, by the negation of the conditions of case 1. Furthermore,

$$\mathsf{MakeSetImp}\,(C \cup C_{NC}, \mathsf{subA}\,(\mathtt{NIL}))$$

   is equal to

$$\mathsf{MakeSetImp}\,(C \cup C_{NC}, E),$$

   by the same reasoning as in case 1. Thus, we need only prove that

$$\mathsf{MakeImp}\,(\mathsf{MakeConsp}\,(A_1, \mathsf{MakeSetImp}\,(C, E))$$

   is valid if and only if

$$\mathsf{MakeSetImp}\,(C/\sigma, subA(X)/\sigma)$$

   is valid. This follows from Lemma 9 and the reasoning in case 1.

6. Note again that $A_1 \in \mathbb{S}$. From the validity of

```
(IMPLIES (NOT (EQUAL A1 NIL)) (EQUAL (IF A1 A2 A3) A2)),
```

   it follows that

$$\mathsf{MakeSetImp}\,(C \cup C_{NEQ}, \mathsf{subA}\,(A_2))$$

   is equal to

$$\mathsf{MakeSetImp}\,(C \cup C_{NEQ}, E).$$

   Therefore, we need only prove that

$$\mathsf{MakeImp}\,(\mathsf{MakeEq}\,(\ulcorner\mathtt{NIL}\urcorner, A_1), \mathsf{MakeSetImp}\,(C, E))$$

   is valid if and only if

$$\mathsf{MakeSetImp}\,(C/\sigma, subA(A_3)/\sigma).$$

   This follows from Lemma 10 and the validity of

```
(IMPLIES (EQUAL A1 NIL) (EQUAL (IF A1 A2 A3) A3)).
```

7. Note that $\mathsf{Fn}\,(A) = \ulcorner\mathtt{EQUAL}\urcorner$, and either $A_1 \in \mathbb{S}$ or $A_2 \in \mathbb{S}$. Therefore,

$$\mathsf{MakeSetImp}\,(C \cup C_{NEQ}, \mathsf{subA}\,(\ulcorner\mathtt{NIL}\urcorner))$$

   equals

$$\mathsf{MakeSetImp}\,(C \cup C_{NEQ}, E).$$

   Thus, we need only prove that

$$\mathsf{MakeImp}\,(\mathsf{MakeEq}\,(A_S, A_O), \mathsf{MakeSetImp}\,(C, E))$$

   is valid if and only if

$$\mathsf{MakeSetImp}\,(C/\sigma, subA(\ulcorner\mathtt{T}\urcorner)/\sigma),$$

   which follows from Lemma 10.

□

Lemma 11. *Given a term* $X \in \mathbb{E}_\mathrm{T}$ *and* $Y \in \mathbb{E}_\mathrm{T}$*, let* $A \triangleq \mathsf{MakeNotEq}\,(X, Y)$*; let* $\sigma \subset (\mathbb{S}\times\mathbb{E})$ *be a substitution mapping all the variables in A to unique natural numbers that do not appear in A; Then, if* $\mathsf{EvPrfp}\,(H_{\mathrm{GZ}}, A/\sigma)$*, then A is a valid ACL2 formula.*

Proof Sketch: First note that since σ maps all variables to constants and all functions in A are ACL2 primitives, $\mathsf{EvPrfp}\,(H_{GZ}, A/\sigma)$ is equivalent to $\mathsf{EvPrf}\,(H_{GZ}, A/\sigma) \neq \ulcorner\mathtt{NIL}\urcorner$.

Induct on the number of `CONS` applications in A.

- If there are no `CONS` applications, then X and Y are each either a constant or a variable. Since σ never maps a variable to a constant that occurs in A and never maps two variables to the same constant, X and Y must be constant. In that case, the formula is clearly valid, since $A = A/\sigma$.

- If either X or Y is a `CONS` application, but not both, then the theorem follows from contradiction. $\mathsf{EvPrf}\,(H, A\sigma) = \ulcorner\mathtt{T}\urcorner$, since no natural number can equal the result of a `CONS` application.

- Otherwise, both X and Y are `CONS` applications. Thus, the theorem follows from induction, and the validity of

```
(IMPLIES (OR (NOT (EQUAL X0 Y0)) (NOT (EQUAL X1 Y1)))
         (NOT (EQUAL (CONS X0 X1) (CONS Y0 Y1)))).
```

□

Lemma 12. *Given an ACL2 history H,* $C \subset \mathbb{E}_{\mathrm{NT}}$*, let* $E \triangleq \mathsf{MakeSetImp}\,(C, \ulcorner\mathtt{NIL}\urcorner)$*. Furthermore, let* $v_1 \in \mathbb{S}$ *through* $v_n \in \mathbb{S}$ *denote the n variables that occur in E, let M be a natural number at least as large as the largest natural number constant that appears in E, and let* σ *be the substitution that maps each* v_i *to the natural number* $i + M$*. Then, E is a valid formula in H if and only if* $\mathsf{EvPrfp}\,(H_{\mathrm{GZ}}, E/\sigma)$*.*

Proof Sketch: First note that since σ maps all variables to constants and all functions in A are ACL2 primitives, $\mathsf{EvPrfp}\,(H_{GZ}, A/\sigma)$ is equivalent to $\mathsf{EvPrf}\,(H_{GZ}, A/\sigma) \neq \ulcorner\texttt{NIL}\urcorner$.

The validity of E implies the validity of E/σ, by instantiation. Furthermore, since σ maps all variables in E to constants and $E \in \mathbb{E}_{CS}$, the validity of E/σ implies that $\mathsf{EvPrf}\,(H_{GZ}, E/\sigma)$ is a constant and not equal to $\ulcorner\texttt{NIL}\urcorner$.

We prove that the validity of E follows from the validity of E/σ by contradiction. Assume that E/σ is valid, since $\mathsf{EvPrf}\,(H_{GZ}, \mathsf{MakeSetImp}\,(C, \ulcorner\texttt{NIL}\urcorner/\sigma)) \neq \ulcorner\texttt{NIL}\urcorner$, there must be some $A \in C$ such that $\mathsf{EvPrf}\,(H_{GZ}, A/\sigma) = \ulcorner\texttt{NIL}\urcorner$.

- First, assume $A = \mathsf{MakeNotConsp}\,(A_x)$ for some $A_x \in \mathbb{E}_T$. If A_x is constant, then $A_x = \ulcorner\texttt{NIL}\urcorner$, and therefore E is valid. Otherwise, A_x must be an application of `CONS`, since $A_x \in \mathbb{S}$ implies $\mathsf{EvPrf}\,(H_{GZ}, A/\sigma) = $`T`. Thus, the validity of E follows from

```
(EQUAL (NOT (CONSP (CONS X0 X1))) NIL).
```

- Otherwise, the validity of E follows from Lemma 11.

□

Theorem 8. *Let H be an ACL2 history, $C \subset \mathbb{E}_{\mathrm{NT}}$, and $E \subset \mathbb{E}_H$. Then,* $\mathsf{MakeSetImp}\,(C), E)$ *is a valid formula in H if and only if* $\mathsf{ValidFFp}\,(C, E)$.

Proof Sketch: If $E = \ulcorner\texttt{NIL}\urcorner$, then the theorem follows directly from Lemma 12. If $E \in \mathbb{S}$, then the theorem follows from Lemma 10 and Lemma 12. Otherwise, the theorem follows from the validity of $\ulcorner$`(NOT (EQUAL (CONS X Y) NIL))`$\urcorner$. □

6.5 Counterexample Generation

It is possible to construct a counterexample to any invalid SULFA formula. In this context a counterexample replaces uninterpreted functions with concrete functions and variables with values such that a formula evaluates to `NIL`.

From the previous section, recall that if $F \in \mathbb{E}_{SC}$ is an invalid formula, then there exists a $(C, E) \in \mathsf{SimplifyCore}(\{(\emptyset, F)\})$ such that $\neg\mathsf{ValidFFp}(C, E)$. Note that when $\mathsf{ValidFFp}(C, E)$ is false $\mathsf{ValidFFp}$ creates a counterexample to $\mathsf{MakeSetImp}(C, E)$. By following the substitutions leading to (C, E), it is possible to construct a counterexample to the original formula F.

Furthermore, by inverting the substitutions used to remove uninterpreted functions in Section 6.3, it is possible to determine concrete functions for each uninterpreted function such that the formula evaluates to `NIL`. Furthermore, the unrolling algorithm in Section 6.2 produces an equivalent formula to the original SULFA formula with no new variables and only provably irrelevant variables removed. Therefore, a counterexample to the unrolled formula is also a counterexample to the original SULFA formula.

6.6 Summary

Chapter 5 defines SULFA, a decidable subclass of ACL2 formulas, involving unbounded tree structures, if-then-else, equality, uninterpreted functions, and a definition principle for extending it with user-defined functions. SULFA intuitively resembles the theory of list structures with uninterpreted functions, which is known to be decidable. The axioms of ACL2 are somewhat different from the axioms of the theory of list structures (more can be proven from the axioms of ACL2). Also, ACL2 contains a more sophisticated array of constants than the traditional theory of list structures, such as strings, symbols, and complex numbers. Thus the decidability of SULFA does not directly follow from previous proofs of decidability of similar theories.

This chapter shows that SULFA is, in fact, a *decidable* subclass, by presenting a procedure that determines whether any SULFA formula is valid. The procedure first unrolls user-defined functions, then removes uninterpreted functions, and finally solves ACL2 formulas involving SULFA core primitives. A proof sketch of the correctness of each step of the procedure is provided.

The decision procedure presented here is intended to be as simple as possible, rather than efficient. Chapter 7 describes a more efficient SULFA solver, though with a less rigorous correctness justification. Chapter 8 and 9 then apply the SULFA solver to hardware verification problems.

Chapter 7

Developing a SAT-Based SULFA Solver

7.1 Introduction

Chapter 5 defines SULFA, a decidable subclass of ACL2 formulas made up of the tree data structure, equality, if-then-else, uninterpreted functions, and (unrollable) user-defined functions. Chapter 6 shows that SULFA is a decidable subclass by presenting a procedure that determines whether any SULFA formula is valid. In order to apply a SULFA solver to problems of practical interest, however, a more efficient procedure than this is required.

This chapter outlines an algorithm for solving SULFA formulas, based on SAT solvers, which Chapters 8 and 9 apply to non-trivial problems from hardware verification. The intuition behind the SAT-Based SULFA Solver is to find a finite set of equality and `CONSP` predicates that are relevant to a given formula. Then, the relationship between the predicates is codified as a satisfiability problem in Boolean Conjunctive Normal Form (CNF), and passed to a SAT solver. The algorithm is described in more detail in our workshop paper [32] and an implementation of it, along with its source code, is available with the ACL2 distribution [37].

Section 7.2 begins by showing how nested if-then-else terms can be translated into CNF. Section 7.3 then defines Boolean SULFA predicates, which form the basis for our SAT-based algorithm. Section 7.4 next presents an overview of the algorithm used to translate SULFA formulas into CNF and the intuition used to optimize it. Section 7.5 illustrates how our algorithm translates a few SULFA formulas into CNF.

Note that this chapter reuses some of the terminology from Chapter 5, as well as the simplifications to the ACL2 logic described in Section 5.3. While the actual implementation supports mutual recursion and lambda functions, these features are not addressed in this chapter.

7.2 Translating Nested If Terms to CNF

We begin by showing how nested `IF` terms are translated into CNF, which forms the basis for our full SULFA to CNF translation algorithm.

Define a *normalized* `IF` *term* as an `IF` term where every condition (the first argument of `IF`) is a symbol. The validity of normalized `IF` terms can be easily translated into the satisfiability of a Boolean CNF formula. For example, the universally-quantified ACL2 formula

```
α:  (IF A (IF B C D) (IF E F G))
```

translates to the following (negated) existentially-quantified CNF formula:

$$\beta : (\neg a \vee \neg b \vee \neg c) \wedge (\neg a \vee b \vee \neg d) \wedge (a \vee \neg e \vee \neg f) \wedge (a \vee e \vee \neg g)$$

Note that α is valid if and only if β is unsatisfiable. Thus a SAT solver that checks whether β is satisfiable also solves whether α is valid.

A simple algorithm for translating a normalized `IF` term to CNF creates one clause for each possible result of the `IF` condition. This has $O(N^2)$ worst case complexity, where N is the size of the normalized `IF` term, since the generated CNF formula has N clauses and each clause has at most N literals.

The straightforward method for normalizing a nested `IF` term, however, requires exponential time, because the branches of the `IF` must be duplicated during normalization. As Tseitin showed, however, this problem can be alleviated by introducing new variables into the CNF term[89]. To formalize this concept, we introduce the notion of an ACL2 term in *CNF-ready* form, defined by the following grammar:

CNF-ready ::= *norm-if* | `(IMPLIES (AND` *norm-iff*`)` *norm-if*`)` |
`(IMPLIES (AND` *norm-iff-list*`)` *norm-if*`)`
norm-iff-list ::= *norm-iff* | *norm-iff norm-iff-list*
norm-iff ::= `(IFF` **symbol** *norm-if*`)` | *norm-if*
norm-if ::= **symbol** | `T` | `NIL` | `(IF` **symbol** *norm-if norm-if*`)`

where **symbol** is an arbitrary symbol. For example, the nested `IF` term `(IF (IF A B C) D E)` is valid if and only if the CNF-ready formula

```
(IMPLIES (IFF V (IF A B C)) (IF V D E))
```

is valid.

The simple algorithm for translating normalized `IF` terms into CNF can be extended into an algorithm to translate CNF-ready formulas. For example, the formula `(IMPLIES (IFF V (IF A B C)) (IF V D E))` is translated into:

$$(\neg v \vee \neg a \vee b) \wedge (\neg v \vee a \vee c) \wedge (v \vee \neg a \vee \neg b) \wedge$$
$$(v \vee a \vee \neg c) \wedge (\neg v \vee \neg d) \wedge (v \vee \neg e)$$

where $v \triangleq (\ulcorner\texttt{V}\urcorner \neq \ulcorner\texttt{NIL}\urcorner)$, $a \triangleq (\ulcorner\texttt{A}\urcorner \neq \ulcorner\texttt{NIL}\urcorner)$, $b \triangleq (\ulcorner\texttt{B}\urcorner \neq \ulcorner\texttt{NIL}\urcorner)$, $c \triangleq (\ulcorner\texttt{C}\urcorner \neq \ulcorner\texttt{NIL}\urcorner)$, $d \triangleq (\ulcorner\texttt{D}\urcorner \neq \ulcorner\texttt{NIL}\urcorner)$, and $e \triangleq (\ulcorner\texttt{E}\urcorner \neq \ulcorner\texttt{NIL}\urcorner)$. The above formula is unsatisfiable if and only if

```
(IMPLIES (IFF V (IF A B C)) (IF V D E))
```

is valid.

The algorithm used above has $O(N^2)$ complexity. Each hypothesis in the CNF-ready formula is translated into CNF independently of the conclusion and other hypotheses, since the disjunction in the ACL2 formula becomes a conjunction in the CNF formula. Furthermore, each `IFF` hypothesis translates into CNF by translating its normalized `IF` term twice, once for the true case and once for the false case. Therefore, the time required is $O(m_0^2 + m_1^2 ... + m_n^2)$, where m_i is the size of the ith `IF` term. Given a CNF-ready formula of size N, the complexity of the translation is $O(N^2)$, since $(m_0 + m_1 ... + m_n)^2$ is greater than $m_0^2 + m_1^2 ... + m_n^2$.

The translation of a CNF-ready formula into CNF is linear if the size of each `IFF` is bounded. Therefore, any nested `IF` term can be translated into CNF in linear time, if variables are created for every `IF` term. For example,

```
(IF A (IF B C D) (IF (IF E F G) H J))
```

can be translated to

```
(IMPLIES (AND (IFF V0 (IF B C D))
              (IFF V1 (IF E F G))
              (IFF V2 (IF V1 H J)))
         (IF A V0 V2)).
```

The above CNF-ready formula is translated to CNF in linear time, since each hypothesis and conclusion contains only a single `IF` term.

7.3 Boolean SULFA Predicates

Note that in the previous section each Boolean variable in the CNF formula represents a predicate on ACL2 terms of the form $\alpha \neq$ `NIL` for some ACL2 variable α. The problem is then decidable since only a finite number of such Boolean predicates are relevant to the nested `IF` term. Similarly every SULFA formula can be reduced to a finite number of Boolean predicates.

Define a *SULFA Boolean predicate* as an ACL2 term in the following grammar:

SBP ::= `T` | `NIL` | `(CONSP` *ccExpr*`)` | `(EQUAL` *ccExpr* *ccExpr*`)` |

ccExpr ::= **symbol** | `(CAR` *ccExpr*`)` | `(CDR` *ccExpr*`)`

where **symbol** is an arbitrary symbol. For example, `(CONSP (CDR (CAR X)))` and `(EQUAL (CAR A) (CAR (CDR B)))` are SULFA Boolean predicates.

Furthermore, a SULFA nested `IF` predicate is defined by the grammar:

ifpred ::= *SBP* | `(IF` *ifpred ifpred ifpred*`)`

where *SBP* is an arbitrary SULFA Boolean predicate. For example,

```
(IF (EQUAL (CAR X) 4) (CONSP Y) (EQUAL (CDR X) NIL))
```

is a SULFA nested `IF` predicate.

Given a SULFA Boolean predicate X and a substitution σ mapping variables to terms in $\mathbb{E}_{\text{SC}}$ (terms containing only SULFA core primitives), X/σ can be reduced to a SULFA nested `IF` predicate by successive applications of the rules in Figure 7.1. Furthermore, any ACL2 formula F is equivalent to $\ulcorner$`(EQUAL X NIL)`$\urcorner/[$`X`$\mapsto F]$. Therefore, since any SULFA formula can be reduced to a formula involving only SULFA core primitives, any SULFA formula can be reduced to a SULFA nested `IF` predicate. For example,

```
(IMPLIES (EQUAL (CAR X) A)
         (EQUAL (CAR (IF (CONSP X) X (CONS A B)))
                A))
```

is a SULFA formula that is equivalent to:

```
(IF (EQUAL (CAR X) A)
    (EQUAL (CAR (IF (CONSP X) X (CONS A B)))
           A)
  T).
```

```
1. (CAR (IF A B C)) ⇒ (IF A (CAR B) (CAR C))

2. (CDR (IF A B C)) ⇒ (IF A (CDR B) (CDR C))

3. (CAR (EQUAL A B)) ⇒ (CDR (EQUAL A B)) ⇒ NIL

4. (CAR (CONS A B)) ⇒ A

5. (CDR (CONS A B)) ⇒ B

6. (CAR (CONSP A)) ⇒ (CDR (CONSP A)) ⇒ NIL

7. (EQUAL (IF A B C) D) ⇒
   (IF (EQUAL A NIL) (EQUAL C D) (EQUAL B D))

8. (EQUAL (CONS A B) C) ⇒

   (IF (CONSP C)
        (IF (EQUAL (CAR C) A) (EQUAL (CDR C) B) NIL)
        NIL)

9. (EQUAL (CONSP A) B) ⇒ (IF (CONSP A) (EQUAL B T) (EQUAL B NIL))

10. (EQUAL (EQUAL A B) C) ⇒
    (IF (EQUAL A B) (EQUAL C T) (EQUAL C NIL))

11. (CONSP (IF A B C)) ⇒ (IF (EQUAL A NIL) (CONSP C) (CONSP B))

12. (CONSP (CONS A B)) ⇒ T

13. (CONSP (CONSP A)) ⇒ (CONSP (EQUAL A B)) ⇒ NIL
```

Figure 7.1: Let X be a nested IF predicate, and σ a substitution mapping symbols to terms in $\mathbb{E}_{SC}$ (terms that include only SULFA core primitives). Then, the rewrite rules above can be used to reduce X/σ into a nested IF predicate.

The above formula is equivalent to the following instantiation of ⌜`(EQUAL X NIL)`⌝:

```
(NOT (EQUAL
      (IF (EQUAL (CAR X) A)
          (EQUAL (CAR (IF (CONSP X) X (CONS A B)))
               A)
        T)
      NIL)),
```

where `(NOT E)` is an abbreviation of `(IF E NIL T)`. By repeatedly applying the simplification rules in Figure 7.1 and evaluating ground terms, the above formula simplifies to the following SULFA nested `IF` predicate:

```
(NOT (IF (NOT (EQUAL (CAR X) A))
         NIL
       (NOT (IF (NOT (NOT (CONSP X)))
                (EQUAL (CAR X) A)
              (EQUAL A A))))).
```

If the negation in the above formula is removed, it becomes:

```
(IF (EQUAL (CAR X) A)
    (IF (CONSP X)
        (EQUAL (CAR X) A)
      (EQUAL A A))
  T).
```

Let F be a SULFA nested `IF` predicate containing the set $Z \subset \mathbb{E}$ of SULFA Boolean predicates. Then, if each member of Z is generalized to a variable, the result is a nested `IF` that can be translated into CNF. If a SAT solver returns unsatisfiable, then the original SULFA formula is valid. Otherwise, the original formula is valid only if the formula

MakeSetImp$(C, \ulcorner$NIL$\urcorner)$ is valid, where each element of C either is $E \in Z$, if E corresponds to a **true** variable in the satisfying instance; or MakeNot(E), if E corresponds to a **false** variable in the satisfying instance. In our example, a satisfying instance returned by the SAT solver could lead to (NOT (CONSP X)), (EQUAL (CAR X) A), and (NOT (EQUAL A A)), which is a spurious counter-example

Any SULFA Boolean predicate, such as (EQUAL X X), that is reducible to a constant can be easily simplified. If E is an instance of (EQUAL X X), then it is simplified to 'T; otherwise, if E is an application of EQUAL such that one of its arguments, X, is a subterm of the other, then it is simplified to (EQUAL X 'NIL). For example, (EQUAL A (CAR A)) is simplified to (EQUAL A 'NIL). MakeEq$(X,$ NIL$)$. Our example formula therefore simplifies to:

```
(IF (EQUAL (CAR X) A)
    (IF (CONSP X)
        (EQUAL (CAR X) A)
      T)
  T).
```

The above formula generalizes to a valid nested IF term, which can be proven using a SAT solver.

Further spurious counterexamples can occur due to relations between SULFA Boolean predicates. For example, if (CONSP X) is true, then (EQUAL X NIL) must be false. Many spurious counterexamples can be avoided, however, by instantiating the theorems in Figure 7.2. For example, by adding instances of the theorems in Figure 7.2, our example becomes:

```
1. (IMPLIES (AND (EQUAL X Y) (EQUAL Y Z)) (EQUAL X Z))

2. (IMPLIES (NOT (CONSP X)) (EQUAL (CAR X) NIL))

3. (IMPLIES (NOT (CONSP X)) (EQUAL (CDR X) NIL))

4. (IMPLIES (AND (CONSP X) (NOT (CONSP Y))) (NOT (EQUAL X Y)))

5. (IMPLIES (CONSP X) (EQUAL (CONSP X) T))

6. (IMPLIES (EQUAL X Y) (EQUAL (EQUAL X Y) T))

   (IMPLIES (AND (EQUAL (CAR X) (CAR Y))
                 (EQUAL (CDR X) (CDR Y))
7.               (CONSP X)
                 (CONSP Y))
            (EQUAL X Y))
```

Figure 7.2: ACL2 theorems instantiated after generalization to prevent spurious counter examples.

```
(IMPLIES
 (IMPLIES (NOT (CONSP X)) (EQUAL (CAR X) NIL))
 (IF (EQUAL (CAR X) A)
   (IF (CONSP X)
       (EQUAL (CAR X) A)
     T)
   T)).
```

Note that IMPLIES and NOT are easily translated to IF terms.

Since each of the theorems in Figure 7.2 is an ACL2 theorem, adding instantiations of them as hypotheses does not affect the validity of an ACL2 formula. Our algorithm instantiates every instance of the theorems in Figure 7.2 that we believe may be relevant to the SAT solver, to avoid as many spurious counterexamples as possible.

7.4 Developing an Efficient SAT-Based Proof Procedure

The previous section shows that it is possible to use SAT solvers to aid in the proof and disproof of SULFA formulas. This section describes the optimizations needed to make an efficient SAT-based SULFA verification technique, and provides an overview of the resulting tool. Note that the subject of uninterpreted functions is omitted from this section and instead discussed in Section 7.7.

To develop an efficient algorithm for translating a SULFA formula to CNF, a trade off is managed between the following three goals:

1. **Avoid duplication of terms**: In simpler versions of our algorithm, complex terms are often duplicated. For example, unrolling `(F (IF A B C))` might lead to the duplication of `(IF A B C)` since the formal parameter of `F` may occur multiple times in its body. Such duplication leads to an exponential explosion that can be avoided by creating new variables. For example, `(F (IF A B C))` is equal to

   ```
   (IMPLIES (EQUAL V (IF A B C)) (F V)),
   ```

   so that `F` can be unrolled without duplicating `(IF A B C)`. Therefore, the efficient translation algorithm creates variables for complex terms before those terms are likely to be duplicated.

2. **Avoid the creation of unnecessary variables**: On the other hand, the performance of a SAT solver on a given problem is often closely related to the number of Boolean variables present in the problem. Furthermore, the size of the term can increase from the unnecessary creation of variables. For example,

   ```
   (IF (F0 A0) (F1 A1) (F2 A2))
   ```

   might unroll to `A1`, if `(F0 A0)` expands to `T` and `(F1 A1)` expands to A1. If a variable is created for `(F0 A0)`, however,

```
(IF (F0 A0) (F1 A1) (F2 A2))
```

becomes

```
(IMPLIES (EQUAL V (F0 A0)) (IF V (F1 A1) (F2 A2))).
```

Now, `(F2 A2)` is likely to be unrolled, despite the fact that it is irrelevant to the original formula.

The efficient translation algorithm avoids unrolling irrelevant terms such as `(F2 A2)` above, by restricting variable creation to `IF` terms, and performing minimal simplification on `IF` terms before unrolling or variable creation.

3. **Avoid simplifying irrelevant subterms**: Note that the rewrite rules in Figure 7.1, implement a form of cone of influence reduction. For example, the formula

```
(EQUAL (CAR (IF A
                (CONS (CONS (F X) (F Y)) (F Z))
               (CONS X (G Y))))
       NIL).
```

simplifies to

```
(EQUAL (IF A (CONS (F X) (F Y)) X)
       NIL)
```

by rules 1 and 4 in Figure 7.1. This further simplifies to

```
(IF (EQUAL A NIL)
    (EQUAL X NIL)
  (EQUAL (CONS (F X) (F Y)) NIL))
```

by rules 7 in Figure 7.1. Finally, this is simplified to

```
(IF (EQUAL A NIL) (EQUAL X NIL) NIL)
```

by rule 8 in Figure 7.1, which produces an IF with a trivial condition that is then simplified to NIL—essentially, the CONS of two elements is never equal to NIL. Note that the subterms (F X), (F Y), (F Z), and (G Y) in the original example formula have no bearing its validity.

The efficient translation algorithm, for the most part, avoids manipulating or creating variables for subterms that can be removed by the rewrite rules in Figure 7.1. In practice, this is an important technique for hardware verification, since it enables the verification of shallow properties of large hardware designs.

An overview of the optimized SAT-based proof procedure, which manages a trade-off between the three above goals, is illustrated in Figure 7.4. The procedure is divided into the following phases:

- **SULFA Recognizer**. The recognizer described in Section 5.5 is used to efficiently determine if the given formula is in SULFA. If it is not, then an error message is presented to the user. Otherwise, the procedure continues.

- **Basic Simplification.** This phase performs some minimal simplification of the *current term*, which is initially the formula and later the definition of a symbol occurring in the formula (the current term is updated at the end of the normalization phase). The simplification in this phase generally involves only the evaluation proof technique, simplification based on the axioms of CONS, and simplification based on the axioms of IF. For example, (CAR (IF T (CONS X Y) Z)) is simplified to X.

- **Definition Creation.** First, define an *inner-if term* as an IF term that occurs either within the argument of a user-defined function, the condition of an IF, or an argument of EQUAL. Furthermore, an *inner-if* must not have any user-defined functions

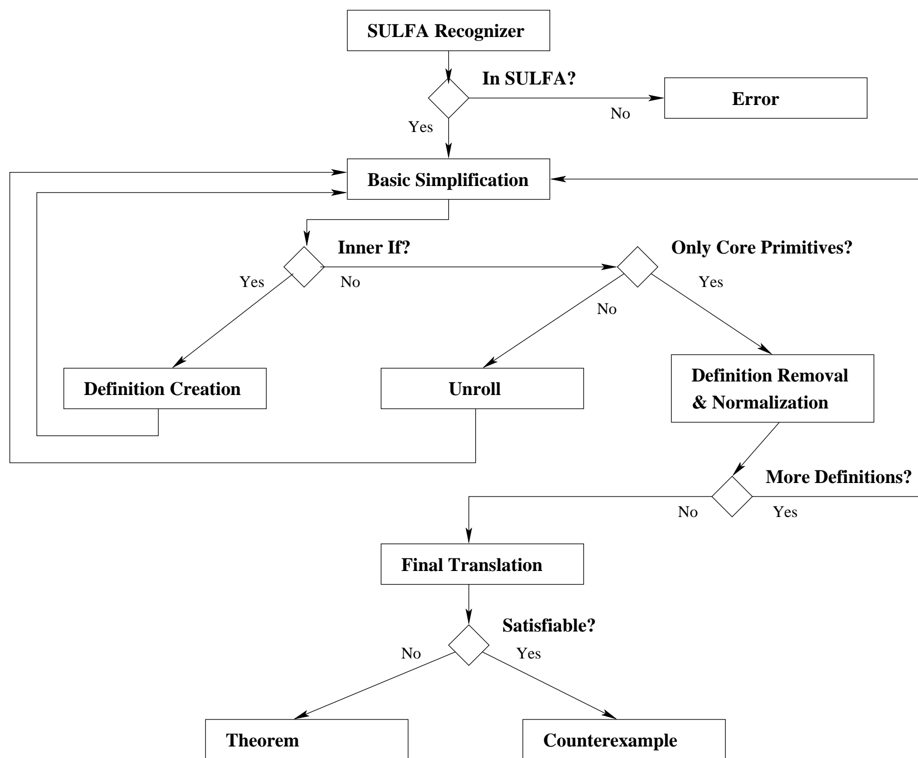


Figure 7.3: An overview of our tool that verifies SULFA formulas using SAT Solvers

in its condition. For example, `(F (IF (CAR X) (F Y) Z))` contains an inner-if, whereas `(F (IF (F X) (F Y) Z))`, `(IF (CAR X) (F Y) Z)`, and `(IF A (IF (CAR X) (F Y) Z))` do not.

During this phase, new symbols are defined to represent inner-if terms occurring in the current term. For example, `(F (IF (CAR X) (F Y) Z))` is replaced with `(F V)`, where `V` is defined to be `(IF (CAR X) (F Y) Z)`.

- **Unroll.** This phase expands functions in the current term.

- **Definition Removal and Normalization.** If the current term is the formula, then it is equivalent to ⌜`(NOT (EQUAL X NIL))`⌝/[`X` $\mapsto F$], which is reduced to a SULFA nested `IF` by outside-in rewriting based on the theorems in Figure 7.1.

  Otherwise, the formula is a SULFA nested `IF` predicate and the current term defines a symbol that occurs in the formula. The definition is now removed by mapping its symbol to the current term. Next, the formula is again reduced to a SULFA nested `IF` predicate by outside-in rewriting, based on the theorems in Figure 7.1.

  New variables are also created for the resulting SULFA Boolean predicates in the same manner. For example, if `V0` is a symbol with a definition that has not yet been traversed, then `(IF A (EQUAL NIL V0) (EQUAL NIL V0))` is translated to

  ```
  (IMPLIES (IFF V0-P0 (EQUAL NIL V0)) (IF A V0-P0 V0-P0)).
  ```

  When the symbol removal phase later substitutes a term for `V0`, this term will not be duplicated.

  Once the formula is normalized, the most recent definition not already removed, is chosen as the new current term. If no definitions exist that have not been removed then the algorithm proceeds to the final translation phase.

- **Final Translation.** The final translation phase generalizes any remaining SULFA Boolean predicates, instantiates the theorems in Figure 7.2, as described in Sec-

tion 7.3, and then translates the resulting nested IF into CNF by the method described in Section 7.2.

- **Counterexample.** If the SAT solver returns satisfiable, then the satisfying instance is a mapping from SULFA Boolean predicates to Booleans. In a manner similar to that described in Section 6.5, this mapping is translated, if possible, into a concrete ACL2 object that is a counterexample to the original formula.

The algorithm in Figure 7.4 is further optimized to avoid simplifying irrelevant subterms. Note that in the definition removal and normalization phase, there is some finite set of SULFA Boolean predicates in which the current term will eventually be instantiated. The basic simplification, symbol creation, and unrolling phases are, therefore, optimized to ignore subterms that are irrelevant to the set of predicates in which the current term will eventually be instantiated. Any unsimplified terms are eventually removed by the definition removal and normalization phase.

7.5 Example

This section translates an example SULFA formula into CNF using the algorithm shown in Figure 7.4. First define UNARY-AND as:

```
(DEFUN UNARY-AND (N X)
  (IF (ZP N) T
    (IF (CAR X)
        (UNARY-AND (1- N) (CDR X))
      NIL))),
```

which computes the conjunction of the first n-bits in a bit vector. In the body of UNARY-AND, the function ZP returns true if its argument is the natural number 0, and the function "1-"

subtracts 1 from its argument. The following is a valid SULFA formula, which we we name *uAndForm*:

```
(EQUAL (UNARY-AND 2 (IF A (CONS B (CONS C (CONS (F A) NIL)))
                          (CONS C (CONS B NIL))))
       (IF B (IF C T NIL) NIL)).
```

where F is any arbitrary function with a valid ground formal set.

The formula *uAndForm* is valid since both the conjunction of B and C and the conjunction of C and B, are equal to T if B and C are non-NIL; otherwise, both are NIL.

We will now translate the theorem *uAndForm* to CNF. The first step is to simplify the formula, using the axioms of IF, CONS, and evaluation, which yields no change to the formula. Next, the definition creation phase produces the following:

```
(LET* ((V0 (IF A (CONS B (CONS C (CONS (F A) NIL)))
                 (CONS C (CONS B NIL))))
       (V1 (IF B (IF C T NIL) NIL)))
  (EQUAL (UNARY-AND 2 V0)
         V1)).
```

Two symbols, V0 and V1, are created to represent inner-if terms. These symbols are defined in a LET* (LET* is described in Chapter 3).

The next phase that alters the formula is the unroll phase, which produces:

```
(LET* ((V0 (IF A (CONS B (CONS C (CONS (F A) NIL)))
                 (CONS C (CONS B NIL))))
       (V1 (IF B (IF C T NIL) NIL)))
  (EQUAL (IF (ZP 2) T
           (IF (CAR V0) (UNARY-AND (1- 2) (CDR V0))
             NIL))
         V1)).
```

The above formula is equivalent to *uAndForm* by the definition of UNARY-AND.

The translation algorithm next uses the basic simplification phase to simplify the body of the LET* into:

```
(LET* ((V0 (IF A (CONS B (CONS C (CONS (F A) NIL)))
             (CONS C (CONS B NIL))))
       (V1 (IF B (IF C T NIL) NIL)))
  (EQUAL (IF (CAR V0) (UNARY-AND 1 (CDR V0)) NIL)
         V1)).
```

which is equivalent to the previous formula by evaluation and the axioms of IF.

Next, the definition creation phase translates the formula into the following:

```
(LET* ((V0 (IF A (CONS B (CONS C (CONS (F A) NIL)))
             (CONS C (CONS B NIL))))
       (V1 (IF B (IF C T NIL) NIL))
       (V2 (IF (CAR V0) (UNARY-AND 1 (CDR V0)) NIL)))
 (EQUAL V2 V1)).
```

Next, the translation process proceeds to the definition removal and normalization phase. Normally, this phase must reduce the current term into a SULFA nested IF predicate, but in the above example the current term (in this case, the body of the LET*) is already a SULFA nested IF predicate, so no normalization is needed. Instead, a variable is simply created to represent each SULFA Boolean predicate in the current term, as shown below:

```
(LET* ((V0 (IF A (CONS B (CONS C (CONS (F A) NIL)))
             (CONS C (CONS B NIL))))
       (V1 (IF B (IF C T NIL) NIL))
       (V2 (IF (CAR V0) (UNARY-AND 1 (CDR V0)) NIL)))
 (IMPLIES (IFF V2-P0 (EQUAL V1 V2))
          V2-P0)).
```

The new symbol, V2-P0, is associated with V2 since V2 is the more recently defined symbol in the SULFA Boolean predicate it represents. At first glance the above formula appears weaker than the previous formula, since V2-P0 is not restricted to be Boolean. However, the two formulas are equivalent since V2-P0 is never used in a context that distinguishes between non-NIL constants.

Also note that the arguments of the equality have been reordered. Although not previously mentioned, arguments of SULFA Boolean predicates are ordered to avoid creating equivalent predicates.

At this point, the simplification continues with the current term becoming the bottom-most definition, which is the definition of V2. The unroll phase is the next phase to modify the formula, as shown below:

```
(LET* ((V0 (IF A (CONS B (CONS C (CONS (F A) NIL)))
             (CONS C (CONS B NIL))))
       (V1 (IF B (IF C T NIL) NIL))
       (V2 (IF (CAR V0)
               (IF (ZP 1) T
                 (IF (CAR (CDR V0))
                     (UNARY-AND (1- 1) (CDR (CDR V0)))
                   NIL))
             NIL)))
 (IMPLIES (IFF V2-P0 (EQUAL V1 V2))
          V2-P0)).
```

The above formula is equivalent to the previous formula, by the definition of UNARY-AND.

Next, the basic simplification phase produces:

```
(LET* ((V0 (IF A (CONS B (CONS C (CONS (F A) NIL)))
             (CONS C (CONS B NIL))))
       (V1 (IF B (IF C T NIL) NIL))
       (V2 (IF (CAR V0)
               (IF (CAR (CDR V0))
                   (UNARY-AND 0 (CDR (CDR V0)))
                   NIL)
             NIL)))
 (IMPLIES (IFF V2-P0 (EQUAL V1 V2))
          V2-P0)),
```

which is equivalent to the previous formula by evaluation and the axioms of `IF`.

After another round of unrolling and basic simplification, the following formula is produced:

```
(LET* ((V0 (IF A (CONS B (CONS C (CONS (F A) NIL)))
             (CONS C (CONS B NIL))))
       (V1 (IF B (IF C T NIL) NIL))
       (V2 (IF (CAR V0) (IF (CAR (CDR V0)) T NIL) NIL)))
 (IMPLIES (IFF V2-P0 (EQUAL V1 V2))
          V2-P0)),
```

which is equivalent to the previous formula by the definition of `UNARY-AND`, evaluation, and the axioms of `IF`.

Next, `V2` is removed by substituting its body into the body of the `LET*`, leading to:

```
(LET* ((V0 (IF A (CONS B (CONS C (CONS (F A) NIL)))
             (CONS C (CONS B NIL))))
       (V1 (IF B (IF C T NIL) NIL)))
 (IMPLIES (IFF V2-P0 (EQUAL V1
                            (IF (CAR V0)
                                (IF (CAR (CDR V0)) T NIL)
                              NIL)))
          V2-P0)).
```

Note that the definition of V2 in the LET* has been removed, and that the body of the LET* is a SULFA nested IF term.

Normalization of the LET* body now occurs, using outside rewriting based on the theorems in Figure 7.1:

```
(LET* ((V0 (IF A (CONS B (CONS C (CONS (F A) NIL)))
             (CONS C (CONS B NIL))))
       (V1 (IF B (IF C T NIL) NIL)))
 (IMPLIES (IFF V2-P0 (IF (EQUAL NIL (CAR V0)) (EQUAL NIL V1)
                       (IF (EQUAL NIL (CAR (CDR V0)))
                           (EQUAL NIL V1)
                         (EQUAL T V1))))
          V2-P0)),
```

which is equivalent to the previous formula, by the axioms of IF and EQUAL.

Next, variables are created for each SULFA Boolean predicate, which produces:

```
(LET* ((V0 (IF A (CONS B (CONS C (CONS (F A) NIL)))
             (CONS C (CONS B NIL))))
       (V1 (IF B (IF C T NIL) NIL)))
  (IMPLIES
   (AND (IFF V0-P0 (EQUAL NIL (CAR V0)))
        (IFF V0-P1 (EQUAL NIL (CAR (CDR V0))))
        (IFF V1-P0 (EQUAL NIL V1))
        (IFF V1-P1 (EQUAL T V1))
        (IFF V2-P0 (IF V0-P0 V1-P0 (IF V0-P1 V1-P0 V1-P1))))
   V2-P0)).
```

Next, the translation algorithm continues, with the current term equal to `V1`. Since `V1` contains no defined functions or inner-if terms, the translation proceeds directly to the definition removal and normalization phase, which produces:

```
(LET* ((V0 (IF A (CONS B (CONS C (CONS (F A) NIL)))
             (CONS C (CONS B NIL)))))
  (IMPLIES
   (AND (IFF B-P0 (EQUAL NIL B))
        (IFF C-P0 (EQUAL NIL C))
        (IFF V0-P0 (EQUAL NIL (CAR V0)))
        (IFF V0-P1 (EQUAL NIL (CAR (CDR V0))))
        (IFF V1-P0 (IF B-P0 T (IF C-P0 T NIL)))
        (IFF V1-P1 (IF B-P0 NIL (IF C-P0 NIL T)))
        (IFF V2-P0 (IF V0-P0 V1-P0 (IF V0-P1 V1-P0 V1-P1))))
   V2-P0)).
```

The above formula is produces by substituting `V1` into the terms for `V1-P0` and `V1-P1` and then performing outside-in rewriting based on the theorems in Figure 7.1.

Next, simplification continues on `V0`. The term `(F A)` is not expanded, since it is irrelevant to the SULFA Boolean predicates involving `V0`. Instead, the translation proceeds directly to the definition removal and normalization phase, which produces:

```
(IMPLIES
 (AND
  (IFF A-P0 (EQUAL NIL A))
  (IFF B-P0 (EQUAL NIL B))
  (IFF C-P0 (EQUAL NIL C))
  (IFF V0-P0 (IF A-P0 C-P0 B-P0))
  (IFF V0-P1 (IF A-P0 B-P0 C-P0))
  (IFF V1-P0 (IF B-P0 T (IF C-P0 T NIL)))
  (IFF V1-P1 (IF B-P0 NIL (IF C-P0 NIL T)))
  (IFF V2-P0 (IF V0-P0 V1-P0 (IF V0-P1 V1-P0 V1-P1))))
 V2-P0).
```

Note that the theorems in Figure 7.1 act here as a form of cone of influence reduction, removing the unwanted application of `F`.

At this point, the remaining SULFA Boolean predicates are generalized by removing the hypotheses involving them. The theorems in Figure 7.2 are instantiated over the generalized predicates, as needed. In this case, since the predicates are `(EQUAL NIL A)`, `(EQUAL NIL B)`, and `(EQUAL NIL C)`, no instantiation is necessary. Finally, the formula is translated into CNF, using the algorithm described in Section 7.2. A SAT solver then returns unsatisfiable, so *uAndForm* is valid.

Next, consider what happens if we attempt to prove the following formula:

```
(EQUAL (UNARY-AND 2 (IF A (CONS B (CONS C (CONS (F A) NIL)))
                          (CONS C (CONS B NIL))))
       (IF B C NIL)).
```

The above formula is similar to *uAndForm*, but is **not** a theorem, because `UNARY-AND` always returns `T` or `NIL`, whereas `(IF B C NIL)` returns the value of `C` when `B` is non-`NIL`.

The translation process follows in a similar manner as previously, except that `V1` is given the value `(IF B C NIL)`, instead of `(IF B (IF C T NIL) NIL)`, which leads to the following formula:

```
(IMPLIES
 (AND
  (IFF A-P0 (EQUAL NIL A))
  (IFF B-P0 (EQUAL NIL B))
  (IFF C-P0 (EQUAL NIL C))
  (IFF C-P1 (EQUAL T C))
  (IFF V0-P0 (IF A-P0 C-P0 B-P0))
  (IFF V0-P1 (IF A-P0 B-P0 C-P0))
  (IFF V1-P0 (IF B-P0 T C-P0))
  (IFF V1-P1 (IF B-P0 NIL C-P1))
  (IFF V2-P0 (IF V0-P0 V1-P0 (IF V0-P1 V1-P0 V1-P1))))
 V2-P0).
```

Note that applying the rules in Figure 7.1 creates a test on `(EQUAL T C)`, since `V1-P1` is `(EQUAL T V1)`. The SAT solver therefore returns the following satisfying instance: `(EQUAL NIL A)` is false, `(EQUAL NIL B)` is false, `(EQUAL NIL C)` is false, and `(EQUAL T C)` is false. This corresponds to the case where `A` is `T`, `B` is `T`, and `C` is `0`, which is a counterexample to the original formula.

7.6 Notes on Complexity and Efficiency

Our SAT-based SULFA procedure suffers from multiple exponential complexity issues. SAT solving is itself an NP-Complete problem, thus the CNF formula may require ex-

```
(DEFUN BAR (N X ANS)
 (DECLARE (XARGS :MEASURE (NFIX N)))
  (COND
   ((ZP N)
    ANS)
   (T
    (OR (BAR (1- N) (CDR X) (OR (EQUAL (CAR X) 0) ANS))
        (BAR (1- N) (CDR X) (OR (EQUAL (CAR X) 1) ANS))))))

(DEFUN BAR-BETTER (N X ANS)
 (DECLARE (XARGS :MEASURE (NFIX N)))
  (COND
   ((ZP N)
    ANS)
   (T
    (BAR-BETTER (1- N)
                (CDR X)
                (OR (EQUAL (CAR X) 0)
                    (EQUAL (CAR X) 1)
                    ANS)))))
```

Figure 7.4: Two equivalent ACL2 definitions, which produce drastically different performance when used in SULFA formulas. To unroll an application of BAR, with a first argument equal to the natural number N, BAR must be expanded 2^N times. On the other hand, BAR-BETTER, only needs to be expanded N times.

ponential time to solve. Furthermore, the complexity of unrolling can, in the worst case, be as large as the highest complexity ACL2 function possible, since any ACL2 function could be used to compute the ordinal that decreases during unrolling. Furthermore, our algorithm that instantiates the theorems in Figure 7.2 produces an exponential number of hypotheses in some cases.

In practice though, these exponential issues often can be avoided:

- SAT solvers, while exponential in the number of variables in the worst case, can often solve problems arising from hardware verification containing tens of thousands

of variables [90].

- Also, many interesting properties can be solved without exponential unrolling. Sometimes the definition of a function can be rewritten to keep the amount of unrolling small. For example, properties involving the `BAR-BETTER` function in Figure 7.4 require significantly less unrolling than properties involving the equivalent function `BAR`.

- Finally, large numbers of instantiations of the theorems in Figure 7.2 only result when the transitivity theorem (theorem number 1) is instantiated. This theorem is only instantiated though when multiple SULFA Boolean Predicates of the form `(EQUAL X Y)` exist, where neither `X` nor `Y` are constant. Usually in our work, however, we use bit-vector equality rather than `EQUAL`, which does not result in such SULFA Boolean Predicates.

7.7 Uninterpreted Functions

The SAT based proof procedure has been extended to include uninterpreted functions using a mechanism similar to that described in Section 6.3. However, the current extension is a prototype and not very efficient.

Note that the algorithm in Section 6.3 requires uninterpreted functions to be removed inside-out. However, in order to support a cone of influence reduction, our proof procedure presented in this chapter is primarily outside-in. The result is that as uninterpreted functions are removed, previously removed definitions must be revisited.

Another problem is that, as mentioned in Section 7.6, our algorithm is inefficient when the transitivity theorem in Figure 7.2 needs to be instantiated a lot. However, removing uninterpreted functions produces exactly the type of SULFA Boolean predicates that lead to multiple instantiations of the transitivity theorem.

Chapter 13 presents some ideas on how uninterpreted functions may be handled

efficiently.

7.8 Summary

This chapter presents a SAT-based proof procedure for SULFA formulas, with a number of important optimizations. New variables are created to reduce the duplication of terms during unrolling and normalization. Furthermore, an outside-in cone of influence reduction is used to avoid reasoning about irrelevant parts of tree structures.

An implementation of the SAT-based procedure is distributed with the ACL2 theorem prover [37] and evidence of its efficiency can be found in later chapters. Chapter 8 uses the procedure, along with the ACL2 theorem prover, to verify a significant hardware design. Furthermore, chapter 9 uses the procedure, along with the ACL2 theorem prover, to implement a solver for a standard bit-vector theory and to verify a set of benchmark problems in that theory.

The SAT-based proof procedure is the first publicly-available integration of an external proof engine, such as SAT solvers, with the ACL2 theorem prover. It also helped guide the creation of the SixthSense integration with ACL2, described in Chapter 10, and motivate ACL2's new general-purpose extension mechanism for external tools, described in Chapter 11. We believe the integration of general-purpose theorem provers, such as ACL2, with other proof engines, such as external tools, BDD packages, model checkers, and resolution provers, will greatly decrease the amount of human-effort required during large-scale verification efforts.

7.9 Development and Bibliographic Notes

Proof techniques based on SAT solvers have been developed for various useful theories, including the theories of linear arithmetic, arrays, and bit vectors. The ICS tool, which was integrated with the PVS general-purpose theorem prover, contains a decision procedure for

tree structures [12]. ICS uses a lazy approach, based on incremental SAT solvers, whereas SULFA uses an eager approach, making only a single call to a SAT solver. When full translation to CNF can be performed efficiently, we believe an eager approach is more effective. In particular, we believe a general-purpose bit-vector SMT solver, such as the one described in Chapter 9, benefits from the eager approach, since bit-vector SMT solvers traditionally use an eager approach.

ICS has recently been replaced with Yices [92]. Yices contains support for inductive datatypes, a general-purpose mechanism capable of supporting tree structures. More recently, CVC3 [85] has also added support for inductive datatypes. It is not clear, however, if the use of a more general datatype leads to a loss in performance or more spurious counterexamples than result from a domain-specific approach.

Chapter 8

The Verification of the TRIPS Processor's Data-Tile Protocol Implementation

8.1 Introduction

One application of the SULFA subclass described in Chapter 5 and its SAT-based SULFA solver described in Chapter 7 is to aid in the verification of ACL2 models of hardware designs. This chapter applies the SAT-based SULFA solver to the verification of a component of the TRIPS processor. The TRIPS processor is a prototype next-generation processor that was designed and fabricated by the University of Texas and IBM [9, 73]. We applied our technique to a unique component of the processor that implements a data tile protocol. The data tile protocol is part of the TRIPS processor's unique decentralized design, which is intended to provide a complexity-effective way for increasing the capacity and bandwidth of a processor's memory system [79, 80].

Our approach involves the automatic extraction of an ACL2 model from the Verilog design. The ACL2 model is then specified and verified using the ACL2 theorem prover. In-

stead of verifying the design using the standard rewriting and induction approach, however, we use rewriting and induction to reduce the proof that the ACL2 model satisfies its specification into a set of formulas that can be verified automatically by the SAT-based SULFA solver. Through our approach, the full power of the ACL2 theorem prover is available, but the SULFA solver provides an alternative that, when applicable, substantially decreases the human effort required to complete ACL2 proofs.

This chapter first overviews the TRIPS processor in Section 8.2. Section 8.3 then describes the data-tile communication protocol, which is necessary for speculative execution. Section 8.4 then describes our verification methodology, before applying it in Sections 8.5 and 8.6 to the verification of the data-tile protocol.

8.2 Overview of the TRIPS Processor

The TRIPS processor is a dual-core processor, with each core further divided into tiles as shown in Figure 8.1 [1]. Different tiles, for the most part, only communicate with neighboring tiles, and only once per-cycle. The TRIPS processor contains sixteen execution tiles (each marked as E), accessing four banks of registers (each marked as R), which allows 16 instructions to be potentially executed in parallel. Furthermore, memory is divided into four partitions, each with its own instruction cache (marked as I) and data cache (marked as D). A single global tile (marked as G) is used to coordinate actions that are universal to the entire processor core.

The tile-based design methodology helps to address the latency (wire delays), power, and complexity issues facing next-generation microprocessors. Latency issues are addressed by localizing most computation within each tile. The power density can be reduced by spreading out the tiles on the chip. Complexity is reduced by reusing a single tile design many times on a chip.

[1] This figure is copied from Burger et al. [9, 73].

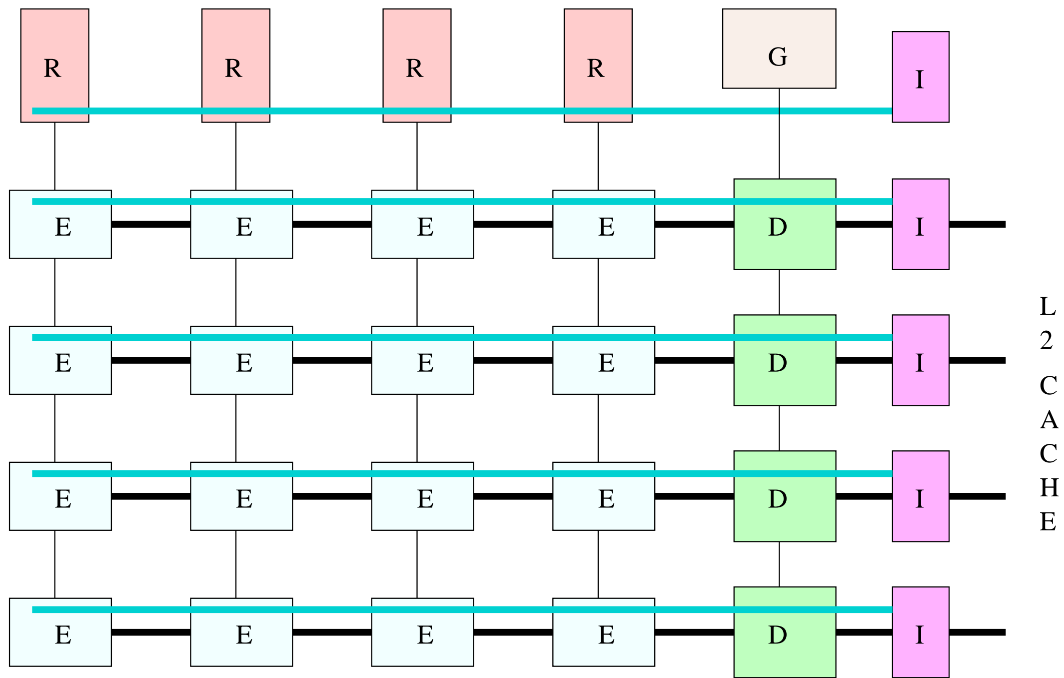


Figure 8.1: An overview of the design of a TRIPS processor core, which contains 16 execution tiles (each marked as E), four register tiles (each marked as R), four data tiles (each marked as D), five instruction cache tiles (each marked as I), and a single global tile (G).

8.3 Overview of the Data-Tile Protocol

The TRIPS processor executes up to 256 memory instructions speculatively. These 256 instructions are divided into eight instructions blocks, each containing 32 instructions. All 32 instructions in a given block are dispatched, committed, and flushed together, as managed by the global tile.

As shown in Figure 8.1, the TRIPS processor core design contains four memory partitions, which are part of its decentralized design for executing load and store instructions [79, 80]. While most data-tile functions can be performed locally, the data tiles must communicate to accumulate two types of information needed for speculative execution: 1) which instruction blocks have caused exceptions and 2) which store instructions have been (or have begun to be) executed.

Before going into more detail, we first introduce the following terminology:

- A data tile is said to be *above* another data tile if it is closer to the global tile, as shown in Figure 8.1. The closest tile to the global tile is referred to as the *top tile* and the furthest from the global tile is referred to as the *bottom tile*.
- We refer to the protocols used to accumulate exception and store information as the *exception protocol* and the *store protocol* respectively. Together these two protocols make up the data-tile protocol.
- The component of the data tile that implements the exception and store protocols is called the *Data Status Network (DSN)*.

An overview of the communication between the four data tiles is shown in Figure 8.2. Information about committed and flushed instruction blocks initiated from the global tile are communicated to the top data tile as 8 bit flush and commit masks and are then communicated downward each cycle (four cycles are required for the flush or commit of an instruction block to affect the bottom tile). Two types of exceptions can occur; each

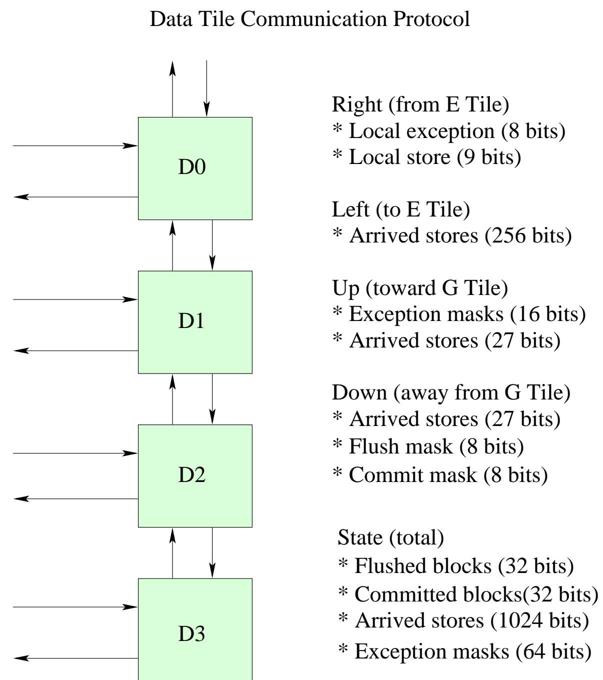


Figure 8.2: An overview of the communication between the four data tiles of the TRIPS processor.

of which are input to each data tile through a one bit enable and three bit instruction block address (actually one of these exceptions is computed by the data tile, but for simplicity we will consider it an input). Also, one store can occur within each memory partition each cycle, and its arrival is announced to each data tile through a one bit enable and eight bit instruction address. The arrived stores are then output as a 256 bit mask to each execution tile (to the left) and the exceptions are output as two 8 bit masks to the global tile (up above).

To create the exception output mask, the data tiles communicate all known exceptions in a mask sent upward each cycle. Thus, an exception occurring in the bottom tile requires four cycles to be reported to the global tile, whereas an exception occurring in the top tile requires only a single cycle to be reported. To create the mask of arrived stores, the arrival of up to three stores are communicated upward and downward each cycle (this is sent as a 27 bit signal including an enable bit and an eight bit address per store). A total of 1152 bits of state are required to implement the protocol, including bits for each each potentially arrived store (256 bits per tile), each possible exception (16 bits per tile), each possible flushed block (8 bits per tile) and each committed instruction block (8 bits per tile).

8.4 Formal Verification Methodology

Figure 8.3 presents an overview of the hardware verification methodology we use to verify the data-tile protocol. We start with a Verilog design and an informal specification, composed of documentation, models, and test suites. The Verilog design is translated automatically into a circuit description in the DE2 language, as described in Chapter 12. The DE2 description is then automatically translated into an ACL2 finite state machine model. A proof of equivalence between the ACL2 model and its DE2 description (relative to the semantics of the DE2 language as encoded in ACL2) is also automatically generated and checked with the ACL2 theorem prover. A formal specification is then written relative to the ACL2 model and proven correct through a mixture of user-guided theorem proving and the SAT-based SULFA solver described in Chapter 7.

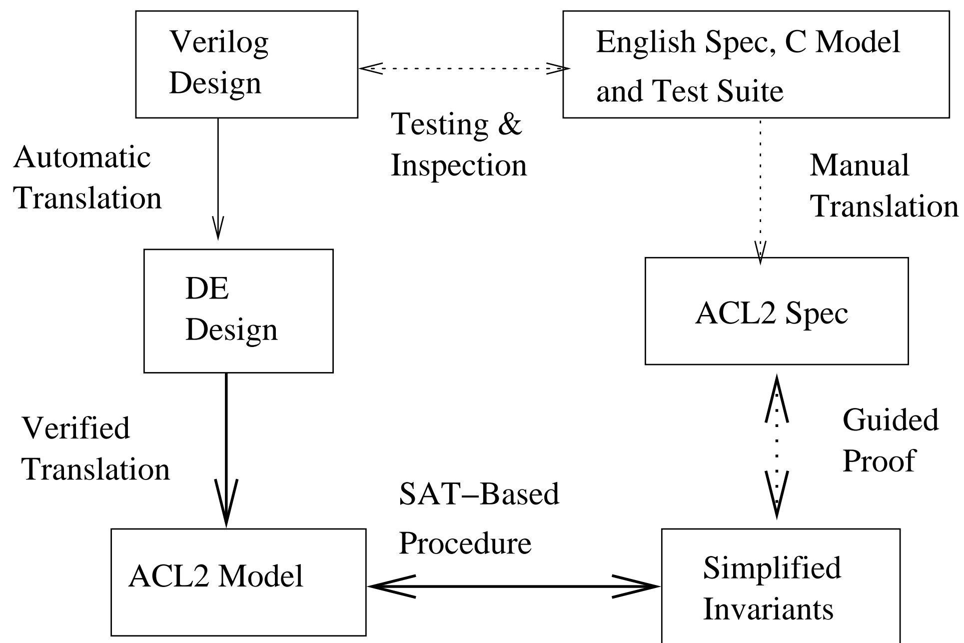


Figure 8.3: An overview of our verification methodology

Safety-Theorem:
$(\tau \in \mathbb{N}) \rightarrow \mathsf{P}(\mathsf{Tth\text{-}state}(\tau, \iota), \mathsf{nth}(\tau, \iota))$

follows from the following three finite-step theorems:

1. $\mathsf{inv}(\mathbf{S_0})$
2. $\mathsf{inv}(S) \rightarrow \mathsf{inv}(\mathsf{step}(S, I))$
3. $\mathsf{inv}(S) \rightarrow \mathsf{P}(S, I)$

Figure 8.4: An outline of our basic strategy for verifying safety properties expressed as ACL2 theorems. Note that all unbound variables, such as S and I in the second finite-step theorem, are implicitly universally quantified over all ACL2 objects.

8.4.1 Verification of Safety Properties

The ACL2 machine model consists of an initial state, $\mathbf{S}_0$; a step function $\mathsf{step}(S, I)$, which maps the machine state S at the beginning of a cycle and its inputs I during the cycle to the machine state at the beginning of the following cycle; and functions that return each output of the machine, given its current state and inputs. Thus, the state at clock cycle τ, given a sequence of machine inputs ι, is defined as:

$$\mathsf{Tth\text{-}state}(\tau, \iota) \triangleq \begin{cases} \mathbf{S}_0, & \text{if } \mathsf{zp}(\tau) \\ \mathsf{step}(\mathsf{Tth\text{-}state}(\tau - 1, \iota), \mathsf{nth}(\tau, \iota)), & \text{otherwise.} \end{cases}$$

A safety property is a predicate on the state of the machine $\mathsf{Tth\text{-}state}(\tau, \iota)$ that is true for all time τ and any machine input sequence ι. The basic strategy for verifying safety properties is shown in Figure 8.4. Note that the function $\mathsf{nth}(n, x)$ returns the nth element of the list x. A machine input sequence ι is represented as a list, so that $\mathsf{nth}(\tau, \iota)$ is the machine inputs at time τ.

The proof of safety properties shown in Figure 8.4 essentially strengthens the target safety property into an inductive invariant, which then follows from three single-step properties. First, the inductive invariant is shown to be true of the initial state. Then, it is shown

Augmented-Safety-Theorem:

$(\tau \in \mathbb{N}) \wedge \mathsf{var}(v) \wedge \mathsf{Tth\text{-}inp}(\tau, \iota) \rightarrow \mathsf{P}(v, \mathsf{Tth\text{-}state}(\tau, \iota), \mathsf{nth}(\tau, \iota))$

follows from the following three finite-step theorems:

1. $\mathsf{var}(v) \rightarrow \mathsf{inv}(v, \mathbf{S_0})$
2. $\mathsf{var}(v) \wedge \mathsf{inp}(S, I) \wedge \mathsf{inv}(v, S) \rightarrow \mathsf{inv}(v, \mathsf{step}(S, I))$
3. $\mathsf{var}(v) \wedge \mathsf{inp}(S, I) \wedge \mathsf{inv}(v, S) \rightarrow \mathsf{P}(v, S, I)$

Figure 8.5: An augmentation of the basic strategy shown in Figure 8.4 for proving safety properties. The augmented strategy includes an additional variable v and predicates $\mathsf{var}(v)$, $\mathsf{inp}(S, I)$, and $\mathsf{Tth\text{-}inp}(\tau, \iota)$.

to be inductive, i.e., its truth at a state S implies its truth at the next state $\mathsf{step}(S, I)$. Finally, it is shown to be a generalization of the intended safety property.

The composition of the finite step theorems in Figure 8.4 into the safety theorem is verified by a straightforward ACL2 proof by induction on τ. Furthermore, given a specific finite state machine model, each of the finite-step properties can be automatically transformed into SULFA formulas. In theory, any SULFA formula can be proven or disproven through the decision procedure in Chapter 5, and in practice many significant formulas can be proven or disproven entirely automatically using the SAT-based SULFA solver described in Chapter 7.

An augmentation of the basic safety property strategy is shown in Figure 8.5. The augmented strategy reduces a safety property into three finite step properties reducible to SULFA formulas, but includes additional variables and predicates. The property P may include an extra variable v, which is restricted into a finite domain by the predicate $\mathsf{var}(v)$. Furthermore, the predicate $\mathsf{Tth\text{-}inp}(\tau, \iota)$ is used to encode assumptions necessary on the machine inputs. This function is defined as:

$$\text{Tth-inp}(\tau, \iota) \triangleq \begin{cases} \textbf{false}, & \text{if } \neg\text{inp}(S_\tau, I_\tau) \\ \textbf{true}, & \text{if } \text{zp}(\tau) \\ \text{Tth-inp}(\tau - 1, \iota), & \text{otherwise.} \end{cases}$$

where the predicate $\text{inp}(S, I)$ returns whether the machine inputs I satisfy all the input assumptions required by inputs occurring in a cycle that begins with the machine state S, $S_\tau = \text{Tth-state}(\tau, \iota)$ is the state at the beginning of cycle number τ, and $I_\tau = \text{nth}(\tau, \iota)$ is the input during cycle number τ.

The augmented strategy is not strictly necessary for verifying safety properties. Since v is restricted to a finite domain, any property verified by the augmented strategy in Figure 8.5 can be reduced to a finite number of safety properties without v. Furthermore, by adding an extra bit to the machine state, the input assumptions Tth-inp can be encoded into the basic strategy.

The advantage of the augmented strategy is that it often leads to a more compact specification and proof. For example, in Section 8.5.2 the exception type is encoded in the variable v, allowing a single specification and proof to be used for two types of exceptions. Furthermore, by encoding the input assumptions in Tth-inp, no input assumptions need to be encoded into the invariant.

8.4.2 Verification of Liveness Properties

To verify the data-tile protocol, only a restricted form of liveness property is required. Using the same ACL2 model as used for safety properties, each liveness property can be expressed in the form of *Liveness-Theorem* in Figure 8.6. The restricted liveness theorem states that if some property P holds of the machine at time τ then eventually there will be a time τ' at which Q holds.

Our basic strategy for verifying liveness properties is shown in Figure 8.6. The

Liveness-Theorem :

$(\tau \in \mathbb{N}) \wedge \mathsf{P}(v, \text{Tth-state}(\tau, \iota), \text{nth}(\tau, \iota))$
$\rightarrow$
$(\exists \tau' : (\tau' \in \mathbb{N}) \wedge (\tau < \tau') \rightarrow \mathsf{Q}(v, \text{Tth-state}(\tau', \iota), \text{nth}(\tau', \iota)))$

follows from the following five finite-step theorems:

1. $\text{inv}(S_0)$
2. $\text{inv}(S) \rightarrow \text{inv}(\text{step}(S, I))$
3. $\text{inv}(\text{step}(S, I)) \wedge \mathsf{P}(S, I) \wedge \neg \mathsf{Q}(S, I) \rightarrow \text{bnd}(b_0, \text{step}(S, I))$
4. $(b \in \mathbb{N}) \wedge (0 \leq b < b_0) \wedge \text{bnd}(b + 1, S) \wedge \neg \mathsf{Q}(S, I) \rightarrow \text{bnd}(b, \text{step}(S, I))$
5. $\text{bnd}(0, \text{step}(S, I)) \rightarrow \mathsf{Q}(S, I)$

Figure 8.6: An outline of our basic strategy for verifying a type of liveness properties expressed as ACL2 theorems.

Augmented-Liveness-Theorem :

$(\tau \in \mathbb{N}) \wedge \text{var}(v) \wedge \mathsf{P}(v, \text{Tth-state}(\tau, \iota), \text{nth}(\tau, \iota))$
$\rightarrow$
$(\exists \tau' : (\tau' \in \mathbb{N}) \wedge (\tau < \tau') \wedge (\text{Tth-inp}(\tau', \iota) \rightarrow \mathsf{Q}(v, \text{Tth-state}(\tau', \iota), \text{nth}(\tau', \iota)))$

follows from the following five finite-step theorems:

1. $\text{var}(v) \rightarrow \text{inv}(v, S_0)$
2. $\text{var}(v) \wedge \text{inp}(S, I) \wedge \text{inv}(v, S) \rightarrow \text{inv}(v, \text{step}(S, I))$
3. $\text{var}(v) \wedge \text{inp}(S, I) \wedge \text{inv}(v, \text{step}(S, I)) \wedge \mathsf{P}(v, S, I) \wedge \neg \mathsf{Q}(v, S, I)$
   $\rightarrow$
   $\text{bnd}(v, b_0, \text{step}(S, I))$
4. $(b \in \mathbb{N}) \wedge (0 \leq b < b_0) \wedge \text{var}(v) \wedge \text{inp}(S, I) \wedge \text{bnd}(v, b + 1, S) \wedge \neg \mathsf{Q}(v, S, I)$
   $\rightarrow$
   $\text{bnd}(v, b, \text{step}(S, I))$
5. $\text{var}(v) \wedge \text{inp}(S, I) \wedge \text{bnd}(v, 0, \text{step}(S, I)) \rightarrow \mathsf{Q}(v, S, I)$

Figure 8.7: An augmentation of the basic strategy shown in Figure 8.6 for proving liveness properties. As in the augmented safety strategy, the augmented strategy includes an additional variable v and predicates $\text{var}(v)$, $\text{inp}(S, I)$ and $\text{Tth-inp}(\tau, \iota)$.

liveness theorem is proven by defining an inductive invariant $\mathsf{inv}(S)$ and a bound predicate $\mathsf{bnd}(b, S)$ satisfying the five finite-step theorems. The first two of the finite-step theorems show that $\mathsf{inv}(S)$ is an inductive invariant of the machine. The next three use the invariant to prove that P implies a bound b_0 on the number of cycles until Q holds.

Our basic strategy for verifying liveness properties is augmented in the same way as our safety property strategy is. The augmented strategy, with an additional variable v predicates $\mathsf{var}(v)$, $\mathsf{inp}(S, I)$ and $\mathsf{Tth\text{-}valid\text{-}ins}(\tau, \iota)$, is shown in Figure 8.7.

Using the ACL2 theorem prover, we have proven, by induction on τ and τ', that *Augmented-Liveness-Theorem* follows from the five finite-step theorems in Figure 8.7. Furthermore, if b_0 is a constant, as it is in the data-tile protocol, then for any specific finite state machine each of the five finite-step theorems in Figure 8.7 can be written as SULFA formulas.

8.5 Verification of the Exception Protocol

As described in Section 8.3, the exception protocol accumulates exceptions detected the the execution and data tiles. The formal specification of the protocol is a correspondence between the four DSN components and a single specification machine, shown in Figure 8.8. The specification machine outputs an exception mask that reports all exceptions that have occurred in all tiles (that have not yet been flushed), whereas the top DSN requires up to four cycles to report (to the global tile) an exception.

The specification is that if the top tile reports an exception, then that exception is also reported by the specification machine; and if the specification machine outputs an exception, then eventually either a flush will occur or the top tile will also report that exception. This is written in temporal logic as the following two properties:

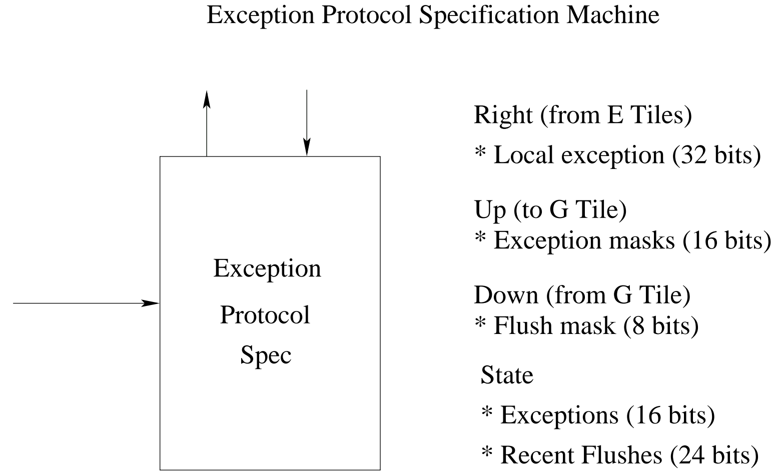


Figure 8.8: A high-level overview of the specification machine for the exception protocol.

Exception-Safety:

$$\square(\mathsf{dsn\text{-}ebit}\,(type, x, 0) \rightarrow \mathsf{spec\text{-}ebit}\,(type, x)).$$

Exception-Liveness:

$$\square(\mathsf{spec\text{-}ebit}\,(type, x) \rightarrow \Diamond(\mathsf{flush}\,(x) \vee \mathsf{dsn\text{-}ebit}\,(type, x, 0))).$$

where

- x is implicitly universally quantified over natural numbers from 0 to 7 inclusive.
- *type* is implicitly universally quantified over the set of exception types, {**miss**, **exec**}.
- $\mathsf{dsn\text{-}ebit}\,(type, x, n)$ is a predicate returning whether tile n of the DSN is reporting an exception of type *type* in instruction block x.
- $\mathsf{spec\text{-}ebit}\,(type, x)$ is a predicate returning whether the specification machine is reporting an exception of type *type* in instruction block x.
- $\mathsf{flush}\,(x)$ is a predicate returning whether the instruction block x is currently being flushed (in tile 0).

The following assumptions are also made about the inputs the exception protocol:

- **No Resets.** The DSN contains a reset input, which resets most of its registers. In our verification effort, however, we assume that the reset signal is off.
- **Properly Connected Wiring.** The predicate dsn-ebit $(type, x, n)$ implements the four DSNs as completely independent and unconnected, without the wiring shown in Figure 8.2. Instead, part of the input assumption is that the four DSN tiles are related in the same way as if they were connected as shown in Figure 8.2.
- **No Exception After Flush.** We also assume that no exception occurs in the cycle after a flush. This assumption is reasonable since exceptions are caused by executing instructions and multiple cycles must occur before a new block of instructions can be executed.

8.5.1 ACL2 Model of the Exception Protocol

The implementation of the exception protocol, its single-tile specification machine, and its input assumptions are modeled in ACL2 as an initial state constant, a step function, an input assumption predicate, and a function for each output. As in Section 8.4, given a cycle number τ and an input sequence ι, we refer to Tth-state (τ, ι) as the state at time τ and Tth-inp (τ, ι) as whether the input assumptions are met for all inputs in the sequence ι up to and including those during the cycle numbered τ. The following are the output functions and state accessor functions used in the following sections:

- dsn-ebit $(type, x, n, S, I)$ is the predicate returning whether an exception in instruction block x of type *type* is being reported by the DSN at tile n, given state S and machine inputs I.
- st-dsn-ebit $(type, x, n, S)$ is the predicate returning whether an exception of type *type* in instruction block x is currently being stored in tile n of the DSN, i.e., it was reported by tile n of the DSN in the previous cycle.

- spec-ebit $(type, x, S, I)$ is the predicate returning whether an exception in instruction block x of type *type* is being reported by the single tile specification machine, given state S and machine inputs I.

- st-spec-ebit $(type, x, S)$ is the predicate returning whether an exception of type *type* in instruction block x is currently being stored in the single-tile specification machine, i.e., it was reported by the specification machine in the previous cycle.

- flush (x, I) is the predicate returning whether the machine inputs I contains a flush of instruction block x (at tile 0).

- st-flush (x, n, S) is the predicate returning whether a flush of instruction block x occurred at tile n of the DSN in the previous cycle.

- st-will-flush (x, n, S) is the predicate returning whether a flush of instruction block x has occurred in one of the blocks above n in the previous cycle. Thus, a flush of x is "on its way" to tile n.

8.5.2 Proof of Exception-Safety

The Exception-Safety property can be translated into the following ACL2 property (note that I_τ and S_τ are abbreviations for terms, not universally quantified free variables like τ, ι, x, and *type*):

DSN-Exception-Safety:

$$(x \in \mathbb{N}) \land (x < 8) \land (\tau \in \mathbb{N}) \land (type \in \{\mathbf{miss}, \mathbf{exec}\})$$
$$\land \text{Tth-inp}(\tau, \iota) \land \text{dsn-ebit}(type, x, 0, S_\tau, I_\tau)$$
$$\rightarrow$$
$$\text{spec-ebit}(type, x, S_\tau, I_\tau)$$

where $S_\tau = \text{Tth-state}(\tau, \iota)$ is the state of the machine after τ clock cycles and $I_\tau = \text{nth}(\tau, \iota)$ is the input to the machine during clock cycle τ.

$\mathsf{var}(v) \triangleq (v_x \in \mathbb{N}) \wedge (v_x < 8) \wedge (v_{type} \in \{\mathbf{miss},\ \mathbf{exec}\})$

$\mathsf{P}(v, S, I) \triangleq \text{dsn-ebit}(v_{type}, v_x, 0, S, I) \rightarrow \text{spec-ebit}(v_{type}, v_x, S, I)$

$$
\begin{aligned}
&\quad \mathsf{inv}(v, S) \\
&\triangleq \\
&\quad (\forall n \in \mathbb{N}, n < 4 : \\
&\qquad \neg\text{st-will-flush}(v_x, n, S) \wedge \text{st-dsn-ebit}(v_{type}, v_x, n, S) \\
&\qquad \rightarrow \\
&\qquad \text{st-spec-ebit}(v_{type}, v_x, S))
\end{aligned}
$$

Figure 8.9: The definitions of the $\mathsf{var}(v)$, $\mathsf{P}(v, S, I)$, and $\mathsf{inv}(v, S)$ functions needed to verify the safety property of the exception protocol using the strategy illustrated in Figure 8.5. Note that v represents the pair $[v_x, v_{type}]$. Thus, v_x and v_{type} are abbreviations for the first and second element of v respectively.

The DSN-Exception-Safety property is verified using the strategy shown in Figure 8.5 with the definitions shown in Figure 8.9. Intuitively, the inductive invariant is that either a previous input was invalid or every exception of type *type* in instruction block x in the DSN is also in the specification machine. Each of the three properties in Figure 8.5 is verified entirely automatically through the SAT-based SULFA solver described in Chapter 7. They are then composed using the ACL2 theorem prover.

8.5.3 Proof of Exception-Liveness

The Exception-Liveness property can be translated into the ACL2 logic as the following first-order property (note that S_τ and I_τ are abbreviations for terms):

DSN-Exception-Liveness:

$$
\begin{aligned}
&(x \in \mathbb{N}) \wedge (x < 8) \wedge (\tau \in \mathbb{N}) \wedge (type \in \{\mathbf{miss}, \mathbf{exec}\}) \wedge \text{spec-ebit}(type, x, S_\tau, I_\tau) \\
&\rightarrow \\
&(\exists \tau' : \ (\tau' \in \mathbb{N}) \wedge (\tau < \tau') \wedge \\
&\qquad (\text{Tth-inp}(\tau, \iota) \rightarrow \text{flush}(x, I_\tau) \vee \text{dsn-ebit}(type, x, 0, S_\tau, I_\tau)))
\end{aligned}
$$

$\mathsf{var}(v) \triangleq (v_x \in \mathbb{N}) \wedge (v_x < 8) \wedge (v_{type} \in \{\textbf{miss}, \textbf{exec}\})$

$\mathsf{P}(v, S, I) \triangleq \mathsf{spec\text{-}ebit}(v_{type}, v_x, S, I)$

$\mathsf{Q}(v, S, I) \triangleq \mathsf{flush}(v_x, I) \vee \mathsf{dsn\text{-}ebit}(v_{type}, v_x, 0, S, I)$

$\mathsf{inv}(v, S) \triangleq \mathsf{st\text{-}spec\text{-}ebit}(v_{type}, v_x, S) \rightarrow \mathsf{bnd}(v, 3, S)$

$$\begin{aligned} &\mathsf{bnd}(v, b, S) \\ \triangleq\ & \begin{cases} \mathsf{bnd1}(v, 0, S), & \text{if } \mathsf{zp}(b) \\ \mathsf{bnd1}(v, b, S) \vee \mathsf{bnd}(v, b-1, S), & \text{otherwise.} \end{cases} \end{aligned}$$

where

$$\begin{aligned} &\mathsf{bnd1}(v, b, S) \\ \triangleq\ & \neg\mathsf{st\text{-}will\text{-}flush}(v_x, b, S) \wedge \neg\mathsf{st\text{-}flush}(v_x, 0, S) \wedge \mathsf{st\text{-}dsn\text{-}ebit}(v_{type}, v_x, b, S) \end{aligned}$$

Figure 8.10: The definitions of the $\mathsf{var}(v)$, $\mathsf{P}(v, S, I)$, $\mathsf{Q}(v, S, I)$ and $\mathsf{inv}(v, S)$ functions needed to verify the liveness property of the exception protocol using the strategy illustrated in Figure 8.7. Note that v represents the pair $[v_x, v_{type}]$. Thus, v_x and v_{type} are abbreviations for the first and second element of v respectively.

where $S_\tau = \mathsf{Tth\text{-}state}(\tau, \iota)$ is the state of the machine at the beginning of clock cycle τ and $I_\tau = \mathsf{nth}(\tau, \iota)$ is the input during clock cycle τ.

The DSN-Exception-Liveness theorem is proven using the strategy for liveness properties shown in Figure 8.7, given the definitions shown in Figure 8.10. The bound function, $\mathsf{bnd}(v, b, S)$, returns whether an exception of the type and instruction block denoted by v is being stored in one of the top b tiles. Therefore, unless a flush occurs, the exception will be reported by the top tile within b cycles. The inductive invariant, $\mathsf{inv}(v, S)$, is equivalent to the statement that if an exception is being stored in the specification machine, then it is also stored somewhere in the actual machine.

Given the definitions in Figure 8.10, each of the five properties in Figure 8.7 is proven entirely automatically by using our SAT-based SULFA solver from Chapter 7.

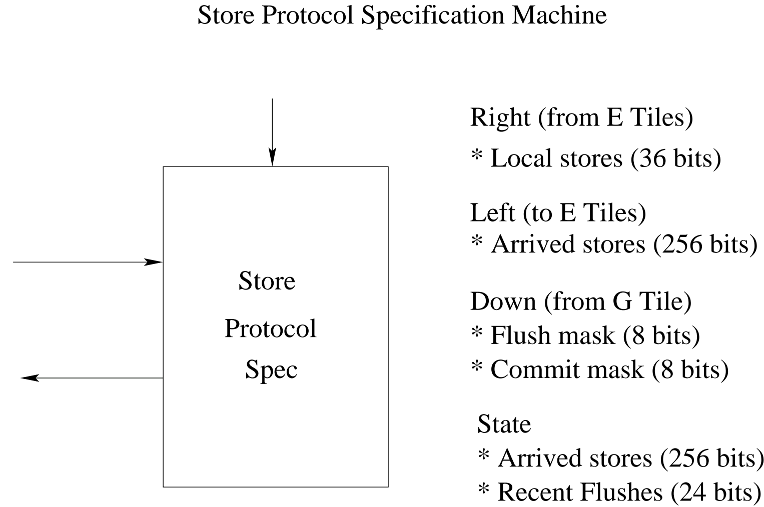


Figure 8.11: A high-level overview of the specification machine for the store protocol.

8.6 Verification of the Store Protocol

As described in Section 8.3, the store protocol accumulates information regarding which store instructions have arrived. The formal specification of the protocol is a correspondence between the four DSN components and a single specification machine, shown in Figure 8.11. The specification machine outputs a store mask that reports all stores that have occurred in all tiles (that have not yet been flushed), whereas a DSN component may require up to four cycles to report that a give store has arrived.

The specification of the store protocol is a correspondence between the four tile DSN and the single tile specification machine, expressed as the following temporal logic properties:

Store-Safety:

$$\Box(\text{dsn-sbit}\,(x,n) \wedge \neg\text{will-flush}\,(\lfloor \tfrac{x}{32} \rfloor, n) \wedge \neg\text{will-commit}\,(\lfloor \tfrac{x}{32} \rfloor, n) \rightarrow \text{spec-sbit}\,(x))$$

Store-Liveness:

$$\Box(\text{spec-sbit}\,(x) \rightarrow \Diamond(\text{flush}\,(\lfloor \tfrac{x}{32} \rfloor) \vee \text{commit}\,(\lfloor \tfrac{x}{32} \rfloor) \vee \text{dsn-sbit}\,(x,n)))$$

where

- x, which represents the instruction address of a store, is implicitly universally quantified over natural numbers from 0 to 255 inclusive.

- n, which represents a tile number, is implicitly universally quantified over natural numbers from 0 to 3 inclusive.

- dsn-sbit $(type, x, n)$ is the predicate returning whether tile n of the DSN is reporting that instruction x is an executed store instruction.

- spec-sbit (x) is the predicate returning whether the specification machine is reporting that the store x has been executed.

- flush (x) is the predicate returning whether the instruction block x is currently being flushed (in tile 0).

- will-flush (y, n) is the predicate returning whether a flush of instruction block y is occurring in a tile above tile n (i.e., it's on its way to tile n).

- commit (x) is the predicate returning whether the instruction block x is currently being committed (in tile 0).

- will-commit (y, n) is the predicate returning whether a commit of instruction block y is occurring in a tile above tile n

Intuitively, the safety property is that if any tile of the DSN illustrated in Figure 8.2 reports that a store has arrived (and its not about to be removed), then the specification machine illustrated in Figure 8.11 must also report that the store has arrived. The liveness property is that if the specification machine reports a store as having arrived, then each tile of the DSN eventually reports that the store has arrived (or the store is removed from the specification machine).

The following assumptions are also made about the inputs to the store protocol:

- **No Store Arrives Twice.** If a store is being reported by the specification machine, then no DSN input can state that it has again been executed. This is justified since if a store has been executed once, it cannot be executed again without a commit or a flush.

- **No Early Commits.** A commit of an instruction block cannot occur unless all stores reported by the specification machine have also been reported by the top tile of the DSN. This assumption is justified because the global tile will not send a commit until the top tile has reported that all stores in the instruction block have arrived.

  Similarly, no local store can arrive in an instruction block that has begun to be committed. For example, if tile 0 has committed instruction block 0, then no store can arrive in block 0 of tile 3 until after tile 3 has also committed it.

- **No Store After Flush.** We also assume that no store occurs in the cycle after a flush. This assumption is reasonable since multiple cycles must occur between the flush of an instruction block and when the first instruction is executed.

- **No Resets.** The DSN contains a reset input, which resets most of its registers. In our verification effort, however, we assume that the reset signal is off.

- **Properly Connected Wiring.** Our DSN machine is actually implemented as four DSN tiles, without the wiring shown in Figure 8.2. Instead, part of the input assumption is that the four DSN tiles are related in the same way as if they were connected as shown in Figure 8.2.

8.6.1 ACL2 Model of the Store Protocol

The implementation of the store protocol, its single-tile specification machine, and its input assumptions are modeled in ACL2 as an initial state constant, a step function, an input assumption predicate, and a function for each output. To coincide with the strategy outlined in Section 8.4, given a cycle number τ and an input sequence ι, we refer to Tth-state (τ, ι) as

the state at time τ and $\mathsf{Tth\text{-}inp}(\tau, \iota)$ as whether the input assumptions are met for all inputs in the sequence ι up to and including those during the cycle numbered τ. The following are the output functions and state accessor functions used in the following sections:

- $\mathsf{dsn\text{-}sbit}(x, n, S, I)$ is the predicate returning whether tile n of the DSN is reporting that a store at instruction address x has been executed, given state S and machine inputs I.

- $\mathsf{st\text{-}dsn\text{-}sbit}(x, n, S)$ is the predicate returning whether the state of tile n of the DSN includes the knowledge that a store at instruction address x has been executed, i.e., it was reported by tile n of the DSN in the previous cycle.

- $\mathsf{spec\text{-}sbit}(x, S, I)$ is the predicate returning whether a store at instruction address x is being reported by the single tile specification machine, given state S and machine inputs I.

- $\mathsf{st\text{-}spec\text{-}sbit}(x, S)$ is the predicate returning whether the state of the single tile specification machine includes the knowledge that a store at instruction address x has been executed, i.e., it was reported by the specification machine in the previous cycle.

- $\mathsf{flush}(y, I)$ is the predicate returning whether the machine inputs I contains a flush of instruction block y (at tile 0).

- $\mathsf{st\text{-}flush}(y, n, S)$ is the predicate returning whether a flush of instruction block y occurred at tile n of the DSN in the previous cycle.

- $\mathsf{st\text{-}will\text{-}flush}(y, n, S)$ is the predicate returning whether a flush of instruction block y is occurring in one of the blocks above n in the previous cycle. Thus, a flush of y is on its way to tile n.

- $\mathsf{commit}(y, I)$ is the predicate returning whether the machine inputs I contains a commit of instruction block y (at tile 0).

- $\mathsf{st\text{-}will\text{-}commit}(y, n, S)$ is the predicate returning whether a commit of instruction block y is occurring in one of the blocks above n in the previous cycle. Thus, a commit of y is on its way to tile n.

8.6.2 Proof of Store-Safety

The Store-Safety property can be translated into the ACL2 logic as the following first order property (note that I_τ and S_τ are abbreviations for terms, not universally quantified free variables):

DSN-Store-Safety:

$$
\begin{aligned}
&(x \in \mathbb{N}) \wedge (x < 256) \wedge (\tau \in \mathbb{N}) \wedge (n \in \mathbb{N}) \wedge (n < 4) \\
&\wedge \mathsf{Tth\text{-}inp}(\tau, \iota) \wedge \mathsf{dsn\text{-}sbit}(x, n, S_\tau, I_\tau) \\
&\wedge \neg\mathsf{st\text{-}will\text{-}flush}(\lfloor \tfrac{x}{32} \rfloor, n, S_\tau) \wedge \neg\mathsf{st\text{-}will\text{-}commit}(\lfloor \tfrac{x}{32} \rfloor, n, S_\tau) \\
&\rightarrow \\
&\mathsf{spec\text{-}sbit}(x, n, S_\tau, I_\tau)
\end{aligned}
$$

where $S_\tau = \mathsf{Tth\text{-}state}(\tau, \iota)$ is the state of the machine after τ clock cycles and $I_\tau = \mathsf{nth}(\tau, \iota)$ is the input to the machine during clock cycle τ.

The DSN-Store-Safety property is verified using the strategy shown in Figure 8.5 with the definitions outlined in Figure 8.12. Without going into too much detail, the invariant is a conjunction of the following:

1. **The safety property.** The first clause in the conjunction is just a rewriting of the store protocol safety property into a predicate on state.

2. **Each store in the channels is in the specification.** All stores in communication channels must also be reported as arrived by the specification machine. This is necessary since such stores are about to be reported by the tile receiving the communication.

$\mathsf{var}(v) \triangleq (v_x \in \mathbb{N}) \wedge (v_x < 256) \wedge (n \in \mathbb{N}) \wedge (n < 4)$

$\mathsf{P}(v, S, I) \triangleq \mathsf{dsn\text{-}sbit}(v_x, n, S, I) \rightarrow \mathsf{spec\text{-}sbit}(v_x, S, I)$

$$
\begin{aligned}
&\mathsf{inv}(v, S) \\
\triangleq \\
&(\forall n' \in \mathbb{N}, n' < 4 : \\
&\quad \neg\mathsf{st\text{-}will\text{-}commit}(\lfloor \tfrac{v_x}{32} \rfloor, n', S) \wedge \neg\mathsf{st\text{-}will\text{-}flush}(\lfloor \tfrac{v_x}{32} \rfloor, n', S) \\
&\quad \wedge (\mathsf{st\text{-}dsn\text{-}sbit}(v_x, n', S) \vee \mathsf{in\text{-}channel}(v_x, n', S)) \\
&\quad \rightarrow \\
&\quad \mathsf{st\text{-}spec\text{-}sbit}(v_x, S)) \\
&\wedge \\
&\mathsf{in\text{-}channel\text{-}implies\text{-}in\text{-}spec}(v_x, S) \\
&\wedge \\
&\mathsf{no\text{-}committed\text{-}stores\text{-}in\text{-}up\text{-}channel}(v_x, S) \\
&\wedge \\
&\mathsf{in\text{-}up\text{-}channel\text{-}implies\text{-}not\text{-}above}(v_x, S)
\end{aligned}
$$

Figure 8.12: An outline of the definitions needed to verify the store protocol safety property using the strategy illustrated in Figure 8.5. Note that v represents the pair $[v_x, v_{type}]$. Thus, v_x and v_{type} are abbreviations for the first and second element of v respectively.

3. **No committed stores are communicated upward.** If stores in an instruction block are in the process of being communicated to the top tile, then the instruction block cannot be committed because the top tile could not have detected that all stores in the block have been executed. This invariant is needed because, unlike with a flushed store, the protocol design contains no extra logic to ensure that a committed store being communicated upwards is properly removed.

4. **If a store is being communicated upward, then it is not already known in the tiles above.** If a store is in one of the upward communication channels, then, since no store can arrive twice, it cannot be in any of the upward communication channels in the tiles above or in the arrived mask of the top tile. This clause is required, since our input assumptions regarding committing and duplicate stores do not directly involve the communication channels and therefore cannot alone imply that no committed stores are in the upward communication channels.

Given the above invariant, each of the three properties in Figure 8.5 can be verified through a mixture of user-guided theorem proving and the SAT-based SULFA solver described in Chapter 7. Only a small amount of user guidance and theorem proving is necessary, because each of the three properties in Figure 8.5 can be proven automatically by the SAT-based SULFA solver for any given store x. Thus, the proofs of the three properties involve a case split on the 256 possible stores, followed by 256 calls to the SAT-based SULFA solver.

8.6.3 Proof of Store-Liveness

The store protocol liveness property can be translated into the ACL2 logic as the following first-order property (note that S_τ, I_τ, and $I_{\tau+1}$, are abbreviations for terms):

$\mathsf{var}\,(v) \triangleq (v_x \in \mathbb{N}) \wedge (v_x < 256) \wedge (n \in \mathbb{N}) \wedge (n < 4)$

$\mathsf{P}\,(v, S, I) \triangleq \mathsf{spec\text{-}sbit}\,(v_x, \mathsf{step}\,(S, I), I')$

$\mathsf{Q}\,(v, S, I) \triangleq \mathsf{flush}\,(v_x, I) \vee \mathsf{dsn\text{-}sbit}\,(v_x, n, \mathsf{step}\,(S, I), I')$

$\mathsf{inv}\,(v, S) \triangleq \mathsf{bnd\text{-}live\text{-}inv}\,(v_x, S) \wedge (\mathsf{st\text{-}spec\text{-}sbit}\,(v_x, S) \rightarrow \mathsf{within}\,(v_x, n, b, S))$

$\mathsf{bnd}\,(v, b, S) \triangleq \mathsf{bnd\text{-}live\text{-}inv}\,(v_x, S) \wedge \mathsf{within}\,(v, b, S)$

Figure 8.13: An outline of the definitions needed to verify the store protocol safety property using the strategy illustrated in Figure 8.7. Note that v represents the pair $[v_x, v_{type}]$. Thus, v_x and v_{type} are abbreviations for the first and second element of v and respectively.

DSN-Store-Liveness:

$$
\begin{aligned}
&(x \in \mathbb{N}) \wedge (x < 256) \wedge (\tau \in \mathbb{N}) \wedge (n \in \mathbb{N}) \wedge (n < 4) \wedge \mathsf{spec\text{-}sbit}\,(x, \mathsf{step}\,(S_\tau, I_\tau), I_{\tau+1}) \\
&\rightarrow \\
&(\exists \tau' :\; (\tau' \in \mathbb{N}) \wedge (\tau < \tau') \wedge \\
&\qquad (\mathsf{Tth\text{-}inp}\,(\tau + 1, \iota) \rightarrow \mathsf{flush}\,(\lfloor \tfrac{x}{32} \rfloor, I_\tau) \vee \mathsf{dsn\text{-}sbit}\,(x, n, \mathsf{step}\,(S_\tau, I_\tau), I_{\tau+1})))
\end{aligned}
$$

where $S_\tau = \mathsf{Tth\text{-}state}\,(\tau, \iota)$ is the state of the machine at the beginning of clock cycle τ and $I_\tau = \mathsf{nth}\,(\tau, \iota)$ is the input during clock cycle τ. The reason for using both I_τ and $I_{\tau+1}$ is that the store mask is the output of a register, which is affected only by inputs in the previous cycle, as shown in Figure 8.11. Thus, a flush of instruction block y in inputs I_τ does not affect $\mathsf{dsn\text{-}sbit}\,(x, n, S_\tau, I_\tau)$, but may affect $\mathsf{dsn\text{-}sbit}\,(x, n, \mathsf{step}\,(S_\tau, I_\tau), I_{\tau+1})$.

Figure 8.13 outlines the functions that map our liveness proof strategy, from Figure 8.7, to a proof of the DSN-Store-Liveness theorem. The function $\mathsf{within}\,(x, n, b, S)$ is defined, roughly, as the predicate returning whether x has either arrived at tile n or is in one of the communication channels headed towards n and within b tiles of n. The predicate $\mathsf{bnd\text{-}live\text{-}inv}\,(x, S)$ is an inductive invariant stating that if x is being flushed in some tile n, then x is not in the communication channel going from tile $n-1$ to n. This is necessary since $\mathsf{within}\,(x, n, 1, S)$ is true if a store is in the channel from $n-1$ to n; however, such a store would be removed if it came the cycle after a flush.

The liveness proof strategy in Figure 8.7 thus reduces DSN-Store-Liveness into five finite-step properties. As with DSN-Store-Safety, the ACL2 theorem prover is used to split each of the five finite-step properties into 256 possible values of x. Then, each property is proven for that specific value of x automatically, by using the SAT-based SULFA solver described in Chapter 7.

8.7 Analysis

The formal verification of the exception and store protocols found no bugs in the protocol designs or their implementation. The verification effort, however, provides a higher degree of assurance than would be possible with simulation. We believe that all bugs found by simulation would have also been found by formal verification and that formal verification also rules out any hidden corner cases. For example, the initial software model of the DSN removed only exceptions and stores flushed in the current cycle, not those flushed in the previous cycle. This leads to a bug when a flush occurs while an exception or store is being communicated upward. This case was only found after considerable testing of the software model. However, given a circuit containing this bug, the SAT-based SULFA solver produces a counterexample when attempting to prove the safety invariant.

Furthermore, formal verification of the DSN clarifies its input assumptions, which is helpful for designing other blocks. For example, it was believed before this verification effort began that the DSN required that no stores arrive for three cycles after an instruction block is committed (in tile 0). However, no such input assumption is required. Now that all input assumptions are clearly specified, units that interact with the DSN can be developed for maximum efficiency without any fear of violating subtle correctness assumptions.

We believe that the SAT-based SULFA solver from Chapter 7 greatly reduced the amount of human effort required to verify the exception and store protocols using ACL2. Without the SAT-based SULFA solver, a significant amount of user guidance would be required to prove each of the finite step properties that were proven automatically by the

SAT-based SULFA solver. Some evidence of this is provided by our initial proof of the liveness of the store protocol, which was proven using an earlier and less efficient SAT-based SULFA solver and before we learned to case-split on the store address to reduce the state space. While still making some use of the SAT-based SULFA solver, our early effort required 152 definitions and lemmas, along with weeks of human effort to create. The current proof is far simpler, requiring only 21 definitions and lemmas to prove the same property.

The greatest decrease in human effort comes from the generation of counterexamples to failed proofs. Almost all of the inductive invariants and bound functions described in this chapter were initially incorrect and had to be debugged by counterexamples and failed proofs. Furthermore, over ten bugs in the initial specification—the input assumptions, top-level DSN output functions, and single tile model—were discovered. Using the ACL2 theorem prover alone, each of these bugs in the specification and proof would have been discovered only after careful inspection of failed proofs. The SULFA solver from Chapter 7, however, produces a mapping from variables to values under which an invalid formula evaluates to false. We believe this is usually far easier to understand than the output of a failed proof. A counterexample tells the user immediately that the formula is invalid, not just that it could not be proven. Plus, the Lisp interpreter and its debugging mechanisms (such as trace) provide a reasonable environment for determining the underlying causes behind each counterexample.

The fully-automatic verification of SULFA formulas promotes reusability. The user guidance within theorem proving is often specific to the internals of a specific design. Similarly, simulation often requires illustrative test suites designed with knowledge of the design's internals. Formal verification, when making use of fully-automated techniques, however, requires no knowledge of the design's internals. Thus, hard to follow low-level optimizations can be attempted late in the design process. If the formal verification succeeds when rerun, then the optimizations can be trusted.

8.8 Summary

The SAT-based procedure for SULFA formulas described in Chapter 7 has been applied to the formal verification of components of the TRIPS processor. Both safety and liveness properties were proven about models extracted automatically from the actual Verilog implementations.

The component verified is a novel component of the TRIPS processor, a protocol required to implement its decentralized memory system. The decentralized memory system is part of the TRIPS processor's unique EDGE architecture, which provides the opportunity to increase thread-level and data-level parallelism within a single processor design.

Our approach uses the theorem prover to reduce safety and liveness properties of hardware designs into finite step problems in the SULFA decidable subclass. The full power of the ACL2 theorem prover then remains available at all times, but automatic approaches are also applicable. During the verification of the exception and store protocols we feel that we were able to prove many formulas fully automatically that would otherwise have required significant further guidance. Our approach speeds up debugging of failed proofs by providing counterexamples to invalid theorems, and promotes reusability by avoiding the need to consider many internal design details. We feel that the SAT-based procedure from Chapter 7 can substantially reduce the human guidance required by hardware verification efforts with the ACL2 theorem prover.

8.9 Development and Bibliographic Notes

The EDGE architecture and the TRIPS processor were developed through a joint effort between many researchers at the University of Texas [9, 73]. The data tile of the TRIPS processor, including its communications protocols was primarily designed by Simha Sethumadhavan. I wrote the initial Verilog implementation of the DSN component of the data tile.

ACL2 and its predecessor Boyer-Moore theorem provers have been used extensively for hardware verification. A simple pipelined processor, the FM8501, was proven to satisfy its ISA specification by Hunt in 1985 [27]. A more complex model of an out-of-order pipelined processor, the FM9801, was verified in 1998 [75]. At AMD, the floating-point unit on the AMDK586 and the floating-point unit on the Athlon™K7 were verified with ACL2 [54, 72]. Also, the implementation of Rockwell Collin's AAMP7 separation kernel, the unit that keeps critical and non-critical processes from interfering with each other, was verified with ACL2 [21]. The AAMP7 processor is an industrial processor used for safety-critical applications. To our knowledge, however, none of the above efforts make extensive use of fully-automated techniques.

Outside the ACL2 community, there has been considerable interest in applying a combination of theorem proving and model checking to verify significant hardware systems. At Intel, the FORTE verification tool has been used to verify the floating-point unit of the Pentium® 4 as well as other components, such as the branch-target buffer and the instruction-length decoder [1, 78]. FORTE uses a fully-automated technique, STE, to verify finite-step properties, along with a lightweight HOL theorem prover to compose them.

The SMV model checker includes some compositional theorem proving capabilities [49], which has been used successfully to verify significant hardware designs, such as the cache coherence protocol on the FLASH processor [50] and to verify an implementation of Tomasulo's out-of-order scheduling algorithm [48].

The entire implementation of the VAMP processor, an out-of-order pipelined processor, has also recently been verified using the PVS theorem prover [6]. To our knowledge, however, no inter-tile communication protocol, like the one described in this chapter, has ever been formally verified previously.

One ACL2 verification effort that did make extensive use of fully-automated techniques, was the verification of an out-of-order, pipelined processor model by Manolios and Srinivasan using ACL2 and UCLID [44]. This effort differs from our own in a number

of substantial ways. While Manolios and Srinivasan verified a much larger portion of the processor design, the processor they verified is not nearly as novel or complex as that of the TRIPS processor. Also, the type of properties verifiable by their technique differs from SULFA in interesting ways. SULFA includes ACL2 formulas unrollable into a restricted set of ACL2 primitives, where variables are unified over the full domain of ACL2 objects. The UCLID integration proves ACL2 formulas unrollable into a set of defined ACL2 functions that are equivalent UCLID primitives, where variables must be restricted into either the boolean or integer domains recognized by UCLID.

Chapter 9

The SULFA SMT Solver

9.1 Introduction

A Satisfiability Modulo Theory (SMT) is a standard theory for which fully-automated procedures exist to determine whether formulas are satisfiable. Developing and standardizing SMT theories is an active area of interest in the verification research community. Standard SMT theories have been developed for theories including linear arithmetic, arrays, uninterpreted functions, and bit vectors. Databases of problems in SMT theories are being developed and annual competitions are being held to determine the most efficient and effective procedure for each standard theory [67, 84].

We have applied the SULFA procedure, along with the ACL2 theorem prover, to develop an SMT solver for the Quantifier-Free Theory of Uninterpreted Functions and Bit Vectors of up to 32 bit width (QF_UFBV32). The SULFA SMT solver can solve all 8,246 of the problems in the 2006 QF_UFBV32 benchmark suite. Furthermore, it has a uniquely high degree of flexibility due to its embedding within the ACL2 theorem prover.

This chapter begins with an introductory example in Section 9.2, which shows how an SMT bit-vector problem can be solved through a mixture of theorem proving and the SAT-based SULFA solver described in Chapter 7. Section 9.3 then describes the SMT

2006 bit-vector theory in more detail. Next, Section 9.4 outlines the implementation of the SULFA SMT solver and Section 9.5 describes the unique feature of the SULFA SMT solver, that it can be extended in a sound and verifiable way with new simplification strategies and bit-vector primitives. Finally, Section 9.6 describes the verification of the benchmark suite in more detail before concluding with a summary and bibliographic notes.

9.2 Introductory Example

We begin with an example to introduce the SMT bit-vector theory and show how it can be modeled and solved using the ACL2 theorem prover and our SAT based procedure for SULFA formulas. An SMT solver determines whether a given formula, such as the one below, is satisfiable:

SMT1:

$$\mathsf{bvnot}\,(\mathsf{bvnot}\,(\mathsf{bvnot}\,(\mathsf{uf0}\,()))) = \mathsf{uf1}\,()$$

where $\mathsf{bvnot}\,(x)$ is bitwise negation and $\mathsf{uf0}\,()$ and $\mathsf{uf1}\,()$ are uninterpreted functions declared to return 4-bit, bit vectors. In the SMT language, all functions other than bit vector primitives are uninterpreted, functions, which are constrained only to input and output bit vectors of some finite declared width.

The goal is to determine whether there exist interpretations of the functions $\mathsf{uf0}\,()$ and $\mathsf{uf1}\,()$ returning 4-bit, bit vectors such that *SMT1* is always satisfied. A typical SMT solver begins by performing simplification based on the definitions of the bit-vector primitives. In this case, reducing $\mathsf{bvnot}\,(\mathsf{bvnot}\,(x))$ to x leads to:

$$\mathsf{bvnot}\,(\mathsf{uf0}\,()) = \mathsf{uf1}\,()$$

The above formula may then be shown to be satisfiable (e.g., in binary, $\mathsf{uf0}\,() \triangleq 0000$ and $\mathsf{uf1}\,() \triangleq 1111$) by using SAT solvers, BDDs, or some other fully-automated technique.

In order to model the ACL2 logic in the SMT language, we must first choose a bit-vector representation that maps ACL2 objects to bit vectors. Different mappings can be used to create such a model. For now, let the ith bit of an ACL2 bit vector x be high if $\mathsf{nth}\,(n-i-1, x) = \mathbf{true}$. Thus a bit vector represented in Boolean as "1110" can be represented in ACL2 as the list $[\mathbf{true}, \mathbf{true}, \mathbf{true}, \mathbf{false}]$. The bit vector "1110" can also be represented as the less intuitive $[\mathbf{true}, \mathbf{true}, \mathbf{true}]$, since $\mathsf{nth}\,(n, x) = \mathbf{false}$ if x is a list of less than n elements. In this representation, the number of bits in a bit vector cannot be determined from its value. Thus, the size of the input bit vectors are added as an input to each bit-vector primitive. This leads to the following ACL2 representation of the formula *SMT1*, which is valid if and only if *SMT1* is unsatisfiable:

ACL2-SMT1:

$$\neg\mathsf{nbveq}\,(4, \mathsf{nbvNot}\,(4, \mathsf{nbvNot}\,(4, \mathsf{nbvNot}\,(4, \mathsf{uf0}\,()))), \mathsf{uf1}\,()))$$

where $\mathsf{nbveq}\,(n, x, y) \triangleq \bigwedge_{i\in\mathbb{N}, 0\leq i<n} \mathsf{nth}\,(i, x) \leftrightarrow \mathsf{nth}\,(i, y)$, and $\mathsf{nbvNot}\,(n, x)$ is the n element list whose ith element is $\neg\mathsf{nth}\,(i, x)$. The functions $\mathsf{uf0}\,()$ and $\mathsf{uf1}\,()$ are uninterpreted functions.

ACL2-SMT1 is in SULFA because, given a constant bit width, each call to a bit-vector primitive can be unrolled into an expression involving only propositional logic and list primitives. Rather than solving the problem directly using the SAT-based procedure, however, it is preferable to let the ACL2 rewriter perform some simplification first. For example, the following theorem, which can be proven with the ACL2 theorem prover instructs the rewriter to remove double negation:

NBVNOT-NOT :

$$\mathsf{nbvNot}\,(n, \mathsf{nbvNot}\,(n, x)) = \mathsf{nbvfix}\,(n, x)$$

where $\mathsf{nbvfix}\,(n, x)$ maps x into a list of Booleans of length n representing the same bit vector as x. Since $\mathsf{nbvfix}\,(n, x)$ returns an equivalent bit vector to x, the following is also a theorem:

NBVNOT-FIX :

$$\mathsf{nbvNot}\,(n, \mathsf{nbvfix}\,(n, x)) = \mathsf{nbvNot}\,(n, x)$$

Thus, using the ACL2 rewriter *ACL2-SMT1*, can be simplified in the same manner as *SMT1*, which leads to the formula:

$$\neg\mathsf{nbveq}\,(4, \mathsf{nbvNot}\,(4, \mathsf{uf0}\,(), \mathsf{uf1}\,())).$$

The above formula can be then passed to the SAT-based procedure described in Chapter 7. The SAT-based procedure finds the following counter example to the above formula

$$\mathsf{uf0}\,() \triangleq [\mathbf{true}, \mathbf{false}, \mathbf{false}, \mathbf{true}]$$

$$\mathsf{uf1}\,() \triangleq [\mathbf{false}, \mathbf{true}, \mathbf{true}, \mathbf{false}]$$

Thus *ACL2-SMT1* is invalid.

Note that the process for proving *ACL2-SMT1* invalid follows the same flow as would likely be used by an SMT solver to find that *SMT1* is satisfiable. The advantage of our approach, however, is that the bit-vector primitives are ACL2 functions and the rewrite rules can be verified. This makes it easier to create new bit-vector primitives and simplification strategies.

9.3 The Standard Bit-Vector Theory

This section provides a more detailed description of the SMT 2006 Quantifier-Free Theory with Uninterpreted Functions and Bit Vectors up to 32 bits (QF_UFBV32). QF_UFBV32 provides a standard interface for comparing automated verification strategies on bit vectors and writing higher-level tools that reduce software and hardware verification to bit-vector verification problems. Formulas in the SMT bit-vector theory consist of constants, applications of primitive functions and predicates, and applications of uninterpreted functions and predicates. A constant is a natural number, a bit vector of width up to 32 bits, or a

Table 9.1: SMT Bit-Vector Functions

SMT Function	ACL2 Function	Description
extract $(j,k)(x)$	ebvEx (n_x, j, k, x)	Extract bits j through k of x
fill $(j)(x)$	ebvRepeat $(1, x, j)$	Create j copies of x
concat (x, y)	ebvConcat (n_x, n_y, x, y)	Concatenation
bvnot (x)	ebvNot (n_x, x)	Bitwise negation
bvand (x, y)	ebvAnd (n_x, x, y)	Bitwise conjunction
bvor (x, y)	ebvOr (n_x, x, y)	Bitwise disjunction
bvxor (x, y)	ebvXor (n_x, x, y)	Bitwise exclusive or
bvadd (x, y)	ebvAdd (n_x, x, y)	$x + y \mod 2^{n_x}$
bvneg (x)	ebvNeg (n_x, x)	$-x$
bvsub (x, y)	ebvSub (n_x, x, y)	$x - y \mod 2^{n_x}$
bvmul (x, y)	ebvMul (n_x, x, y)	$x * y \mod 2^{n_x}$
bvnand (x, y)	ebvNand (n_x, x, y)	Negated bitwise conjunction
bvnor (x, y)	ebvNor (n_x, x, y)	Negated bitwise disjunction
bvshift_left0 (x, j)	ebvShiftLeft0 (n_x, x, j)	Shift left j bits, filling with zeros
bvshift_left1 (x, j)	ebvShiftLeft1 (n_x, x, j)	Shift left j bits, filling with ones
bvshift_right0 (x, j)	ebvShiftRight0 (n_x, x, j)	Shift right j bits, filling with zeros
bvshift_right1 (x, j)	ebvShiftRight1 (n_x, x, j)	Shift right j bits, filling with ones
bvrepeat (x, j)	ebvRepeat (n_x, x, j)	Create j copies of x
sign_extend (x, j)	ebvSignExtend (n_x, x, j)	Extend with j copies of the sign bit
rotate_left (x, j)	ebvRotateLeft (n_x, x, j)	Rotate left j bits
rotate_right (x, j)	ebvRotateRight (n_x, x, j)	Rotate right j bits
ite (a, x, y)	ebvIte (n_x, a, x, y)	If a then x else y

Boolean. Uninterpreted functions are strongly typed to input and output bit vectors of a specific, known width.

Tables 9.1 and 9.2 show the primitive functions and predicates in QF_UFBV32. The bit-vector functions and predicates implement standard bit-vector operations, such as conjunction, disjunction, summation, subtraction, multiplication, equality and less-than. Note that some common bit-vector operations like division and exponentiation are not included in the SMT 2006 language.

Each primitive in Tables 9.1 and 9.2 input and output Booleans and bit vectors. The tables informally include type descriptions by using the convention that bit-vector variables

Table 9.2: SMT Bit-Vector Predicates and Logical Operators

SMT	ACL2	Description
$x = y$	$x =_{\text{ebv}} y$	bit-vector equivalence
bvlt (x, y)	ebvLt (n_x, x, y)	Unsigned less than
bvleq (x, y)	ebvLeq (n_x, y, x)	Unsigned less than or equal to
bvgeq (x, y)	ebvGeq (n_x, x, y)	Unsigned greater than or equal to
bvgt (x, y)	ebvGt (n_x, y, x)	Unsigned greater than
bvslt (x, y)	ebvSlt (n_x, x, y)	Signed less than
bvsleq (x, y)	ebvSleq (n_x, y, x)	Signed less than or equal to
bvsgt (x, y)	ebvSgt (n_x, y, x)	Signed greater than
bvsgeq (x, y)	ebvSgeq (n_x, x, y)	Signed greater than or equal to
if_then_else (a, b, c)	eif (a, b, c)	If a then b else c
not (a)	enot (a)	Logical negation
implies (a, b)	eimplies (a, b)	Implication
and $(a_1, a_2, ..., a_k)$	eand $(a_1, a_2, ..., a_k)$	Conjunction
or $(a_1, a_2, ..., a_k)$	eor $(a_1, a_2, ..., a_k)$	Disjunction
xor $(a_1, a_2, ..., a_k)$	exor $(a_1, a_2, ..., a_k)$	Exclusive or
iff (a, b)	eiff (a, b)	Boolean equivalence

are named x, y, or z; Boolean variables are named a, b, or c; and natural number variables are named j or k. The variables n_x and n_y also represent natural numbers, but they only are used in the ACL2 functions, and are therefore not part of the SMT language description. For a precise description of the types of each of these primitives, see the SMT-Lib web page [67]. Also, on the SMT-Lib web page is a precise description of the SMT syntax, which is similar to Common Lisp.

The representation of bit vectors in the SMT language is left to each implementation to define. In our ACL2 representation, a bit vector x is represented as a pair $[n_x, v_x]$ where the size of x is n_x and v_x is a list representing the Boolean values of the bits. The top-level ACL2 functions used to model the SMT bit-vector primitives are also shown in Tables 9.1 and 9.2. These functions also often input the bit-vector width of one or more of their inputs.

9.4 Implementation

The design of the SULFA SMT solver is outlined in Figure 9.1. The SMT solver is divided into six phases, each of which manipulates the SMT problem before passing it to the next phase. Each phase is entirely automatic and most, since they manipulate ACL2 formulas, have the potential to be verified using the ACL2 theorem prover. The following subsections describe each phase and follow how that phase manipulates an example.

9.4.1 Syntactic Translation and Negation

The functions and predicates in Tables 9.1 and 9.2 have been formalized as ACL2 functions. Thus, we have developed a shallow embedding of the SMT language within the ACL2 logic. We refer to the top-level ACL2 functions as the *Embedded Type ACL2* (ET-ACL2) model, since bit vectors are represented as a pair that includes their width. Thus, unlike the representation in Figure 9.2, the type information is embedded into the bit-vector representation.

The reason for embedding type information into the data, despite the fact that it is also input into most functions directly, is that we do not want to input the bit-vector size to the bit-vector equivalence function $=_{\mathrm{ebv}}$. This allows bit-vector equivalence to be a binary equivalence relation, which is important during the ACL2 simplification phase.

Thus, the first step in solving a given SMT problem is to translate it into the ET-ACL2 model. The translation is performed by a mixture of program-mode ACL2 functions and a program written in C. The formula is negated to translate it from a satisfiability problem to a validity problem. For example, consider the following SMT formula:

$$
\begin{aligned}
&\mathit{SMT2}: \\
&\quad \mathsf{uf}\,(\mathbf{0}_{\mathrm{bv4}}) \\
&\quad = \\
&\quad \mathsf{bvand}\,(\,\mathsf{bvor}\,(\mathbf{4}_{\mathrm{bv4}}, \mathsf{bvand}\,(\mathsf{bvand}\,(\mathsf{uf}\,(\mathbf{0}_{\mathrm{bv4}}), \mathsf{uf}\,(\mathbf{1}_{\mathrm{bv4}})), \mathsf{bvnot}\,(\mathsf{uf}\,(\mathbf{0}_{\mathrm{bv4}})))), \\
&\qquad\qquad \mathsf{bvand}\,(\mathbf{5}_{\mathrm{bv4}}, \mathsf{uf}\,(\mathbf{0}_{\mathrm{bv4}})))
\end{aligned}
$$

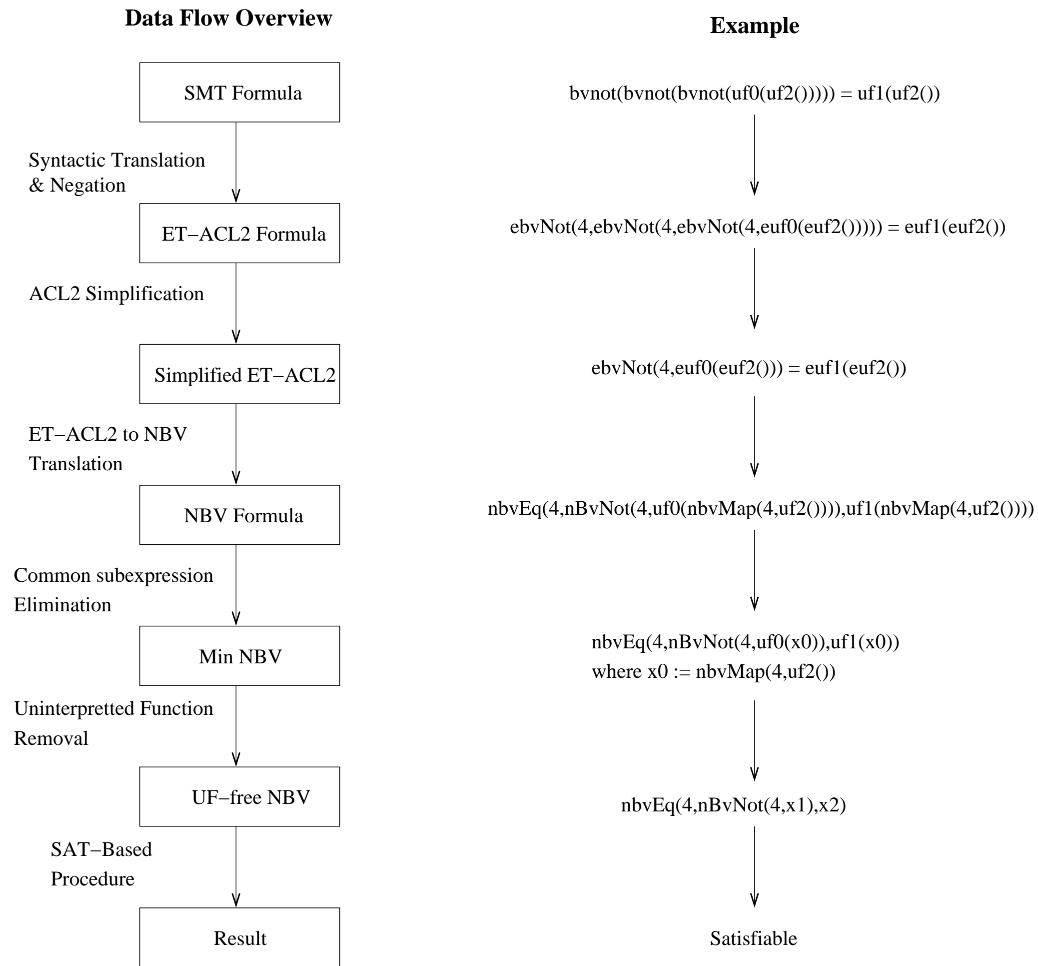


Figure 9.1: An overview of the data flow within the general-purpose SMT solver is shown on the left. On the right is an example formula, similar to the one in Section 9.2, as it is represented in each data flow stage. In the example SMT formula, uf0 (), uf1 (), and uf2 () are each declared to be uninterpreted functions returning 4-bit, bit vectors.

where J_{bv4} is the 4-bit, bit vector representing the natural number J, i.e., the ith bit of J_{bv4} is high if $\lfloor \frac{J}{2^i} \rfloor \mod 2 = 1$. For example, 1_{bv4} is 0001 (where the right most bit is the 0th bit) Also, the function $\mathsf{uf}(x)$ is an uninterpreted functions inputing and outputting a 4-bit, bit vector.

Intuitively, *SMT2* is the formula $x = (4 \mid x \;\&\; y \;\&\; \overline{x}) \;\&\; 5 \;\&\; x$, where | represents bitwise disjunction, & represents bitwise conjunction, $\overline{y}$ represents bitwise negation of y, 4 and 5 represent bit-vector representations of 4 and 5, and all bit vectors are four bits. The uninterpreted function is used simply to represent the variables x and y ($\mathsf{uf}(\mathbf{0}_{\mathrm{bv4}})$ and $\mathsf{uf}(\mathbf{1}_{\mathrm{bv4}})$ respectively).

The goal is to determine whether *SMT2* is satisfiable, which it is. The first step is to translate it into the following ET-ACL2 formula, which is valid when *SMT2* is unsatisfiable:

ACL2-SMT2 :

$$\begin{array}{l}\mathsf{euf}(\mathbf{0}_{\mathrm{ebv4}}) \\ \neq_{\mathrm{euf}} \\ \mathsf{ebvAnd}(4, \mathsf{ebvOr}(4, \mathbf{4}_{\mathrm{ebv4}}, \mathsf{ebvAnd}(4, \mathsf{euf}(\mathbf{0}_{\mathrm{ebv4}}), \mathsf{ebvAnd}(\mathsf{euf}(\mathbf{1}_{\mathrm{ebv4}}), \\ \qquad\qquad\qquad\qquad\qquad\qquad \mathsf{ebvNot}(4, \mathsf{euf}(\mathbf{0}_{\mathrm{ebv4}})))), \\ \qquad \mathsf{ebvAnd}(4, \mathbf{5}_{\mathrm{ebv4}}, \mathsf{euf}(\mathbf{0}_{\mathrm{ebv4}})))\end{array}$$

Descriptions of all the ET-ACL2 primitives used in *ACL2-SMT2* are given in Figure 9.2. The bit-vector representation is defined by function $\mathsf{ebv}(n, x)$, which is defined to create the n-bit, bit vector represented by the Boolean list x; $\mathsf{ebvSize}$, which returns the size of a bit vector x, and $\mathsf{ebvRaw}(x)$, which returns the Boolean list representing the bits of the bit vector x. The function $\mathsf{ebvNbv}(n, x)$ is also defined, which is used to map x to a Boolean list representation when x is expected to be n bits. If x is not n bits, then the empty list is returned, which is equivalent to all low bits.

SMT uninterpreted functions are translated into ACL2 uninterpreted functions by mapping their inputs and outputs into the appropriate embedded type data structure, as shown in Figure 9.3. Thus, the $\mathsf{euf}(x)$ function in *ACL2-SMT2* is defined to return the same 4-bit, bit vector result for any 4-bit, bit vector input.

$$\mathsf{ebv}\,(n, x) \triangleq [n, x]$$

$$\mathsf{ebvSize}\,(x) \triangleq \mathsf{nth}\,(0, x)$$

$$\mathsf{ebvRaw}\,(x) \triangleq \mathsf{nth}\,(1, x)$$

$$\mathsf{nbvEq}\,(n, x, y) \triangleq \begin{cases} \textbf{true}, & \text{if } \mathsf{zp}\,(n) \\ \textbf{false}, & \text{if } \neg(\mathsf{car}\,(x) \leftrightarrow \mathsf{car}\,(y)) \\ \mathsf{nbvEq}\,(n-1, \mathsf{cdr}\,(x), \mathsf{cdr}\,(y)), & \text{otherwise.} \end{cases}$$

$$x =_{\text{ebv}} y \triangleq \begin{cases} \textbf{false}, & \text{if } \mathsf{ebvSize}\,(x) \neq \mathsf{ebvSize}\,(y) \\ \mathsf{nbvEq}\,(\mathsf{ebvSize}\,(x), \mathsf{ebvRaw}\,(x), \mathsf{ebvRaw}\,(y)), & \text{otherwise.} \end{cases}$$

$$\mathsf{ebvNbv}\,(n, x) \triangleq \begin{cases} \mathsf{ebvRaw}\,(x), & \text{if } \mathsf{ebvSize}\,(x) = n \\ [], & \text{otherwise.} \end{cases}$$

$$\mathsf{nbvNot}\,(n, x) \triangleq \begin{cases} [], & \text{if } \mathsf{zp}\,(n) \\ \mathsf{cons}\,(\neg\mathsf{car}\,(x), \mathsf{nbvNot}\,(n-1, \mathsf{cdr}\,(x))), & \text{otherwise.} \end{cases}$$

$$\mathsf{ebvNot}\,(n, x) \triangleq \mathsf{ebv}\,(n, \mathsf{nbvNot}\,(n, \mathsf{ebvNbv}\,(n, x)))$$

$$\mathsf{nbvAnd}\,(n, x, y) \triangleq \begin{cases} [], & \text{if } \mathsf{zp}\,(n) \\ \mathsf{cons}\,(\mathsf{car}\,(x) \wedge \mathsf{car}\,(y), \mathsf{nbvAnd}\,(n-1, \mathsf{cdr}\,(x), \mathsf{cdr}\,(y))), & \text{otherwise.} \end{cases}$$

$$\mathsf{ebvAnd}\,(n, x, y) \triangleq \mathsf{ebv}\,(n, \mathsf{nbvAnd}\,(n, \mathsf{ebvNbv}\,(n, x), \mathsf{ebvNbv}\,(n, y)))$$

$$\mathsf{nbvOr}\,(n, x, y) \triangleq \begin{cases} [], & \text{if } \mathsf{zp}\,(n) \\ \mathsf{cons}\,(\mathsf{car}\,(x) \vee \mathsf{car}\,(y), \mathsf{nbvOr}\,(n-1, \mathsf{cdr}\,(x), \mathsf{cdr}\,(y))), & \text{otherwise.} \end{cases}$$

$$\mathsf{ebvOr}\,(n, x, y) \triangleq \mathsf{ebv}\,(n, \mathsf{nbvOr}\,(n, \mathsf{ebvNbv}\,(n, x), \mathsf{ebvNbv}\,(n, y)))$$

Figure 9.2: The definitions of the bit-vector primitives used in our example.

$$\mathsf{nbvMap}(n,x) \triangleq \begin{cases} [], & \text{if } \mathsf{zp}(n) \\ \mathsf{cons}(\mathbf{false}, \mathsf{nbvMap}(n-1, \mathsf{cdr}(x))), & \text{if } \mathsf{car}(x) = \mathbf{false} \\ \mathsf{cons}(\mathbf{true}, \mathsf{nbvMap}(n-1, \mathsf{cdr}(x))), & \text{otherwise.} \end{cases}$$

$$\mathsf{bvMap}(x) \triangleq \mathsf{nbvMap}(\mathsf{ebvSize}(x), \mathsf{ebvRaw}(x))$$

$$\mathsf{euf}(x_0, x_1, ..., x_k) \triangleq \mathsf{ebv}(n_{\mathsf{euf}}, \mathsf{uf}(\mathsf{bvMap}(x_0), \mathsf{bvMap}(x_1), ..., \mathsf{bvMap}(x_k)))$$

Figure 9.3: A generic SMT uninterpreted function $\mathsf{euf}(...)$, returning a bit vector of size n_{euf}, is defined in the ET-ACL2 model by mapping an ACL2 uninterpreted function $\mathsf{uf}(...)$. into the appropriate domain.

9.4.2 ACL2 Simplification

After translating the problem into an ET-ACL2 formula, the next step is to simplify it using the ACL2 rewriter. First, $=_{\text{ebv}}$ is proven to be an equivalence relation, as shown in *ebvEq-is-equivalence* in Figure 9.4. Next, congruence rules are provided for every function in the ET-ACL2 model, showing that equivalence of its inputs implies the equivalence of its output. Congruence rules for each of the functions in our example are shown in Figure 9.4.

Proving *ebvEq-is-equivalence* (or forcing the theorem prover to accept the rule without proof) enables ACL2 proof strategies specific to equivalence relations, and, most importantly, enables the rewriter to treat the theorem

$$P(x) \to E =_{\text{ebv}} E'$$

as an instruction to rewrite a proof goal F to F' when a subterm G of F and substitution σ can be found satisfying $E/\sigma = G$. F' then is the term formed from F by replacing G with G/σ. For the rewriting to succeed it is also required that P/σ can be proven and, using the congruence rules, it can be shown that

$$E =_{\text{ebv}} E' \to F \leftrightarrow F'.$$

We have therefore created a library of theorems of the form $P \to E =_{\text{ebv}} E'$ that can be used to simplify ET-ACL2 problems. Some of these theorems are shown in Figure 9.5. Note that $\mathsf{syntaxp}(x)$ is a function that logically returns **true**, but is used to implement

ebvEq-is-an-equivalence:
$\mathsf{Booleanp}\,(x =_{\text{ebv}} y) \wedge (x =_{\text{ebv}} x) \wedge ((x =_{\text{ebv}} y) \rightarrow (y =_{\text{ebv}} x)) \wedge$
$((x =_{\text{ebv}} y) \wedge (y =_{\text{ebv}} z) \rightarrow (x =_{\text{ebv}} z))$

ebvAnd-congruence-1:
$(x_0 =_{\text{ebv}} x_1) \rightarrow (\mathsf{ebvAnd}\,(n, x_0, y) =_{\text{ebv}} \mathsf{ebvAnd}\,(n, x_1, y))$
ebvAnd-congruence-2:
$(y_0 =_{\text{ebv}} y_1) \rightarrow (\mathsf{ebvAnd}\,(n, x, y_0) =_{\text{ebv}} \mathsf{ebvAnd}\,(n, x, y_1))$

ebvOr-congruence-1:
$(x_0 =_{\text{ebv}} x_1) \rightarrow (\mathsf{ebvOr}\,(n, x_0, y) =_{\text{ebv}} \mathsf{ebvOr}\,(n, x_1, y))$
ebvOr-congruence-2:
$(y_0 =_{\text{ebv}} y_1) \rightarrow (\mathsf{ebvOr}\,(n, x, y_0) =_{\text{ebv}} \mathsf{ebvOr}\,(n, x, y_1))$

ebvNot-congruence:
$(x_0 =_{\text{ebv}} x_1) \rightarrow (\mathsf{ebvNot}\,(n, x_0) =_{\text{ebv}} \mathsf{ebvOr}\,(n, x_1))$

Figure 9.4: The theorems needed to recognize $=_{\text{ebv}}$ as an equivalence relation and to enable $E =_{\text{ebv}} E'$ to be used as a rewrite rule rewriting E to E' where E is an expression occurring in our example.

ebvAnd-sort1:
$\mathsf{syntaxp}\,(\neg \mathsf{ebvAndOrd}\,(y, x)) \rightarrow (\mathsf{ebvAnd}\,(n, y, x) =_{\text{ebv}} \mathsf{n,ebvAnd}\,(x, y))$

ebvAnd-sort2:
$\mathsf{syntaxp}\,(\neg \mathsf{ebvAndOrd}\,(y, x))$
$\rightarrow$
$(\mathsf{ebvAnd}\,(n, y, \mathsf{ebvAnd}\,(n, x, z)) =_{\text{ebv}} \mathsf{ebvAnd}\,(n, x, \mathsf{ebvAnd}\,(n, y, z)))$

ebvAnd-cancel:
$\mathsf{ebvAnd}\,(n, x, \mathsf{ebvNot}\,(n, x)) =_{\text{ebv}} \mathsf{ebvFill}\,(n, \mathbf{false})$

ebvAnd-zero:
$\mathsf{syntaxp}\,(\mathsf{quotep}\,(y)) \wedge (y =_{\text{ebv}} \mathsf{ebvFill}\,(n, \mathbf{false}))$
$\rightarrow$
$(\mathsf{ebvAnd}\,(n, x, y) =_{\text{ebv}} y)$

Figure 9.5: Some examples of ACL2 theorems that also serve as rewrite rules to simplify the SMT problem.

theorem proving heuristics. In *ebvAnd-sort1* and *ebvAnd-sort2*, syntaxp is used to ensure that the rules only are used when x and y are not in the target order for ebvAnd expressions, as defined by $\mathsf{ebvAndOrd}(x, y)$. In *ebvAnd-zero*, syntaxp is used to ensure that the rule is only applied when y is constant, which prevents the theorem prover from wasting a lot of time attempting to prove the hypothesis when *ebvAnd-zero* is not applicable.

For example, since the target ordering puts $\mathsf{euf}(\mathbf{0}_{\text{ebv4}})$ before $\mathsf{euf}(\mathbf{1}_{\text{ebv4}})$, the rewriting instructions implied by *ebvAnd-sort2* cause *ACL2-SMT2* to be simplified to the following:

$$\begin{array}{l} \mathsf{euf}(\mathbf{0}_{\text{ebv4}}) \\ \neq_{\text{euf}} \\ \mathsf{ebvAnd}(4, \mathsf{ebvOr}(4, \mathbf{4}_{\text{ebv4}}, \mathsf{ebvAnd}(4, \mathsf{euf}(\mathbf{0}_{\text{ebv4}}), \mathsf{ebvAnd}(\mathsf{euf}(\mathbf{1}_{\text{ebv4}}), \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mathsf{ebvNot}(4, \mathsf{euf}(\mathbf{0}_{\text{ebv4}}))))), \\ \qquad\quad \mathsf{ebvAnd}(4, \mathbf{5}_{\text{ebv4}}, \mathsf{euf}(\mathbf{0}_{\text{ebv4}}))) \end{array}$$

then, by theorem *ebvAnd-cancel*, the formula further simplifies to:

$$\begin{array}{l} \mathsf{euf}(\mathbf{0}_{\text{ebv4}}) \\ \neq_{\text{euf}} \\ \mathsf{ebvAnd}(4, \mathsf{ebvOr}(4, \mathbf{4}_{\text{ebv4}}, \mathsf{ebvAnd}(4, \mathsf{euf}(\mathbf{0}_{\text{ebv4}}), \mathsf{ebvFill}(4, \mathbf{false}))), \\ \qquad\quad \mathsf{ebvAnd}(4, \mathbf{5}_{\text{ebv4}}, \mathsf{euf}(\mathbf{0}_{\text{ebv4}}))) \end{array}$$

where $\mathsf{ebvFill}(n, a)$ creates an n-bit, bit vector where each bit has the Boolean value a. Next, using the rewriting strategy implied by *ebvAnd-cancel* and evaluation, the formula further simplifies to the following:

$$\mathsf{euf}(\mathbf{0}_{\text{ebv4}}) \neq_{\text{euf}} \mathsf{ebvAnd}(4, \mathbf{4}_{\text{ebv4}}, \mathsf{ebvAnd}(4, \mathbf{5}_{\text{ebv4}}, \mathsf{euf}(\mathbf{0}_{\text{ebv4}})))$$

Then, using the sorting strategy implied by *ebvAnd-sort1* and *ebvAnd-sort2* and evaluation the formula is further simplified to:

$$\mathsf{euf}(\mathbf{0}_{\text{ebv4}}) \neq_{\text{euf}} \mathsf{ebvAnd}(4, \mathsf{euf}(\mathbf{0}_{\text{ebv4}}), \mathbf{4}_{\text{ebv4}})$$

The above formula cannot be further simplified by our library of rewrite rules and thus is passed to the next phase.

9.4.3 ET-ACL2 to NBV Translation

The next step in solving the SMT problem is to translate the ET-ACL2 formula into a SULFA formula that can be solved with the SAT-based procedure in Chapter 7. The translation is performed by the ACL2 rewriter, and is relatively simple and fast (only a single inside-out pass is required), since the ET-ACL2 functions are defined from Boolean list functions, such as those used in the introductory example from Section 9.2.

Our example translates into the following formula, based on the definitions shown in Figure 9.2:

$$\mathsf{nbvEq}\,(4, \mathsf{uf}\,([\mathbf{false}, \mathbf{false}, \mathbf{false}, \mathbf{false}])$$
$$\mathsf{nBvAnd}\,(4, \mathsf{uf}\,([\mathbf{false}, \mathbf{false}, \mathbf{false}, \mathbf{false}]), [\mathbf{false}, \mathbf{true}, \mathbf{false}, \mathbf{false}]))$$

The above formula is in SULFA, since given constant bit widths, $\mathsf{nbvEq}\,(n, x, y)$ and $\mathsf{nBvAnd}\,(n, x, y)$ are unrollable into list primitives.

9.4.4 Common Subexpression Elimination

Even though the formula is now in SULFA, some more simplification is performed to reduce the size of the problem before solving it with our SAT-based procedure. In the common subexpression elimination phase, new variables are created for expressions that occur multiple times in the formula. This prevents the SAT-based conversion algorithm from translating such expressions multiple times. In the example, a variable is created for the expression $\mathsf{uf}\,([\mathbf{false}, \mathbf{false}, \mathbf{false}, \mathbf{false}])$, as follows:

$$\neg\mathsf{nbvEq}\,(4, x, \mathsf{nBvAnd}\,(4, x, [\mathbf{false}, \mathbf{true}, \mathbf{false}, \mathbf{false}]))$$
$$\text{where } x := \mathsf{uf}\,([\mathbf{false}, \mathbf{false}, \mathbf{false}, \mathbf{false}])$$

Copying the expression for x above would not hinder the performance of our SAT-based procedure much. However, when a complex expression is copied multiple times, common subexpression elimination is significant.

9.4.5 Uninterpreted Function Removal

Next uninterpreted functions are removed. Removing them directly from the bit-vector SMT problem is more efficient than removing them via our SAT-based procedure, since we can create a specialized removal procedure taking into account that uninterpreted functions in bit vector SMT problems only occur at the top level and that each uninterpreted function has a type declaration stating that it inputs and outputs bit vectors of a particular finite width.

The technique to remove uninterpreted functions is the same straightforward method used by our SAT-based procedure. For example, given m calls of an uninterpreted function, $\mathsf{uf}(x_1), \mathsf{uf}(x_2), ...\mathsf{uf}(x_m)$, which inputs an n-bit, bit vector, m new variables are created $u_1, u_2, ...u_m$. Then, the ith call, $\mathsf{uf}(x_i)$, is replaced with the following expression:

$$\mathsf{ite}(\mathsf{nbvEq}(n, x_i, x_1), u_1, \mathsf{ite}(\mathsf{nbvEq}(n, x_i, x_2), u_2, ...\mathsf{ite}(\mathsf{nbvEq}(n, x_i, x_{i-1}), u_{i-1}, u_i)...))$$

Only a single uninterpreted function call occurs in our example after common subexpression elimination. Thus, a single variable u_1 is created, which replaces x, leading to the following expression:

$$\neg\mathsf{nbvEq}(4, u_1, \mathsf{nBvAnd}(4, u_1, [\mathbf{false}, \mathbf{true}, \mathbf{false}, \mathbf{false}]))$$

9.4.6 SAT-Based Procedure

After removing uninterpreted functions, the problem is then passed to our SAT-based procedure for SULFA formulas described in Chapter 7. In the example, the SAT-based procedure finds the counterexample $u_1 \mapsto [\mathbf{false}, \mathbf{true}, \mathbf{false}, \mathbf{false}]$. This is also a satisfying instance the original SMT problem, *SMT2*.

$$\text{addZeros}(n, x) \triangleq \begin{cases} x, & \text{if } \text{zp}(n) \\ \text{addZeros}(n-1, \text{cons}(\mathbf{false}, x)), & \text{otherwise.} \end{cases}$$

$$\text{addZerosC}(a, n, y) \triangleq \begin{cases} y, & \text{if } \neg a \\ \text{addZeros}(n, y), & \text{otherwise.} \end{cases}$$

$$\text{nbvDecode1}(n, x, y) \triangleq \begin{cases} y, & \text{if } \text{zp}(n) \\ \text{nbvDecode1}(n-1, \text{cdr}(x), \text{addZerosC}(\text{car}(x), 2^{n-1}, y)) & \text{otherwise.} \end{cases}$$

$$\text{nbvDecode}(n, x) \triangleq \text{nbvDecode1}(n, x, [t])$$

$$\text{ebvDecode}(n, x) \triangleq \text{ebv}(2^n, \text{nbvDecode}(n, \text{ebvNbv}(n, x)))$$

$$\text{nbvEncode1}(n, m, x) \triangleq \begin{cases} \text{nat2nbv}(n, 0), & \text{if } \text{zp}(m) \\ \text{nat2nbv}(n, m), & \text{if } \text{car}(x) \\ \text{nbvEncode1}(n, m-1, \text{cdr}(x)) & \text{otherwise.} \end{cases}$$

$$\text{nbvEncode}(n, x) \triangleq \text{nbvEncode1}(log_2(n), n-1, x)$$

$$\text{ebvEncode}(n, x) \triangleq \text{ebv}(2^n, \text{nbvEncode}(n, \text{ebvNbv}(n, x)))$$

Figure 9.6: The definitions of bit-vector primitives $\text{ebvDecode}(n, x)$ and $\text{ebvEncode}(n, x)$, which are used to encode and decode bit vectors into and from exponentially larger one-hot signals.

9.5 Adding New Functions and Rewriting Strategies

The main advantage to our approach is its flexibility. Whereas most SMT solvers have a fixed set of primitives and a single simplification strategy, our system is extendable with new primitives and strategies. Any user created functions and rewrite rules will be treated no differently from the primitive functions in the original system. For example, the SMT 2006 bit-vector library includes no primitives to encode and decode wires into one-hot signals—an operation common in the TRIPS Load Store Queue implementation. In Figure 9.6, the $\text{ebvEncode}(n, x)$ and $\text{ebvDecode}(n, x)$ functions are defined to implement these opera-

tions. Now, encoding and decoding can be added to our suite of bit-vector primitives, with the same mixture of ACL2 and SAT-based verification as the previous bit-vector primitives. Furthermore, simplification rules, such as the following *Encode-decode-elimination* rule, can be proven using the ACL2 theorem prover and added to the ACL2 simplification phase.

Encode-decode-elimination:

$$\mathsf{syntaxp}\,(\mathsf{quotep}\,(n) \wedge \mathsf{quotep}\,(n_2)) \wedge (2^n = n_2)$$
$$\rightarrow$$
$$(\mathsf{ebvEncode}\,(n_2, \mathsf{ebvDecode}\,(n, x)) =_{\mathrm{ebv}} x)$$

The above rule states that when n and n_2 are constants, and $n_2 = 2^n$, then decoding and then encoding a bit vector x from size n-bits to n_2 bits results in something equivalent to x.

Note that libraries of rules over the SMT 2006 primitives can also be created for domain-specific applications. For example, rules with hypotheses, especially with free variables, like those discussed in Chapter 4, are inefficient in many applications but critical to others. Rules for each application can be proven correct, so that soundness of the system need not rely on soundness of user-generated rules.

9.6 Results

All 8,246 problems in the SMT 2006 QF_UFBV32 benchmark suite were solved by the SULFA SMT solver. The average time to solution on our machine is about 3 seconds, with a maximum time of 2 minutes [1]. These times are not competitive with the fastest SMT solvers on these benchmarks, but we are the only SMT solver we know of to make use of a general-purpose rewriter and definition mechanism.

Figure 9.7 shows the result of removing the ACL2 simplification phase from the SULFA SMT solver. While using the general-purpose theorem prover to perform simplification generates overhead, in the great majority of problems, the overhead is smaller than the benefit from doing simplification. Also, four problems were omitted from Figure 9.7

[1] An Intel Pentium® IV Dual Core 3.0 GHz with 2GB of RAM

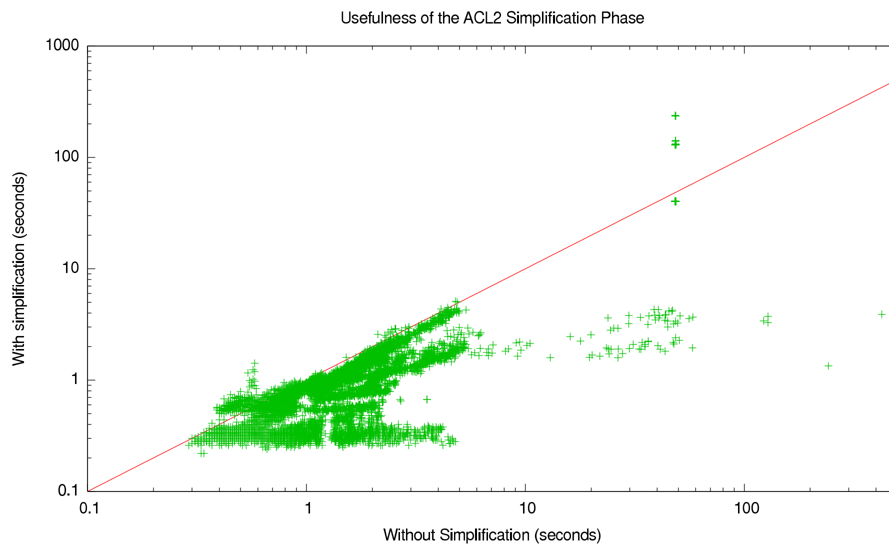


Figure 9.7: A graph comparing the performance of the SULFA SMT solver with and without the ACL2 simplification phase. Each point represents a single problem in the SMT 2006 QF_UFBV32 bit vector suite. The x-axis is the time (in seconds) required without the ACL2 simplification phase and the y-axis is the time (in seconds) required with the ACL2 simplification phase. A point is on the line if the same time is required by both techniques.

because they required over ten minutes to solve without the ACL2 simplification phase (but could be solved in under ten minutes with the ACL2 simplification phase).

9.7 Summary

The SAT-based procedure for verifying SULFA formulas described in Chapter 7 has been used, along with the ACL2 theorem prover, to develop a solver for the standard SMT 2006 QF_UFBV32 theory of bit vectors. Our solver for the theory of bit vectors was able to solve all the problems in the 2006 benchmark suite. Furthermore, it is more flexible than any other known SMT solvers, since it provides a powerful mechanism for extending it with new primitives and rewrite rules as ACL2 definitions and theorems.

9.8 Development and Bibliographic Notes

The initial high-level design of the SULFA SMT solver was developed during discussions with Panagiotis Manolios and Sudarshan Srinivasan.

A number of SMT solvers exist that can automatically solve the SMT 2006 benchmark problems. The most efficient of these on the 2006 benchmarks are the yices SMT solver [92] and the STP solver. The design of the STP solver is described in detail in the proceedings of CAV 2007 [17]. STP includes a simplification phase, similar to the ACL2 simplification phase in the SULFA SMT solver. Instead of a general-purpose theorem prover though, STP uses a special purpose simplifier. STP also includes a linear arithmetic mechanism and an automated refined mechanism for arrays that are not implemented in the SULFA SMT solver. It may be possible to add a similar linear arithmetic mechanism to the SULFA SMT solver using the ACL2 theorem prover. However, implementing the automated refinement mechanism requires modifications to the SAT-based SULFA procedure to enable incremental additions of hypotheses.

Chapter 10

Integrating ACL2 with the SixthSense Model Checker

10.1 Introduction

We have developed a hardware verification methodology that uses an industrial model checker to automate many hardware verification proofs and avoid the semantic embedding of a hardware description language in the ACL2 logic. In this approach, which we call the ACL2SIX methodology, the ACL2 theorem prover is combined with the industrial model checker SixthSense through a new proof mechanism, named the ACL2SIX hint. The ACL2SIX methodology models the hardware design as a set of axioms in the ACL2 logic that are never explicitly given to the ACL2 theorem prover. Instead, the ACL2SIX hint proves properties from axioms outside the ACL2 theorem prover by using the SixthSense model checker on the actual hardware implementation. Theorems proven by the ACL2SIX hint can then be combined using the ACL2 theorem prover to prove properties beyond the scope of what can be verified by SixthSense alone.

This chapter gives an overview of the ACL2SIX hardware verification methodology and then shows how it has been applied to the verification of a high performance multiplier,

used in a floating-point unit designed at IBM. This methodology is described in more detail in the proceedings of the Sixth conference on Formal Methods in Computer Aided Design [77]. The verification of the multiplier is also described in more detail at the Sixth International Workshop on the ACL2 theorem prover and its Applications [71].

10.2 ACL2VHDL Property

First, we define the notion of an ACL2VHDL property, which is a set of first-order formulas that can be translated into a VHDL assertions. An *ACL2VHDL property* is of the form:

$$(n \in \mathbb{N}) \land (n \leq n_0) \rightarrow (E_0 = E_1).$$

where n is a natural number variable representing the current clock cycle; n_0 is a natural number, representing the number of cycles needed to initialize the hardware; and E_0 and E_1 are ACL2VHDL bit-vector expressions. An *ACL2VHDL bit-vector expression* is one of the following function applications:

- **Bit-vector constant generators.** Each bit-vector constant generator is a defined function that maps ACL2 constants to bit-vector constants. A *bit vector* is a pair (n, x), where n is the size of the bit vector and x is a natural number representing its value. A *bit* is either 0 or 1. For example, given a natural number n, $\mathsf{pad0}\,(n) = (n, 0)$ and $\mathsf{pad1}\,(n) = (n, 2^n - 1)$, create bit vectors of size n containing all zeroes or all ones respectively. Furthermore, given an $n \in \{0, 1\}$ the function $\mathsf{make\text{-}bit}\,(n) = n$ returns the corresponding bit.

  Bit-vector primitives. A bit-vector primitive is a member of a finite set of functions that map bit or bit-vector arguments to an output that is either a bit or a bit vector. bit-vector primitives are defined in both the ACL2 logic and in VHDL, and therefore can be easily translated from one to the other. Many of the typical bit-vector functions are implemented, such as follows:

- bvPlus (x, y) is the bit vector representing the addition of the values represented by x and y (all values are unsigned and the resulting bit vector is truncated to the size of x).
- bvTimes (x, y) is the bit vector representing the multiplication of bit vectors x and y (all values are unsigned and the resulting bit vector is truncated to the size of x).
- bvNot (x) is the bitwise negation of x.
- bvIf (a, x, y) is either the bit vector x or y, depending on whether bit a is 1 or 0 respectively.

Note that, unlike in Chapters 8 and 9, the above bit-vector primitives do not input the bit widths explicitly, since the width of bit vectors is encoded as part of its value (e.g., the 4-bit, bit vector 1 is encoded as the pair (4, 1)).

In an ACL2VHDL bit-vector expression, the argument to a bit-vector primitive can be any ACL2VHDL bit-vector expression. Therefore bvPlus (bvNot $(x), x)$ is an ACL2VHDL bit-vector expression.

- **Sigbit and Sigvec** represent signals in the hardware design.

  Before defining a well-formed application of sigbit and sigvec, first define a *cycle expression* as either n or $n - n_0$, where n is a variable symbol, representing the current clock cycle and n_0 is a natural number constant.

  Given a constant e, a constant w, and a cycle expression c, then sigbit $(e, w, c, 0)$ represents the bit associated with the signal named w in the design entity e during the first half of clock cycle c. Similarly, sigbit $(e, w, c, 1)$ represents the bit associated with the signal names w in the design entity e during the second half of clock cycle c.

  Given a constant e, a constant w, natural number b_l, a natural number b_h, a cycle expression c, and a phase number $p \in \{0, 1\}$, then sigvec (e, w, b_l, b_h, c, p) represents

the bit vector formed from bits b_l through b_h of wire w in design entity e at clock cycle c and clock phase p. For example, given a design entity **ent** (an entity is actually a constant containing file names, module names, and other design info) and a wire name **a**, then $\mathsf{sigvec}(\mathbf{ent}, \mathbf{a}, 2, 5, n-1, 0)$ represents the four bit, bit vector made up of bits 2 through 5 of wire a during the first half of clock cycle $n-1$.

ACL2VHDL properties are further restricted to contain only a single variable symbol, denoting the current clock cycle, and only a single hardware design entity. For example, the following is an ACL2VHDL property:

$$(n \in \mathbb{N}) \wedge (n \leq 4) \rightarrow \mathsf{sigvec}(\mathbf{E}, \mathbf{a}, 0, 4, n-1, 1) = \mathsf{bvNot}(\mathsf{sigvec}(\mathbf{E}, \mathbf{b}, 1, 5, n, 0))$$

the above property states that for all cycles after the first 4, in the hardware design represented by entity **E**, bits 1 through 5 of signal **b** at the beginning of a clock cycle is equal to the negation of bits 0 through 4 of signal **a** in the middle of the previous cycle. Note that the above property would not be an ACL2VHDL property if the second application of sigvec were modified to contain a design entity or a clock cycle variable other than those used in the first application of sigvec.

10.3 Overview of ACL2SIX

We have modified the ACL2 theorem prover to include a new proof procedure, named the ACL2SIX hint. The ACL2SIX hint is called directly by the user to prove ACL2VHDL properties. The ACL2SIX hint uses the SixthSense model checker to determine if the given ACL2VHDL property holds on the hardware design entity it references. If so, then it is valid; otherwise, a waveform can be viewed showing why the implementation does not satisfy the given ACL2VHDL property.

Figure 10.1 presents an overview of the ACL2SIX hardware verification methodology. A hardware design, written in VHDL, is specified in the ACL2 logic as a first-order

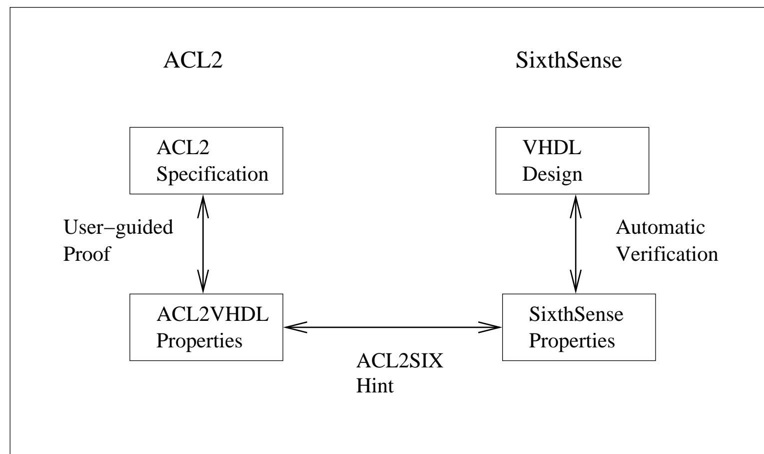


Figure 10.1: An overview of the ACL2SIX hardware verification methodology.

property $P(w_0, w_1, ..., w_n)$, where each w_i is an application of sigbit or sigvec corresponding to a wire in the design. The property P is decomposed, via user guided proof, into ACL2VHDL properties. The ACL2SIX hint translates each ACL2VHDL property into VHDL assertions, checkable by the SixthSense model checker. If one of the resulting assertion is not valid, then a waveform showing how the hardware design does not satisfy the ACL2VHDL property is presented; otherwise, the validity of P follows from the validity of its decomposed ACL2VHDL properties.

Note that the SixthSense model checker is mostly automatic. Therefore, by using SixthSense, a considerable amount of reasoning on the details of the implementation is avoided. Optional arguments can also be passed through the ACL2SIX hint, to help guide or configure the SixthSense run for any particular problem.

As an example, consider the circuit in Figure 10.2. A valid property of the circuit is:

$$\Box(\mathbf{B} \rightarrow \Box\mathbf{B})$$

which states that once **B** is true, then **B** continues to be true for all future cycles. The

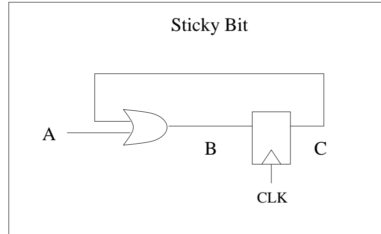


Figure 10.2: A simple example circuit, in which a true bit **A** causes bit **B** and bit **C** to be true for all later times.

property can be specified in first order logic as:

$$(n_0 \in \mathbb{N}) \wedge (n_1 \in \mathbb{N}) \wedge (n_0 \leq n_1) \wedge \mathsf{sigbit}\,(\mathbf{stickyBit}, \mathbf{B}, n_0, 0) \rightarrow \mathsf{sigbit}\,(\mathbf{stickyBit}, \mathbf{B}, n_1, 0)$$

where **stickyBit** is the constant representing the circuit design entity.

Note that the above formula is not an ACL2VHDL formula, since it contains two different cycle number variables. However, by induction on $n_1 - n_0$, it can be reduced to the following ACL2VHDL property:

$$(n \in \mathbb{N}) \wedge (1 \leq n)$$
$$\rightarrow$$
$$\mathsf{bvIf}\,(\mathsf{sigbit}\,(\mathbf{stickyBit}, \mathbf{B}, n-1, 0), \mathsf{sigbit}\,(\mathbf{stickyBit}, \mathbf{B}, n, 0)), \mathsf{make\text{-}bit}\,(1))$$
$$=$$
$$\mathsf{make\text{-}bit}\,(1)$$

the above property is an ACL2VHDL property, and, therefore, can be translated by the ACL2SIX hint into VHDL assertions. SixthSense can then check that the hardware design corresponding to **stickyBit** satisfies the resulting assertions.

10.3.1 Soundness

Our FMCAD paper [77] contains a proof sketch justifying the soundness of the ACL2SIX methodology. The proof sketch shows that sigbit and sigvec could have been introduced as

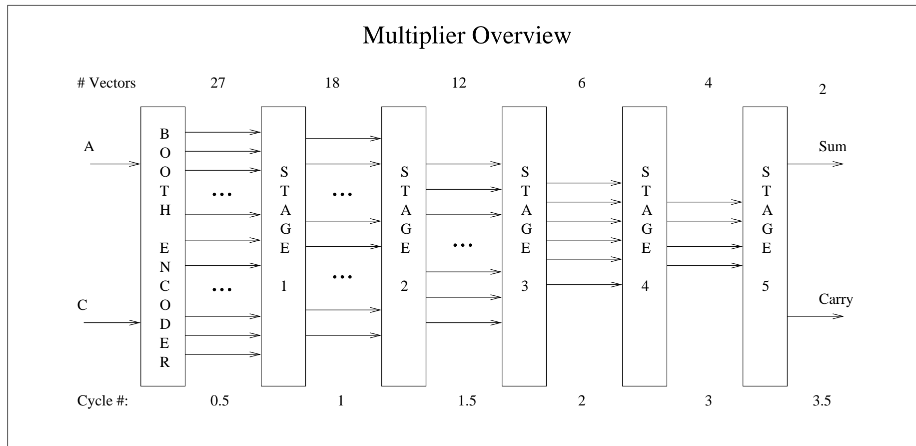


Figure 10.3: An overview of a high performance multiplier design that was verified using the ACL2SIX methodology.

ACL2 constrained functions where all of the unwritten axioms are constraints. Therefore, the unwritten axioms do not result in an unsound theory.

A few assumptions must be made to ensure the soundness of the ACL2SIX hint, including the soundness of SixthSense itself and that the bit-vector primitives have equivalent definitions in VHDL and ACL2. Another assumption is that `sigbit` and `sigvec` are never functionally instantiated, which is necessary because functional instantiation assumes that all of the axioms regarding a function have been given to the ACL2 theorem prover. Originally, ACL2 had no support for functions with axioms outside the theorem prover itself. Thus, the soundness depended on a manual check that `sigbit` and `sigvec` were not instantiated. The new implicitly theory mechanism described in Chapter 11, however, supports such functions.

10.4 Overview of Multiplier Verification

Using the ACL2SIX methodology, we verified a 54x53 bit multiplier design used in an industrial high performance floating-point unit. An overview of the multiplier design is shown in Figure 10.3. First, a Booth encoder [63] is used to reduce the multiplication of two bit vectors into the summation of 27 bit vectors. The summation of 27 bit vectors is then reduced in successive stages to 18, 12, 6, 3, and then 2 bit vectors. The summation of the final two bit vectors occurs in a different design.

The correctness of the multiplier design is expressed as the following first-order property:

$$(7 \leq n) \rightarrow \mathsf{bvPlus}\,(\mathsf{sum}\,(n, 1), \mathsf{carry}\,(n, 1)) = \mathsf{bvTimes}\,(\mathsf{a\text{-}in}\,(n - 4, 2), \mathsf{c\text{-}in}\,(n - 4, 2))$$

where $\mathsf{sum}\,(n, p)$, $\mathsf{carry}\,(n, p)$, $\mathsf{a\text{-}in}\,(n, p)$, and $\mathsf{c\text{-}in}\,(n, p)$ are the values of the **sum**, **carry**, **A**, and **C** wires in Figure 10.3 at cycle number n and phase p. Note that the wires in Figure 10.3 are a slight abstraction of the inputs and outputs of the actual design. For example, the actual design contains input signals that when true, imply **A** is zero. Therefore, the function $\mathsf{a\text{-}in}\,(n, p)$ represents an ACL2VHDL bit-vector expression involving multiple inputs in the actual design.

The correctness theorem of the multiplier in Figure 10.3 is an ACL2VHDL property. However, due to the complexity of the design and the well-known problems that occur during the automatic verification of multipliers, the correctness of the multiplier cannot be verified automatically by SixthSense. Instead, the ACL2 theorem prover is used to verify a Booth encoder. Then, SixthSense is used to verify that each vector output by the ACL2 Booth encoder is equivalent to a vector output by the Booth encoder implementation. SixthSense is also used to verify each individual summation compressor unit, which reduces the sum of four or three bit vectors into the sum of two bit vectors. Finally, the ACL2 theorem prover is used to show that the multiplier's correctness theorem follows from the above theorems.

Our ACL2 workshop paper [71] presents a more detailed description of the multiplier verification.

10.5 Summary

We have integrated an industrial-strength model checker, SixthSense, with an industrial-strength theorem prover, ACL2. The hardware verification problem is divided into an ACL2 problem, which reasons primarily about high-level arithmetic properties, and a SixthSense problem, which reasons primarily about low-level details of the VHDL implementation. By combining the two tools, larger problems can be verified than is possible with SixthSense alone and a greater degree of automation is provided than is available through the ACL2 theorem prover alone. Furthermore, by accessing the hardware model through the model checker, we are able to avoid writing a formal semantics for VHDL in ACL2.

10.6 Development and Bibliographic Notes

The work described in this chapter is joint work with Jun Sawada. The translator from ACL2VHDL properties to VHDL is based on the translator described by Sawada in the ACL2 workshop [76]. Also, Sandip Ray wrote an early prototype of the ACL2SIX hint. I completed the implementation of the ACL2SIX hint and used it to verify the multiplier design.

A significant amount of previous work involves the integration of model checking and theorem proving. Some examples are that the HOL theorem prover was integrated with the Voss model checking system [36], a lightweight theorem prover was integrated with the FORTE automated verification tool [78], a μ-calculus model checker was integrated with PVS [61], a unifying framework for connecting model checking and theorem proving has been presented [5], the μ-calculus has been formalized in ACL2 [43], ACL2 has been integrated with the UCLID automated verification tool [44], and some verification of SMV's

compositional model checking has been done in ACL2 [69].

Our work is distinguished from most previous work in its use of a model checker and theorem prover each of which on its own has been shown to be applicable to industrial hardware verification problems [52, 72]. Furthermore, our integration technique is unique in that it avoids a full formal embedding of the logic from one tool in the other. A huge amount of effort was saved by avoiding the formalization of the semantics of VHDL within ACL2.

Furthermore, the industrial verification techniques involving FORTE and HOL-Voss rely primarily on symbolic trajectory evaluation, which can only express finite step properties (similar to those expressible in SULFA). ACL2VHDL properties, on the other hand, can express more general invariants. For example, let **P** be an invariant in some finite state machine. Thus, the following temporal logic property holds

$$\Box(\mathbf{P}),$$

where the valid inputs and initial machine assumptions are implicit. The above property can be expressed directly in ACL2VHDL, but not directly in STE. If given to ACL2SIX, the above property is translated into a VHDL assertion, which the SixthSense model checker will then attempt to verify through reachability analysis. Verifying the above property using STE, on the other hand, may require a non-trivial strengthening of **P** until some inductive invariant holds, which can be verified through finite-step properties, as in Chapter 8.

Chapter 11

A General-Purpose Mechanism for Integrating External Tools with ACL2

11.1 Introduction

The initial implementations of the SULFA and SixthSense extensions to the ACL2 theorem prover, described in Chapter 7 and Chapter 10, required modifications to the ACL2 source code. This chapter, however, presents a general-purpose mechanism for extending the theorem prover with new proof engines without modifying the ACL2 source code. The general-purpose extension, which was described in more detail at the International Workshop on Implementation of Logics [42] and in our paper in the Journal of Applied Logic [40], is now included in the distributed version of the ACL2 theorem prover.

Aside from avoiding future source code modification, the general-purpose mechanism clarifies the criteria needed for an extension of ACL2 to be sound. The correctness criteria effectively form a contract between the extension writer and user, of which the ACL2 system itself need not be a part. The ACL2 system merely ensures that only extensions

on which the user has agreed to trust are used. Therefore, potentially unsound extensions can be distributed, even with the theorem prover, since a careful user will only trust sound extensions.

Our mechanism relies on a new proof engine called a *clause processor*. Section 11.2 describes verified clause processors, which use the theorem prover to ensure soundness of the combined system. Section 11.3 discusses unverified clause processors, which rely on the writer of the clause processor to ensure soundness of the combined system. These unverified clause processors use a new mechanism, called a *trust tag*, to ensure that the only proof engines used are those the user has acknowledged as trusted.

11.2 Verified Clause Processors

An *ACL2 clause* is a list of ACL2 terms, representing a disjunction of those terms. A clause processor is a function that inputs an ACL2 clause and outputs a (hopefully simpler) list of clauses from which its input clause can be derived. For example, given the following ACL2 clause:

$$[\ulcorner \texttt{(IF A B C)} \urcorner, \ulcorner \texttt{D} \urcorner]$$

which represents the ACL2 formula

$$\texttt{(IF A B C)} \neq \texttt{'NIL} \vee \texttt{D} \neq \texttt{'NIL}$$

then a clause processor that implements case splitting might produce the following list of clauses:

$$[[\ulcorner \texttt{(EQUAL A 'NIL)} \urcorner, \ulcorner \texttt{B} \urcorner, \ulcorner \texttt{D} \urcorner], [\ulcorner \texttt{A} \urcorner, \ulcorner \texttt{C} \urcorner, \ulcorner \texttt{D} \urcorner]]$$

which represents

$$(\texttt{A} = \texttt{'NIL} \vee \texttt{B} \neq \texttt{'NIL} \vee \texttt{D} \neq \texttt{'NIL}) \wedge (\texttt{A} \neq \texttt{'NIL} \vee \texttt{C} \neq \texttt{'NIL} \vee \texttt{D} \neq \texttt{'NIL}).$$

Since ACL2 clauses can be expressed as ACL2 constants, a clause processor function can be written in ACL2. Furthermore, sometimes *verified clause processors* can be

created, which reduce the soundness of the clause processor to the soundness of the underlying ACL2 system.

The interface to a verified clause processor has two components: (i) the clause processor rule class, which identifies an ACL2 function as a clause processor and (ii) the clause processor hint mechanism, which directs the theorem prover to use a clause processor on a particular problem. Before describing the clause processor rule class and hint in more detail, the following terminology is introduced:

- first (X), given a list X, is the first element of the list.
- rest (X), given a list X, is the list formed from X after deleting the first element.
- insert (e, X) is the list formed from the list X by inserting the element e at its front.
- $|X|$, given a list X, is the length of the list.
- list $(x_0, x_1, ..., x_N)$ is the list composed of elements x_0 through x_N, which may also be written as $[x_0, x_1, ..., x_N]$.
- [] is the empty list.
- We use the same definition of MakeFn and MakeIf as in Chapter 5. Given a function symbol f and ACL2 terms X_1 through X_n, define MakeFn $(n, X_1, X_2, ..., X_n)$ as the term representing the application of f to arguments X_1 through X_n. MakeIf $(x, y, z) \triangleq$ MakeFn $(\ulcorner\texttt{IF}\urcorner, x, y, z)$.
- Given an ACL2 function application E and a natural number i, or Arg (i, E), is defined, as in Chapter 5, as the ith argument of E. For example, Arg $(1, \ulcorner\texttt{(IF A B C)}\urcorner) = \ulcorner\texttt{A}\urcorner$.
- NA (E), given a function application $E \in \mathbb{E}$, is its number of arguments. For example, NA $(\ulcorner\texttt{(IF A B C)}\urcorner) = 3$.

- Given an ACL2 term E, the *size of* E, or $|E|$, is also defined as in Chapter 5:

$$|E| \triangleq \begin{cases} 0, & \text{if } E \text{ is a constant} \\ 1, & \text{if } E \text{ is a symbol} \\ 2 + \sum_{i=1}^{\mathsf{NA}(E)} |\mathsf{Arg}(i, E)|, & \text{otherwise.} \end{cases}$$

- $\mathsf{Args}(E)$, given an ACL2 term E representing a function application returns the list of ACL2 terms representing the arguments of that function application. For example,

$$\mathsf{Args}(\ulcorner\texttt{(F A B C)}\urcorner) = [\ulcorner\texttt{A}\urcorner, \ulcorner\texttt{B}\urcorner, \ulcorner\texttt{C}\urcorner.]$$

- Given a list of ACL2 terms C, define

$$\mathsf{disjoin}(C) \triangleq \begin{cases} \ulcorner\texttt{NIL}\urcorner, & \text{if } C = [] \\ \mathsf{MakeIf}(\mathsf{first}(C),\ \ulcorner\texttt{T}\urcorner,\ \mathsf{disjoin}(\mathsf{rest}(C))), & \text{otherwise.} \end{cases}$$

Intuitively, $\mathsf{disjoin}(C)$ is the ACL2 term representing the disjunction of the elements of C.

- Given a list of lists of ACL2 terms F, define

$$\mathsf{conjoin^*}(F) \triangleq \begin{cases} \ulcorner\texttt{T}\urcorner, & \text{if } F = [] \\ \mathsf{MakeIf}(\mathsf{disjoin}(\mathsf{first}(F)),\ \mathsf{conjoin^*}(\mathsf{rest}(F)),\ \ulcorner\texttt{NIL}\urcorner), & \text{otherwise.} \end{cases}$$

Intuitively, $\mathsf{conjoin^*}(F)$ is the ACL2 term representing the conjunction of each $\mathsf{disjoin}(C)$, where C is an element of F. For example, $\mathsf{conjoin^*}([[\ulcorner\texttt{A}\urcorner, \ulcorner\texttt{B}\urcorner], [\ulcorner\texttt{C}\urcorner]])$ is:

```
(IF (IF A 'T (IF B 'T 'NIL))
   (IF (IF 'C 'T 'NIL) 'T 'NIL)
  'NIL)
```

$$\begin{array}{l} \mathsf{ev}\,(\mathsf{conjoin^*}\,(\mathsf{tool0}\,(C, args)), \mathsf{tool0\text{-}env}\,(C, \sigma, args)) \\ \rightarrow \\ \mathsf{ev}\,(\mathsf{disjoin}\,(C), \sigma) \end{array}$$

Figure 11.1: A formula stating the correctness of the clause processor tool0 .

which represents $(a \vee b) \wedge c$.

Using the above terminology, Figure 11.1 illustrates an ACL2 theorem stating the correctness of an arbitrary clause processor tool0 $(C, args)$. The clause processor is correct if for any ACL2 clause C and substitution σ such that the clause evaluates to true under σ, there exists a substitution σ' (provided by tool0-env $(C, \sigma, args)$) such that the conjunction of clauses returned by the clause processor evaluates to true under σ'.

A verified clause processor with the name tool0 is introduced by tagging the theorem in Figure 11.1 with the new clause processor rule-class. Only theorems that match the syntax of the formula in Figure 11.1, for some clause processor tool0 , evaluator ev , and function tool0-env , may be tagged with the new clause processor rule class.

Once a clause processor rule with the clause processor tool0 has been added to ACL2's rule database, then clause processor hints referring to tool0 may be used. The hint tells the ACL2 theorem prover when encountering a given goal to replace it with tool0 $(C, args)$, where C is the proof object at that point in the proof and *args* is any user guidance passed to the clause processor hint.

11.2.1 Example

As an example, we have developed a verified clause processor that sorts the arguments of 32 bit, bit-vector addition. The ACL2 function `BV-ADD` implements binary 32 bit, bit-vector addition. Since modular addition is commutative and associative, all permutations of the summands in 32 bit, bit-vector addition are equivalent. Thus,

```
(BV-ADD A (BV-ADD B (BV-ADD C D)))
```

is equivalent to

```
(BV-ADD A (BV-ADD C (BV-ADD B D))).
```

Sorting the summands in a bit-vector addition helps to create a normal form that is important in many verification problems, including the verification of the multiplier in Chapter 10.

A naive approach to sorting is to use the ACL2 general-purpose rewriter to rewrite all instances of `(BV-ADD X Y)` to `(BV-ADD Y X)` and `(BV-ADD X (BV-ADD Y Z))` to `(BV-ADD Y (BV-ADD X Z))` when the term being substituted for `X` is greater than the term being substituted for `Y` by some syntactic measure. However, this approach requires $O(N^2)$ applications of the rewrite rules to sort a nested `BV-ADD` term with N arguments.

Figure 11.2 defines a verified clause processor that uses mergesort to sort the summands of a nested `BV-ADD` term. The clause processor, named sortBVAddCP (L) is defined as a mutually-recursive function described as follows:

- Given a list X, the function makeBVAdd (X) creates the nested `BV-ADD` term such that the ith element of X is the ith summand in makeBVAdd (X). For example,

  makeBVAdd (list (⌜`A`⌝, ⌜`B`⌝, ⌜`(F X)`⌝)) = ⌜`(BV-ADD A (BV-ADD B (F X)))`⌝.

- Given an ACL2 term E, the function sortBVAdd (E) creates an equivalent term with `BV-ADD` summands sorted. For example,

  sortBVAdd (⌜`(BV-ADD B (BV-ADD C A))`⌝) = ⌜`(BV-ADD A (BV-ADD B C))`⌝,

  This is accomplished by collecting the summands into a list, using mergesort to sort the list, and then using makeBVAdd to make a `BV-ADD` term from the sorted list. In our example, `(BV-ADD B (BV-ADD C A))` is collected into the list [⌜`B`⌝, ⌜`C`⌝, ⌜`A`⌝], which is then sorted to created [⌜`A`⌝, ⌜`B`⌝, ⌜`C`⌝], before using makeBVAdd to create the final `BV-ADD` expression.

$$\mathsf{makeBVAdd}(L) \triangleq \begin{cases} \mathsf{first}(L), & |L| < 2 \\ \mathsf{MakeFn}(\ulcorner\texttt{BV-ADD}\urcorner, \mathsf{first}(L),\ \mathsf{makeBVAdd}(\mathsf{rest}(L))), & \text{otherwise.} \end{cases}$$

$$\mathsf{sortBVAdd}(E) \triangleq \begin{cases} E, & \text{if } |E| < 2 \\ \mathsf{MakeFn}(\mathsf{Fn}(E),\ \mathsf{sortBVAddL}(\mathsf{Args}(E))), & \text{if } \mathsf{Fn}(E) \neq \ulcorner\texttt{BV-ADD}\urcorner \\ \mathsf{makeBVAdd}(\mathsf{mergesort}(\mathsf{collect}(E))), & \text{otherwise.} \end{cases}$$

$$\mathsf{sortBVAddL}(L) \triangleq \begin{cases} [], & \text{if } L = [] \\ \mathsf{insert}(\mathsf{sortBVAdd}(\mathsf{first}(L)),\ \mathsf{sortBVAddL}(\mathsf{rest}(L))), & \text{otherwise.} \end{cases}$$

$$\mathsf{collect}(E) \triangleq \begin{cases} \mathsf{list}(E), & \text{if } |E| < 2 \\ \mathsf{list}(\mathsf{sortBVAdd}(E)), & \text{if } \mathsf{Fn}(E) \neq \ulcorner\texttt{BV-ADD}\urcorner \\ \mathsf{insert}(\mathsf{sortBVAdd}(\mathsf{Arg}(1,\ E)), \mathsf{collect}(\mathsf{Arg}(2,\ E))), & \text{otherwise.} \end{cases}$$

$$\mathsf{sortBVAddCP}(L) \triangleq \mathsf{list}(\mathsf{sortBVAddL}(L)).$$

Figure 11.2: The mutually-recursive definition of a clause processor, $\mathsf{sortBVAddCP}(L)$, which sorts the arguments of all instances of `BV-ADD` that occur in L. In the above definition, E is always an ACL2 term, L is always a list of ACL2 terms, and $\mathsf{mergesort}(L)$ sorts a list of terms using the mergesort algorithm.

- Similarly, the function sortBVAddL (L), given a list of ACL2 terms L, creates a list of terms equivalent to L with `BV-ADD` summands sorted.

- The clause processor sortBVAddCP (L), given a list of ACL2 terms L representing an ACL2 clause, creates an equivalent (singleton) list of clauses with the summands within `BV-ADD` subterms sorted.

The correctness of the sortBVAddCP (L) can be stated as:

$$\mathsf{ev}\,(\mathsf{conjoin^*}\,(\mathsf{sortBVAddCP}\,(C)), \sigma) \rightarrow \mathsf{ev}\,(\mathsf{disjoin}\,(C), \sigma)$$

which is proven using the ACL2 theorem prover. The proof involves the following key lemmas:

1. $\mathsf{perm}\,(x, y) \rightarrow \mathsf{ev}\,(\mathsf{makeBVAdd}\,(x), \sigma) = \mathsf{ev}\,(\mathsf{makeBVAdd}\,(y), \sigma)$

   which states that any permutation of a nested `BV-ADD` applications evaluates to the same result.

2. $\mathsf{perm}\,(x, \mathsf{mergesort}\,(x))$

   which states that the mergesort algorithm always returns a permutation of its input.

The sorting clause processor is added to the ACL2 system the sortBVAddCP correctness theorem with the clause processor rule class. Once it has been added, a hint can be used to apply sortBVAddCP to any ACL2 clause.

The verified clause processor shows significantly better performance than the naive approach. A nested `BV-ADD` with 500 summands is sorted by the verified clause processor in 0.01 seconds and a nested `BV-ADD` with 1000 summands is sorted in 0.02 seconds. By contrast, the naive approach that relies on the general-purpose rewriter requires 11.24 seconds and 64.41 seconds respectively [1].

[1] These results were obtained on a 2.6GHz Pentium® IV desktop computer with 2.0GB of RAM.

11.3 Basic Unverified Clause Processors

For some clause processors, it is difficult or impossible to verify the property in Figure 11.1 using the ACL2 theorem prover. For example, a clause processor that implements the SULFA decision procedure described in Chapter 5 cannot be proven correct using the ACL2 theorem prover. One reason is that such a proof requires a second order axiom about all ACL2 functions: the expansion of any function application is equal to its body. Also, admitting the decision procedure as a function in the ACL2 logic would be impossible, since its proof of termination requires ordinals greater than ϵ_0.

The unverified clause processor mechanism allows clause processors that are not proven correct to be added to the ACL2 theorem prover. Like a verified clause processor, an unverified clause processor is an ACL2 function that maps an ACL2 clause to a list of clauses. Unlike verified clause processors, unverified clause processors may use *program-mode* ACL2 functions, which are not required to terminate on all inputs. Furthermore, unverified clause processors can use raw Lisp procedures (i.e., functions defined in the Common Lisp environment in which ACL2 executes), which may make operating system calls executing external programs or reading and writing files.

Unverified clause processors are implemented using a new feature, called trust tags. A *trust tag* is a tag associated with a block of code that the user must declare trustworthy before being executed. Before the declaration of a trust tag, the authors of ACL2 were essentially responsible for its soundness. After a trust tag is declared, however, the soundness depends on the soundness of ACL2 and the soundness of the block of code associated with the trust tag. The block of code associated with a trust tag can be any arbitrary Lisp code, including code that alters ACL2's internal state.

Trust tags allow non-authors of the theorem prover to develop new features without having to distribute their own versions of the ACL2 theorem prover. Instead, new features can be distributed in an ACL2 file that is linked in dynamically with the ACL2 system, in the same manner that ACL2 theorem and function databases are. To use any features associated

with a trust tag, however, users must declare their intention to trust the new features (or their author). Each trust tag is associated with the file containing its associated block of code, providing careful users a means to find and inspect each block of code before declaring it trustworthy.

The correctness criteria needed for a user to trust an unverified clause processor is carefully laid out in our paper in the Journal of Applied Logic [40] and in the ACL2 documentation [37]. When a user declares the extension of ACL2 with an unverified clause processor to be trustworthy they are declaring that the unverified clause processor meets the correctness criteria. For the most part, an unverified clause processor is sound unless it produces a list of clauses that do not imply its input clause. There are some subtle issues, however, which are explained in the paper. For instance, each unverified clause processor is associated with a list of supporting functions that extend ACL2's ground zero theory, on which the correctness of the clause processor depends. If a clause processor depends on a function definition that is not listed, then extending ACL2 with it may be unsound.

11.3.1 Applications

The integration of ACL2 with other verification tools has been an area of considerable interest. ACL2 has been previously integrated with the Cadence SMV model checker [69] and the UCLID automated verification tool [44], as well with SAT solvers and the SixthSense model checker, as discussed in Chapters 7 and 10. In the past, the extension of ACL2 with an external verification tool has necessitated "hacking" the ACL2 source code, which, as evidenced by the subtlety of the unverified clause processor correctness criteria, is an error prone process. Now, an extension to ACL2 requires only a single definition of a clause processor, which has a well-documented specification.

We have used the new unverified clause processor interface to develop a clause processor for SULFA formulas, based on the SAT-based SULFA proof technique described in Chapter 7. Furthermore, we have applied the SULFA clause processor to the Load Store

Queue protocol described in Chapter 8 and used it to develop the general-purpose SMT solver described in Chapter 9. The SULFA clause processor and the general-purpose SMT solver are now distributed with the ACL2 theorem prover [37].

11.4 Unverified Clause Processors with Implicit Theories

The unverified clause processors described in Sections 11.2 and 11.3 work under the assumption that a clause processor proves properties that follow from the axioms currently in the ACL2 theorem prover's database. However, as discussed in Section 10.3.1, the integration of ACL2 with SixthSense requires the introduction of an *implicit theory*, an ACL2 theory that is not formally introduced into the ACL2 system. In particular, the ACL2SIX system contains axioms about the functions `SIGBIT` and `SIGVEC` that are derived from the hardware design, which is external to the ACL2 system.

Support for implicit theories has been created using a new mechanism called *encapsulation templates*. Function symbols introduced using encapsulation templates are similar to constrained functions introduced through the encapsulation principle described in Section 3.5.1. The difference is that the axioms given to the theorem prover in an encapsulation template are not assumed to be all the axioms about the introduced function symbols. Therefore, the functional instantiation proof technique described in Section 3.6.3 is disallowed on function symbols introduced in an encapsulation template, since functional instantiation needs a full list of axioms regarding the function symbol (or symbols) being instantiated.

If an unverified clause processor is associated with an encapsulation template, then it may prove properties about an implicit theory. The correctness criteria for an unverified clause processor with an implicit theory is explained in our paper [40], as well as in the ACL2 documentation [37]. The main idea is that it must be possible to replace the encapsulation template with an admissible encapsulation event that contains all the axioms needed for the unverified clause processor to be correct. Thus, an unverified clause processor with an implicit theory is verifying properties about an admissible ACL2 theory that is simply

not given to the theorem prover.

The ACL2SIX hint described in Chapter 10 has been implemented as an unverified clause processor with an implicit theory, called the ACL2SIX clause processor. The functions `SIGBIT` and `SIGVEC` are introduced using an encapsulation template. The justification of the soundness of the clause processor is the same as the justification provided in our FMCAD paper [77]. Essentially, there exists a potential hardware model based on the VHDL design, which contains axioms only about `SIGBIT` and `SIGVEC`, from which each theorem proven by the ACL2SIX clause processor can be derived.

11.5 Summary

We have designed and implemented a general-purpose mechanism for dynamically extending the ACL2 theorem prover with new proof techniques, called clause processors. Our general-purpose mechanism supports both verified and unverified clause processors, including unverified clause processors that extend a theory with an implicit theory, containing axioms not directly provided to the theorem prover.

Verified clause processors, unverified clause processors, and unverified clause processors with implicit theories have each been implemented. We developed a verified clause processor that uses mergesort to efficiently sort the summands in bit-vector addition operations. An unverified clause processor was implemented based on the SULFA decision procedure described in Chapter 7. Finally, an unverified clause processor with an implicit theory was developed from the SixthSense integration described in Chapter 10.

The unverified clause processors mechanism is superior to previous approaches used to integrate ACL2 with external tools. Previous attempts to integrate ACL2 with external tools required developers to create their own version of the ACL2 theorem prover, which complicates the implementation and distribution of such tools. Furthermore, we have developed a correctness criteria for unverified clause processors, which formalizes the requirements needed to ensure the soundness. Having such criteria simplifies both the pro-

cess of developing sound extensions and evaluating the potential for unsoundness risk with a given extension. The new mechanism also requires the user to declare an unverified clause processor as trustworthy before it is used. Therefore, distributions can include extensions with various levels of unsoundness risk, and users can determine the level of acceptable risk suitable for their work.

11.6 Development and Bibliographic Notes

The design of trust tags, encapsulated templates, and the new clause processor mechanisms is joint work with Matt Kaufmann, J Moore, Sandip Ray. The design was then implemented by Matt Kaufmann. Besides helping to design the mechanism, I created the initial applications for the new mechanism, including the sorting verified clause processor described in Section 11.2 and the SULFA clause processor described in Section 11.3. Sandip Ray translated my implementation of the ACL2SIX hint, described in Chapter 10, into an unverified clause processor with an implicit theory.

Some related work involves the development of "interface logics" [23], that combine automated reasoning tools by defining a single logic L such that each reasoning tool is a sub-logic of L. Similarly, Berezin's thesis involves the creation of a unified logic for combining theorem proving and model checking [5].

The PVS theorem prover contains decision procedures based on model checkers and SAT solvers [61, 12], though we are not aware of any general-purpose mechanism to integrate external tools with PVS.

Previous work with the HOL family of theorem provers [20], such as Isabelle [59] and HOL4 [26], integrates external proof tools as *oracles* using the "tagging" system introduced by Else Gunter [22]. Oracles have been used to integrate Isabelle with model checkers and arithmetic decision procedures [56, 4]. Furthermore, the PROSPER project [13] uses oracles within HOL98 to integrate several verification tools. Oracles have also been used to solve large Boolean propositional formulas within HOL4, by appealing to

eternal SAT solvers and BDD engines [18, 34]. Furthermore, the ACL2 theorem prover itself has been connected with HOL4 through an HOL4 oracle [19].

The tagging system within HOL is somewhat different from the trust tags discussed in this chapter. A tag in HOL is implemented *in the logic* as an additional hypothesis on each theorem that requires the use of a given oracle. Thus, in HOL, one can track external tool dependencies at the granularity of theorems. ACL2 trust tags, on the other hand, track dependencies at the level of files. Our approach has the disadvantage that a user cannot use only the portion of a file that does not depend on a given external tool. However, in practice, we believe this disadvantage is not likely significant, since ACL2 users already frequently move events across files. Furthermore, tracking dependencies at the level of files, at least within the ACL2 theorem prover, leads to a cleaner implementation.

Some work also exists that uses external tools to search for proofs within a more general framework. Ivy is one such system, which uses Otter to find ACL2 proofs for certain theories [46]. Hurd also describes an interface for connecting HOL with first-order logic for the purpose of finding HOL proofs more automatically [33].

Chapter 12

The DE2 Language

12.1 Introduction

This chapter provides an overview of the hardware description language, DE2, which is described in more detail in our paper in the proceedings of the Conference on Correct Hardware Design and Verification Methods (CHARME) [31].

DE2 has unique features that make it suited for formal verification:

- DE2 is a hierarchical language and supports hierarchical verification. A design, or DE2 module, is constructed by composing submodules. Similarly, the verification of a module can be constructed from the verification of its submodules.
- DE2 has a simple, two-pass semantics, in which a module is defined by two passes through its submodules. Having a simple semantics greatly simplifies the composition process. Furthermore, the DE2 language is restricted to finite state machines, so that modules can be modeled as functions that input and output an implicit state structure.
- The semantics of the DE2 language is deeply-embedded in the ACL2 logic, meaning that a DE2 description is an ACL2 constant and ACL2 functions are defined that

provide a formal semantics for the DE2 language.

- The formal semantics of DE2 also serves as an efficient evaluation and testing mechanism for DE2 circuits.

- An infrastructure has been created to automate the verification of DE2 designs in ACL2.

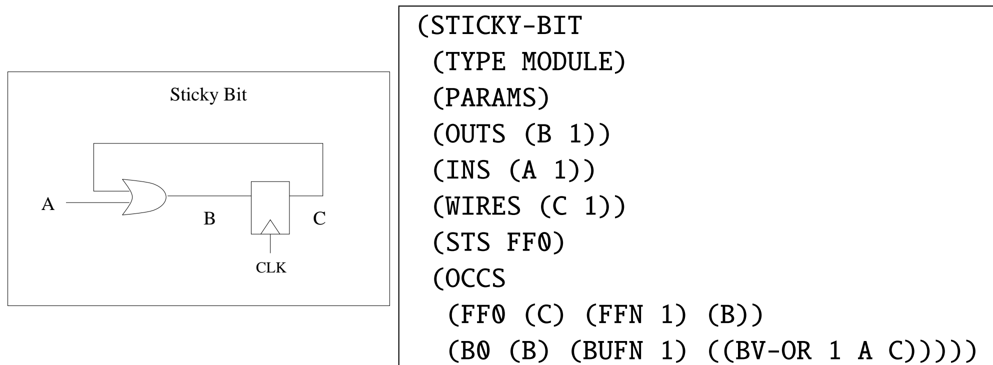
- The infrastructure also supports the verification of (possibly infinite) sets of DE2 designs, described by ACL2 functions, and the verification of optimization and simplification programs that operate on DE2 code.

- The structure of the DE2 language closely corresponds to a subset of Verilog, which enables the formal verification of Verilog circuits.

- The DE2 language is designed to be extensible. Parameters, which are inputs that must be constant in a synthesizable design, are built into the language. Parameters promote reusability by allowing a single module description to represent an infinite number of actual modules. Extensibility is also promoted by allowing primitive modules to be defined by the user, rather than built in to the language.

- Annotations are built into DE2 as first class objects, rather than comments. Thus, users can embed non-functional information directly into the design and write ACL2 functions that reason about such information. For example, information about testing, layout, and power can be embedded in annotations and reasoned about using ACL2 functions.

This chapter begins with an introductory example of a DE2 module in Section 12.2. Section 12.3 and 12.4 then describe the syntax and semantics of DE2 in more detail. Section 12.5 then overviews the DE2 verification system and Section 12.6 describes a circuit generator on which the DE2 verification system has been applied.




```
(STICKY-BIT
 (TYPE MODULE)
 (PARAMS)
 (OUTS (B 1))
 (INS (A 1))
 (WIRES (C 1))
 (STS FF0)
 (OCCS
  (FF0 (C) (FFN 1) (B))
  (B0 (B) (BUFN 1) ((BV-OR 1 A C)))))
```

Figure 12.1: A simple example circuit and its DE2 description, in which a high bit **A** causes bit **B** and bit **C** to be high for all later times.

12.2 Introductory Example

As a first example, the **sticky-bit** circuit (previously described in Chapter 10), is shown in Figure 12.1, along with its DE2 description. The **sticky-bit** circuit has a single input, `A`, a single output `B`, and an internal wire `C`.

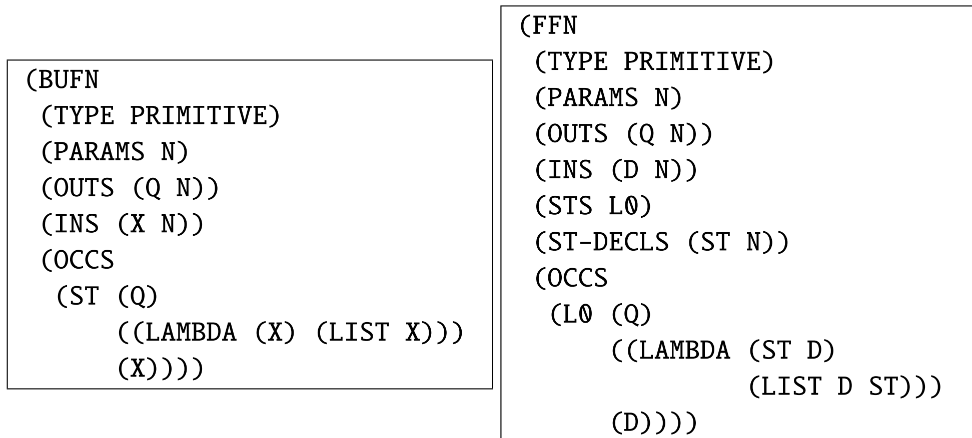
The DE2 description is an ACL2 term that specifies the name of the circuit and contains various named *annotations*, such as `TYPE`, `PARAMS`, and `OCCS`, that describe the circuit. The **sticky-bit** circuit's `OCCS` annotation describes that the **sticky-bit** module is built from two submodules, a flip-flop and a buffer. The occurrence

```
(FF0 (C) (FFN 1) (B))
```

describes that **sticky-bit** contains an instance, named `FF0`, of a 1 bit flip-flop module (an n bit flip-flop module has a description named `FFN`), with input `B` and output `C`. Similarly, the occurrence

```
(B0 (B) (BUFN 1) ((BV-OR 1 A C)))
```

describes that **sticky-bit** contains an instance, name `B0`, of a 1 bit buffer module (an n bit buffer has a description named `BUFN`) with output `B` and input



```
(BUFN
 (TYPE PRIMITIVE)
 (PARAMS N)
 (OUTS (Q N))
 (INS (X N))
 (OCCS
  (ST (Q)
      ((LAMBDA (X) (LIST X)))
      (X))))
```

```
(FFN
 (TYPE PRIMITIVE)
 (PARAMS N)
 (OUTS (Q N))
 (INS (D N))
 (STS L0)
 (ST-DECLS (ST N))
 (OCCS
  (L0 (Q)
      ((LAMBDA (ST D)
               (LIST D ST)))
      (D))))
```

Figure 12.2: DE2 descriptions of N bit buffer and flip-flop modules.

```
((BV-OR 1 A C)),
```

which is the "or" of the 1 bit, bit vectors (or wires) A and C.

The TYPE annotation is used to differentiate between primitive modules and non-primitive modules, where primitive modules do not instantiate submodules. The TYPE annotation is actually a non-functional annotation, because it is not used in the DE2 formal semantics. However, a type checker ensures that TYPE annotations are present and correct. Similar checkers and annotations can be created to contain and ensure the consistency of other information. For example, we have developed annotations to record whether signals are active-high or active-low, whether signals are data or control signals, and to record coverage points (logic that is high when some interesting case is being tested).

The descriptions of the flip-flop and buffer submodules, which are used in the **sticky-bit** module, are shown in Figure 12.2. Note that both modules contain PARAMS annotations and LAMBDA module occurrences. The PARAMS annotation enables the module descriptions to be used to create any N bit buffer or sequence of N flip-flops. The LAMBDA module is an unnamed module that, given its state and inputs, explicitly creates its state and

output using an ACL2 term (a call to the `LIST` function). Note that any state declared by the `STS` annotation, which is normally an implicit input and output of the submodule, is made explicit by the use of `LAMBDA` modules.

The `ST-DECLS` annotation is another annotation that does not affect the semantics of a module. The `ST-DECLS` annotation declares that a piece of state is implemented using a bit vector with a specific width. DE2, in principle, can be used with infinite memory models. By declaring the state to be a finite bit vector, however, the `ST-DECLS` annotation enables the use of finite-state verification tools.

As stated in Chapter 10, one property we may wish to prove about the **sticky-bit** module in Figure 12.1 is:

$$\Box(\mathbf{B} \rightarrow \Box\mathbf{B})$$

which states that once the **B** signal is high, it remains so for all time.

One way of writing this using the semantics of DE2 is:

$$(n_0 \in \mathbb{N}) \wedge (n_1 \in \mathbb{N}) \wedge (n_0 < n_1) \wedge \mathsf{stickyBitsInp}\,(\mathit{netlist}) \wedge$$
$$\mathsf{nth}\,(0, \mathsf{RunOuts}\,(n_0, \ulcorner\texttt{BV-EV}\urcorner, \ulcorner\texttt{STICKY-BIT}\urcorner, \mathit{params}, \gamma, S, \mathit{netlist}))$$
$$\rightarrow$$
$$\mathsf{nth}\,(0, \mathsf{RunOuts}\,(n_1, \ulcorner\texttt{BV-EV}\urcorner, \ulcorner\texttt{STICKY-BIT}\urcorner, \mathit{params}, \gamma, S, \mathit{netlist}))$$

where

- $\mathsf{RunOuts}\,(n, ev, m, params, \gamma, S_0, netlist)$ is part of the DE2 semantics described in Section 12.4; it returns a list of outputs of the module named m after n clock cycles, given parameters *params*, given a list of at least n cycles of inputs γ, given initial state S_0, and given a list of DE2 descriptions *netlist* including m and its submodules. The *ev* input specifies an evaluator to be used, in this case the bit vector evaluator `BV-EV`, under the substitution *env*, to evaluate ACL2 terms in the DE2 description or inputs.

- stickyBitsInp (*netlist*) is a predicate that is true when the **sticky-bit** module description shown in Figure 12.1 are in *netlist*, as well as descriptions of its required submodules shown in Figure 12.2.

- nth (i, x) is the function that returns the ith element of the list x.

The above property can then be proven using the methodology described in Section 12.5.

12.3 Formal Syntax

A circuit is described in DE2 as a *netlist* of module descriptions. Both the netlist and each module description is represented as a Lisp expression recognized by the following grammar:

$$\begin{aligned}
\textit{netlist} &::= (\textit{module-list}) \\
\textit{module-list} &::= \epsilon \mid \textit{module module-list} \\
\textit{module} &::= (\textbf{symbol}\ \textit{module-body}) \\
\textit{module-body} &::= \epsilon \mid \textit{annotation module-body} \\
\textit{annotation} &::= (\textbf{symbol expression})
\end{aligned}$$

where **expression** is an arbitrary Lisp expression, such as `((A B) (C D E) F)`.

The grammar in Figure 12.3 recognizes the built-in annotations, `PARAMS`, `OUTS`, `INS`, `WIRES`, `STS`, and `OCCS`. Intuitively, the built-in annotations describe a module's submodules and wiring.

12.4 DE2 Semantics

The state of a machine with top-level module m after n steps, or cycles, is defined as:

```
   annotation ::= (PARAMS sym-list) | (OUTS decl-list) | (INS decl-list) |
                  (WIRES decl-list) | (STS sym-list) | (OCCS occ-list)

    decl-list ::= ϵ | (symbol param) decl-list

     occ-list ::= ϵ | occurrence occ-list
   occurrence ::= (symbol occ-out-list (symbol param-list) wire-list) |
                  (symbol occ-out-list (lambda-module param-list) wire-list)

lambda-module ::= ((LAMBDA (sym-list) (LIST expr-list)) param)

 occ-out-list ::= ϵ | occ-out occ-out-list
      occ-out ::= symbol | (symbol param param)

    expr-list ::= ϵ | wire expr-list | param expr-list

    wire-list ::= ϵ | wire expr-list
         wire ::= symbol | constant | (N-NILS param) |
                  (BV-CONST param param) | (BV-BIN-CONST param param) |
                  (BV-IF wire wire wire) | (UNARY-AND param wire) |
                  (UNARY-OR param wire) | (BV-AND param wire wire) |
                  (BV-OR param wire wire) | (BV-NOT param wire) |
                  (BV-XOR param wire wire) | (BV-EQ param wire wire) |
                  (BV-LEQ param wire wire) | (BV-DECODE param wire) |
                  (GET-SUBLIST wire param param) | (G wire param param) |
                  (APPEND-N param wire wire) | (A-N param wire wire) |
                  (BV-DUPLICATE param param wire) | (BV-ADD param wire wire) |
                  (UPDATE-SUBLIST wire param param wire) |
                  (US wire param param wire)

   param-list ::= ϵ | param expr-list
        param ::= symbol | number | (1- param) | (1+ param) |
                  (+ param param) | (- param param) | (EXPT param param)

     sym-list ::= ϵ | symbol sym-list
```

Figure 12.3: A grammar specifying the syntax of the built-in annotations, used to specify the functionality of a DE2 module. Note that **symbol** is an arbitrary symbol and **number** is an arbitrary number.

$$\mathsf{RunSt}\,(n, ev, m, params, \gamma, S_0, netlist) \triangleq \begin{cases} \texttt{st}, & \text{if } n = 0 \\ \mathsf{de}\,(ev, m, params, \gamma_{n-1}, S_{n-1}, env, netlist), & \text{otherwise.} \end{cases}$$

where

- S_{n-1}, is the recursively computed machine state at time $n-1$:

$$S_{n-1} \triangleq \mathsf{RunSt}\,(n-1, ev, m, params, \gamma, S_0, netlist).$$

- Similarly, S_0 is the initial state of the machine, denoted as an ACL2 term, to be evaluated using the *ev* evaluator under the substitution *env*.

- *ev* is a symbol specifying an evaluator for ACL2 terms. This chapter assumes ev = ⌜`BV-EV`⌝, specifying the bit-vector evaluator. Note that a *wire* expression in Figure 12.3 recognizes an ACL2 term involving bit-vector primitives. If a different evaluator were used, then a different set of primitives would be recognized. The `BV-EV` evaluator is also used to evaluate simple arithmetic terms, including those needed to evaluate the *param* expressions in Figure 12.3.

- *env* is a substitution mapping symbols to constants.

- *netlist* is an ordered list of DE2 module descriptions, including the descriptions of *m* and all its submodules.

- *params* is a list of ACL2 terms corresponding to the module's `PARAMS` annotation. Each term evaluates to a natural number, when evaluated by the *ev* evaluator under the substitution *env*.

- γ contains all the inputs to the module such that γ_i is a list of inputs, corresponding to the module's `INS` annotation, after *i* clock cycles. Each input is an ACL2 term that can be evaluated to a constant using the *ev* evaluator under the substitution *env*.

- $\mathsf{de}(ev, m, params, ins, S, env, netlist)$ is the state of the machine m described by netlist *netlist* at the next cycle, given its parameters *params*, inputs *ins*, and state S (all ACL2 terms evaluated by the *ev* evaluator under the *env* substitution) during the current cycle.

The output of machine m after n cycles is defined as

$$\begin{aligned} & \mathsf{RunOuts}(n, ev, m, params, \gamma, S_0, netlist) \\ \triangleq \ & \\ & \mathsf{se}(ev, m, params, \gamma_n, S_n, env, netlist) \end{aligned}$$

where

- ev, S_0, $params$, γ, and *netlist* are the same as in RunSt.
- S_i is the machine state after i clock cycles, defined as:

$$S_i \triangleq \mathsf{RunSt}(i, ev, m, params, \gamma, S_0, netlist).$$

- $\mathsf{se}(ev, m, params, ins, S, env, netlist)$ is the output list of the machine m described by netlist *netlist* at the next cycle, given its parameters *params*, inputs *ins*, and state S (all ACL2 terms evaluated by the *ev* evaluator under the *env* substitution) during the current cycle.

Thus, the semantics of a module are defined through the functions:

$$\mathsf{se}(ev, m, params, ins, S, env, netlist)$$
$$\mathsf{de}(ev, m, params, ins, S, env, netlist)$$

which return the outputs and next state of a module.

Intuitively, given the inputs and state of a DE2 module, the value of each wire can be computed through a linear traversal of the submodule occurrences, as given by the `OCCS` annotation. The se function returns the module outputs by computing the wire values and then taking the subset of wires specified by the `OUTS` annotation. The de function returns a

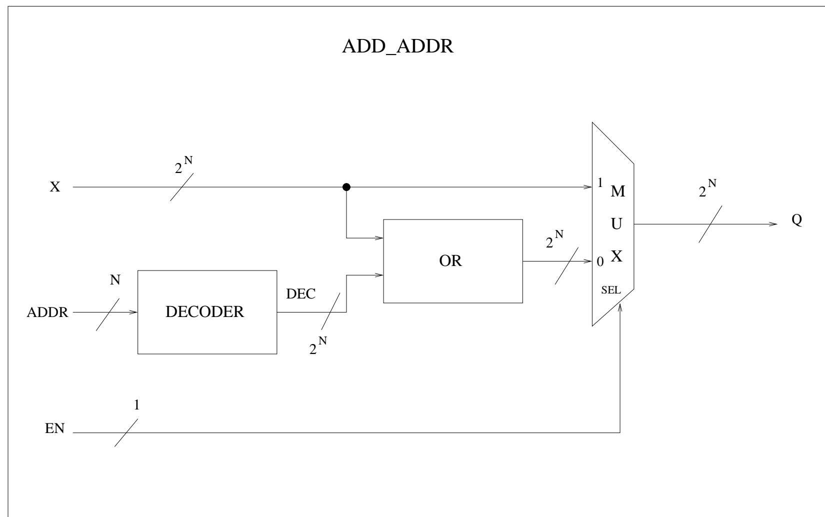


Figure 12.4: The **ADD_ADDR** circuit adds (by bitwise disjunction) the block addressed by **ADDR** to the mask **X**, when **EN** is high. The DECODER unit translates an n bit, bit vector into 2^n bit, bit vector with only the nth bit high. For example, given a four bit, bit vector with value 3, the DECODER outputs a sixteen bit, bit vector with value 8 (the third bit is high, the rest low).

DE2 module's state by computing the wire values and then making a second, "dual" pass, through the submodule occurrences, in which the state of each submodule is updated. For example, in Figure 12.1, the wire `B` is computed during the first pass and, in the second pass, the `FF0` submodule occurrence updates its internal state using the new value of `B`. A more detailed description of the `se` and `de` functions can be found in our CHARME paper [31].

As another example, the **ADD_ADDR** circuit, has the schematic shown in Figure 12.4 and the Verilog and DE2 descriptions shown in Figure 12.5. The **ADD_ADDR** circuit is a slight generalization of a circuit present in the implementation of the TRIPS Load-Store Queue Protocol, described in Chapter 8. It adds an address to its input mask, when its enable input `EN` is high. It uses a decoder submodule to translate the n bit input address into a 2^n bit, bit vector, in which only the nth bit is high. Then a multiplexer selects

```
module ADD_ADDR (Q,X,EN,ADDR);

parameter N = 3;

output [0:(1<<N)-1] Q;

input [0:(1<<N)-1] X;
input [0:N-1] ADDR;
input EN;

wire [0:(1<<N)-1] DEC;

D0 #(N) decoder(ADDR);

assign Q = EN ? DEC | X : X;

endmodule
```

```
(ADD_ADDR

 (PARAMS N)

 (OUTS (Q (EXPT 2 N)))

 (INS (X (EXPT 2 N))
      (ADDR N)
      (EN 1))

 (WIRES (DEC (EXPT 2 N)))

 (OCCS
  (D0 (DEC) (DECODER N) (ADDR))

  (B0 (Q) (BUFN (EXPT 2 N))
      ((BV-IF EN
              (BV-OR (EXPT 2 N)
                     DEC
                     X)
              X)))))
```

Figure 12.5: A Verilog and a DE2 description of the ADD_ADDR circuit shown in Figure 12.4.

```
(TWO-INV
 (OUTS (Q0 1) (Q1 1))
 (INS (A 1) (B 1))

 (OCCS
  (A0 (Q0) (BUFN 1) (BV-NOT 1 A))
  (A1 (Q1) (BUFN 1) (BV-NOT 1 B))))
```

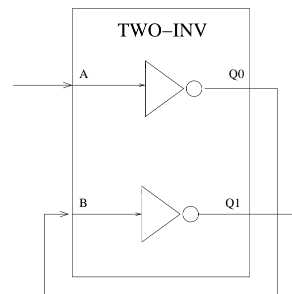


Figure 12.6: The DE2 description on the left implements the internals of the TWO-INV module drawn on the right. The schematic on the right further uses the TWO-INV module to implement a one bit buffer. No DE2 description, however, can be created that wires the TWO-INV module to itself to create such a buffer.

whether to add the decoded signal into the mask, based on the enable bit. Parameters are used to generalize the input and output bit widths.

Note the close correspondence between the Verilog and DE2 descriptions in Figure 12.5. The syntax and semantics of DE2 correspond closely to a restricted subset of Verilog. This enables many Verilog circuits to be easily translated into DE2 and simplifies the Verilog to DE2 compiler described in Section 12.5. It also makes compilation from DE2 to Verilog trivial.

12.4.1 Limitations of the Two Pass Model

The fact that all wire values are calculated in a single traversal of the submodule occurrences greatly simplifies the DE2 semantics and thus the verification of DE2 modules. However, it also restricts the circuits that can be written in DE2. If the output of a submodule *A* is wired to the input of a submodule *B*, then *A* must occur prior to *B* in the list of submodule occurrences.

A combinational loop in a hardware designs prevents it from being represented in DE2. In practice, this restriction is not severely limiting, since combinational loops rarely appear in hardware designs. When combinational loops do occur, they can be modeled as

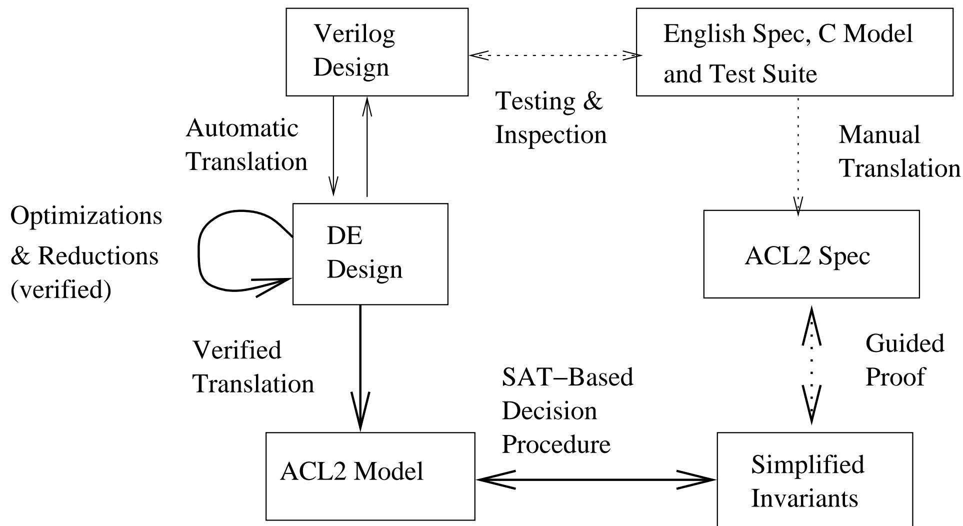


Figure 12.7: An overview of the DE2 verification system.

ACL2 functions evaluated by the *ev* evaluator.

The submodule ordering restriction within DE2, however, rules out more than combinational loops. For example, a DE2 description of the `TWO-INV` module is shown on the left in Figure 12.6 and its schematic is shown on the right. The `TWO-INV` module simply contains two independent inverters. Each output of the module is the inversion of one of its inputs. The DE2 description shown on the left of Figure 12.6, is a valid DE2 description, but it cannot be wired to implement the buffer shown in the schematic on the right. Even though the schematic clearly has no combination loops, the ordering restriction prevents it from being described because the output of an instantiation of `TWO-INV` cannot affect its input. The only way to create a DE2 description of the schematic in Figure 12.6 is to break the `TWO-INV` module in two, so the inverters may be ordered.

12.5 The DE2 Verification System

Having a semantics for DE2 written in the ACL2 logic enables many forms of verification. In Figure 12.7, we illustrate our verification system, which is built around the DE2

language.

The DE2 verification system can be used to verify Verilog designs, which are denoted in the upper left of Figure 12.7. We have developed a compiler that automatically translates a subset of Verilog to DE2 descriptions, which enables the verification of designs of practical interest, such as the Load Store Queue protocol described in Chapter 8. Furthermore, DE2 descriptions can also be compiled automatically into Verilog descriptions, which enables DE2 descriptions to use traditional synthesis and timing tools that operate on Verilog.

To assist in the verification of DE2 designs, a *verifying compiler* has been developed from DE2 descriptions to cycle-accurate ACL2 models of those descriptions. Here, *verifying compiler* means a compiler that produces, along with its output, a proof that its output is equivalent to its input. For each design it compiles, our verifying compiler produces an ACL2 proof that the DE2 description input to it is equivalent, according to the semantics of DE2, to the cycle-accurate ACL2 model it produces. The ACL2 model produced is essentially a model in first-order logic, where each wire in a DE2 module has a corresponding function. Furthermore, a function produces the next state of the finite state machine described by each module from its previous state. Some simple reductions, such as cone-of-influence are performed during the translation process, and these are verified by the ACL2 theorem prover on a case-by-case basis.

The specification of the design begins in the form of English documents, charts, graphs, C-models and test code, which is represented in the upper right of Figure 12.7. This informal specification is then translated manually into a formal ACL2 specification. Through user-guided proof, the formal specification is reduced to invariants and equivalence properties in SULFA. If possible, the SULFA properties are verified automatically by the SAT-based procedure in Chapter 7. If the SAT-based procedure, due to the state explosion problem, fails to verify the SULFA properties, then they are further broken down by user-guided proof until they can be successfully verified by some automatic procedure.

The DE2 verification system can also be used to verify circuit generators, such as the ripple-carry adder generator described in Section 12.6. The circuit generators are implemented as ACL2 functions that produce DE2 code. By proving, through the DE2 semantics, that the ACL2 functions always produce correct code we verify a potentially infinite number of implementations in a single verification effort.

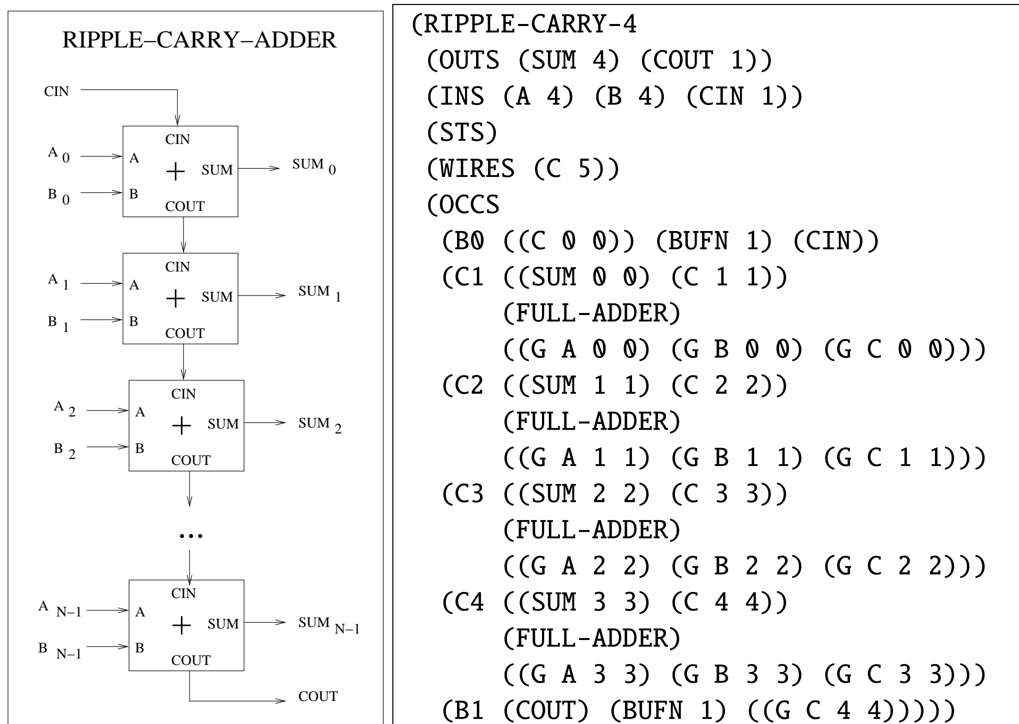
Similarly, ACL2 functions that modify DE2 descriptions can also be verified. Such functions can be used to optimize the DE2 implementation or simplify it for future verification. Thus it is possible to use the DE2 verification system to translate a Verilog design to DE2, run verified optimizations on it, and then translate it back into Verilog.

It is also possible to build static analysis tools, such as extended type checkers, that check the correctness of DE2 annotations. DE2 annotations are first-class objects (i.e., they are not embedded in comments) that can include all manner of functional and non-functional properties and parameters relating to a DE2 module. An extended type checker can then input the DE2 module and read its annotations. The type checker can be written in ACL2 and can therefore be analyzed and used within the verification effort. For example, the `ST-DECLS` annotation is used to produce a finite memory model that enables the use of finite-state verification tools.

12.6 Circuit Generator Example

This section describes circuit generators and their verification through a simple ripple-carry adder example. Figure 12.8 illustrates the schematic of an *n* bit ripple-carry adder, built from full-adders. Every submodule in a DE2 module must exist in all implementations of that module, regardless of parameters. Therefore, the *n* bit ripple-carry adder in Figure 12.8 cannot be described as a parameterized DE2 module. A DE2 module can, however, describe an *n* bit ripple-carry adder for any specific value of *n*. For example, the DE2 description of a 4 bit ripple-carry adder is shown on the right in Figure 12.8.

To reason about the general *n* bit ripple-carry adder, an ACL2 function can be cre-




```
(RIPPLE-CARRY-4
 (OUTS (SUM 4) (COUT 1))
 (INS (A 4) (B 4) (CIN 1))
 (STS)
 (WIRES (C 5))
 (OCCS
  (B0 ((C 0 0)) (BUFN 1) (CIN))
  (C1 ((SUM 0 0) (C 1 1))
      (FULL-ADDER)
      ((G A 0 0) (G B 0 0) (G C 0 0)))
  (C2 ((SUM 1 1) (C 2 2))
      (FULL-ADDER)
      ((G A 1 1) (G B 1 1) (G C 1 1)))
  (C3 ((SUM 2 2) (C 3 3))
      (FULL-ADDER)
      ((G A 2 2) (G B 2 2) (G C 2 2)))
  (C4 ((SUM 3 3) (C 4 4))
      (FULL-ADDER)
      ((G A 3 3) (G B 3 3) (G C 3 3)))
  (B1 (COUT) (BUFN 1) ((G C 4 4)))))
```

Figure 12.8: A general schematic for an *n* bit ripple-carry adder is shown on the left. Its DE2 description in the 4 bit case is shown on the right.

ated to produce its DE2 description. To write such a function, first recall that a DE2 description is a list of modules. We thus define the following functions to manipulate lists:

- MakeDE $(x_0, x_1, ..., x_N)$ is the Lisp expression created by composing the Lisp expressions x_0 through x_N. For example,

  $$\mathsf{MakeDE}(\ulcorner\texttt{(A B)}\urcorner, \ulcorner\texttt{C}\urcorner, \ulcorner\texttt{D}\urcorner) = \ulcorner\texttt{((A B) C D)}\urcorner.$$

  Note that MakeDE is equivalent to Lisp's `LIST` function, but this dissertation uses MakeDE, rather than `LIST`, in order to distinguish between mathematical lists and a Lisp expressions.

- AppDE (x, y) is the Lisp expression created by appending the Lisp expression y to the end of the Lisp expression x. For example,

  $$\mathsf{AppDE}(\ulcorner\texttt{(A B)}\urcorner, \ulcorner\texttt{(C)}\urcorner) = \ulcorner\texttt{(A B C)}\urcorner.$$

  Note that both x and y must be enclosed in parentheses, i.e., they cannot be constants or symbols. AppDE (x, y) is equivalent to Lisp's `APPEND` function.

- SymName (s, n), given a Lisp symbol s and a natural number n, produces the symbol with n appended to s. For example,

  $$\mathsf{SymName}(\ulcorner\texttt{B}\urcorner, 33) = \ulcorner\texttt{B33}\urcorner$$

To define the n bit ripple-carry adder in DE2, we first define the function GenRipOcc (n), which returns the nth full-adder occurrence within the ripple-carry adder's `OCCS` annotation:

$\mathsf{GenRipOcc}\,(n)$

$\triangleq$

$\mathsf{MakeDE}\,(\,\mathsf{SymName}\,(\ulcorner\mathtt{C}\urcorner, n),$
$\quad\quad \mathsf{MakeDE}\,(\,\mathsf{MakeDE}\,(\ulcorner\mathtt{SUM}\urcorner, n-1, n-1), \mathsf{MakeDE}\,(\ulcorner\mathtt{C}\urcorner, n, n))$
$\quad\quad \mathsf{MakeDE}\,(\ulcorner\mathtt{FULL\text{-}ADDER}\urcorner),$
$\quad\quad \mathsf{MakeDE}\,(\,\mathsf{MakeDE}\,(\ulcorner\mathtt{G}\urcorner, \ulcorner\mathtt{A}\urcorner, n-1, n-1),$
$\quad\quad\quad\quad \mathsf{MakeDE}\,(\ulcorner\mathtt{G}\urcorner, \ulcorner\mathtt{B}\urcorner, n-1, n-1),$
$\quad\quad\quad\quad \mathsf{MakeDE}\,(\ulcorner\mathtt{G}\urcorner, \ulcorner\mathtt{C}\urcorner, n-1, n-1)))$

For example, $\mathsf{GenRipOcc}\,(3)$ returns:

```
(C3 ((SUM 2 2) (C 3 3))
    (FULL-ADDER)
    ((G A 2 2) (G B 2 2) (G C 2 2)))
```

which instantiates a full adder submodule, with module description named `FULL-ADDER`, and names this occurrence as `C3`. The expressions `(G A 2 2)`, `(G B 2 2)`, and `(G C 2 2)` are ACL2 terms returning the second bit of bit vectors named `A`, `B`, `C` respectively. The `C3` submodule receives these inputs and outputs its sum and carry bits onto the second bit of the bit vector named `SUM` and the third bit of the bit vector named `C`.

The following function composes m ripple-carry adder occurrences to form the `OCCS` annotation for an n bit ripple-carry adder module.

$\mathsf{GenRipOccs}\,(m, n)$

$$\triangleq \begin{cases} \mathsf{MakeDE}\,(\mathit{firstOcc}), & \text{if } m = 0 \\ \mathsf{AppDE}\,(\mathsf{GenRipOccs}\,(m-1, n), \mathsf{MakeDE}\,(\mathit{lastOcc})) & \text{if } m = n+1 \\ \mathsf{AppDE}\,(\mathsf{GenRipOccs}\,(m-1, n), \mathsf{MakeDE}\,(\mathsf{GenRipOcc}\,(m))), & \text{otherwise.} \end{cases}$$

where

- $\mathit{firstOcc} \triangleq \ulcorner$`(B0 ((C 0 0)) (BUFN 1) (CIN))`$\urcorner$.

- $lastOcc \triangleq$ MakeDE (⌜B1⌝, ⌜(COUT)⌝, ⌜(BUFN 1)⌝, $lastOccIns$).

- $lastOccIns \triangleq$ MakeDE (MakeDE (MakeDE (⌜G⌝, ⌜C⌝, n, n))).

For example, GenRipOccs $(2, 1)$ is:

```
((B0 ((C 0 0)) (BUFN 1) (CIN))
 (C1 ((SUM 0 0) (C 1 1))
     (FULL-ADDER)
     ((G A 0 0) (G B 0 0) (G C 0 0)))
 (B1 (COUT) (BUFN 1) ((G C 1 1))))
```

The following function generates the description for an n bit ripple-carry adder:

genRipple (n)

$\triangleq$

MakeDE (SymName (⌜RIPPLE-CARRY-⌝, n),
 MakeDE (OUTS, MakeDE (SUM, n), MakeDE (COUT, 1)),
 MakeDE (INS, MakeDE (A, n), MakeDE (B, n), MakeDE (CIN, 1)),
 MakeDE (WIRES, MakeDE (CARRY, n)),
 AppDE (MakeDE (OCCS), GenRipOccs $(n + 1, n)$))

The DE2 description shown in Figure 12.8 is equal to the four bit ripple-carry adder, genRipple (4). To complete the description of the ripple-carry adder a netlist must be formed including both genRipple (4) and a DE2 description of the FULL-ADDER submodule.

The specification of the ripple-carry adder can be written as:

$(n \in \mathbb{N}) \wedge (n > 0) \wedge \text{RippleInp}(n, netlist) \wedge \text{bvp}(n, a) \wedge \text{bvp}(n, b)$

$\rightarrow$

$\text{bvToNat}(n, sum) = (\text{bvToNat}(n, c_0) + \text{bvToNat}(n, a) + \text{bvToNat}(n, b)) \mod 2^n$

where

- $sum \triangleq \mathsf{nth}(0, \mathsf{se}(\ulcorner\texttt{BV-EV}\urcorner, ripName, params, ins, S, env, netlist))$, which is the SUM output of the ripple carry adder module.
- $ripName \triangleq \mathsf{SymName}(\ulcorner\texttt{RIPPLE-CARRY-}\urcorner, n)$, which is the name of the ripple carry adder module.
- a is the evaluation, using the bit-vector evaluator, 'BV-EV, of the first input ($\mathsf{nth}(0, ins)$), under the substitution *env*.
- b is the evaluation, using the bit-vector evaluator, 'BV-EV, of the second input ($\mathsf{nth}(1, ins)$), under the substitution *env*.
- c_0 is the evaluation, using the bit-vector evaluator, 'BV-EV, of the third input ($\mathsf{nth}(2, ins)$), under the substitution *env*. This input is the one bit carry input.
- $\mathsf{bvToNat}(x)$ is the natural number corresponding to the bit vector x.
- $\mathsf{bvp}(n, x)$ is the predicate that is true when x is an n bit, bit vector.
- $\mathsf{RippleInp}(n, netlist)$ is the predicate that is true when *netlist* is a list containing the DE2 description of the n bit ripple-carry adder created by $\mathsf{genRipple}(n)$, as well as all its submodule descriptions.

Intuitively, the specification above states that the output of any n bit ripple-carry adder is the summation of its inputs. It is then proven using the ACL2 theorem prover, from the definitions of ripple carry adder generator $\mathsf{genRipple}(n)$, and the formal semantics of DE2, specified by the functions se and de.

12.7 Summary

DE2 is a new language for designing and specifying finite state machines. Its embedding within the ACL2 theorem prover provides a means to formally verify that these machines

satisfy their specifications. In this chapter, we show how to verify some small hardware design examples. Chapter 8 provides a more significant example, where a component of the TRIPS processor [9] is translated from Verilog to DE2 and then verified in ACL2.

Since DE2 programs are ACL2 constants, programs that generate and manipulate DE2 circuits can also be verified. We have shown how one such circuit generator, which generates a ripple-carry adder, has been verified.

Furthermore, in industrial hardware designs, all kinds of design data are presently being added into the code as comments. This process prevents there from being a single design description that is understandable by all the pre- and post-silicon development tools. DE2 incorporates all the design data into a single annotation formalism, which ACL2 functions can analyze and manipulate. We believe DE2 is the first language to incorporate all the design data into a single formalism.

12.8 Development and Bibliographic Notes

The parser used in our Verilog to DE2 compiler was created by Vinod Viswanath.

The DE2 language is a successor to the DUAL-EVAL hardware description language in NQTHM[8] and the DE language in ACL2 [28]. The DE language differs from its predecessors in that it supports user-defined primitives, reusable libraries, parameters and annotations. DE2 also structures state-holding elements in a different manner than its predecessors. The DE2 language also includes a type system and a more automated verification system.

In other hardware verification efforts with ACL2, hardware descriptions have been translated into ACL2 models in the shallow-embedding style [16, 75]. In a first-order theorem prover, however, only a deep-embedding style can reason on functions that operate on hardware descriptions, such as programs that automatically optimize, simplify or analyze hardware descriptions. Furthermore, deep-embedding is somewhat more rigorous, since designs are expressed in a way that more closely matches the actual Verilog or VHDL. For

example, a typical shallow-embedding approach in ACL2 might express hardware designs using the functional model produced by our DE2 to ACL2 compiler. In our system, the DE2 to ACL2 compiler produces a proof of correctness, which would be impossible if we did not have a formal semantics for the input language.

The notion of shallow-embedding v. deep-embedding styles originate from comparisons between different approaches to embed hardware description languages into the HOL family of theorem provers [66]. Initially, the ELLA and SILAGE languages were embedded into HOL using the shallow-embedding style, and a subset of VHDL was embedded using the deep-embedding style. More recently, the hardware description language of the MDG automated proof system was deeply embedded in HOL and used to verify hardware designs [64].

There has also been considerable interest within the functional language community in the development of higher-order hardware description languages. Such languages have the potential to automate low-level optimization techniques on larger designs. For example, the WIRED language has been shown to improve the performance of multipliers by incorporating layout information into the design of circuit generators [3].

The higher-order functional languages Lifted-FL [2] and reFLect [24] have been created at Intel, and also combine theorem proving and fully-automated verification technologies. Differences between DE2 and these language includes DE2's simpler, two-pass, semantics, its minimal syntax, and its close correspondence to a subset of Verilog.

Chapter 13

Future Directions

13.1 Introduction

There is a large number of directions in which one could continue the work described in this dissertation. This chapter presents a brief overview of a few, primarily focusing on future extensions and applications of the SULFA procedure described in Chapter 7.

13.2 Expanding SULFA

Originally, SULFA consisted of ACL2 formulas that could be unrolled into the ACL2 core primitives **if**, **cons**, **car**, **cdr**, and **consp**. Later, SULFA was expanded to include **equal** and uninterpreted functions. It is obviously advantageous to recognize as large a decidable subclass of ACL2 formulas as possible. Thus, one avenue of future work is to continue to expand SULFA.

One way to expand SULFA is to add more core primitives from Table 3.1. Since arithmetic is included in ACL2's core primitives, and arithmetic is undecidable, there is no decision procedure for formulas made up of all ACL2 core primitives. Nevertheless, important subclasses, such as linear arithmetic, could be included.

Another possible SULFA expansion is to include a larger subset of constrained ACL2 functions. Currently, only uninterpreted functions are supported, i.e., constrained functions with no constraints. However, some constraints can clearly be allowed without sacrificing decidability.

13.3 Improving Efficiency

The SAT-based procedure described in Chapter 7 is sufficiently efficient to be applied to problems of practical interest, such as the data-tile protocol in Chapter 8. Performance, however, could likely be improved through a number of known optimizations. The two most promising involve common subexpression elimination and abstraction-refinement loops.

Recall that the algorithm in Chapter 7 creates variables for ACL2 expressions and that these variables eventually become part of the CNF formula given to SAT. All other things being equal, it is advantageous to produce a CNF formula with as few variables as possible. Common subexpression elimination is an optimization that reduces the number of variables in an expression, and sometimes its translation time to CNF, by creating identical variables for identical expressions. For example, it is usually more efficient to translate $\mathsf{f}(a = b, a = b)$ into $\mathsf{f}(x, x)$, where $x \triangleq (a = b)$, than to translate $\mathsf{f}(a = b, a = b)$ into $\mathsf{f}(x, y)$, where both $x \triangleq (a = b)$ and $y \triangleq (a = b)$. Similarly, if $\mathsf{g}(a, b)$ unrolls to $a = b$, then it is best to translate $\mathsf{f}(a = b, \mathsf{g}(a, b))$ into $\mathsf{f}(x, x)$ and define x to be $a = b$.

Note that common subexpression elimination has an inherently inside-out nature. An expression should not be considered until its common subexpressions have been eliminated. For example, a variable should not be created for $\mathsf{f}(a = b, a = b)$, but rather a variable v should be created for $a = b$ and then for $\mathsf{f}(v, v)$. Since the algorithm in Chapter 7 is primarily an outside-in algorithm, common subexpression elimination is currently omitted. A simple preprocessor that eliminates common subexpressions, like the one used in Chapter 9, would likely improve performance of the SULFA solver on many SULFA formulas. A more clever form of common subexpression elimination, incorporated into the

SAT-based procedure itself, may yield even better results.

Another optimization worth noting involves abstraction-refinement loops. The idea here is to perform abstraction when confronted with a difficult verification problem. If the abstract formula is valid, then the original formula is also valid. If the counter example produced is also a counter example to the original formula, then the original formula is invalid. Otherwise, some form of refinement, likely based on the counter-example, is performed to create a less abstract problem. Abstraction-refinement loops can be very useful when dealing with shallow problems over large data structures. Also, they can be necessary to avoid explosion in formula size, when that explosion is not likely to yield a true counter-example. For example, given an uninterpreted function $\mathsf{f}(x)$, if $\mathsf{f}(x)$ and $\mathsf{f}(y)$ occur in an expression and $x = y$ does not, it is probably best to assume that $x \neq y$ so that $\mathsf{f}(x)$ and $\mathsf{f}(y)$ can be given independent variables, unless only counterexamples where $x = y$ are found.

13.4 Verification of Larger Hardware Modules

Another area of future work is to apply our methodology to larger hardware designs. Currently, for the most part, only floating-point units in industrial processors have been formally verified. In order to make formal methods practical for full-scale industrial hardware verification, we must increase our expertise with formal methods outside the floating-point unit. The data-tile protocol verification effort described in Chapter 8 is one such effort, but this effort could be continued to verify more of the data tile. Also, some other hardware designs, such as routers, have never been formally verified.

13.5 Undecidable Domains

Developing decision procedures and identifying decidable classes of problems is a useful endeavor since it can help direct a verification effort into a domain where fully-automated procedures are more applicable. Ultimately though, our goal is to reduce the amount of user

guidance required during large-scale verification, regardless of whether the formula to be verified can be easily placed into a decidable class. Thus we are interested in techniques that automatically reduce undecidable problems into decidable domains. One such technique is to use an abstraction-refinement loop, like the one proposed in Section 13.3. Abstraction can be used to reduce undecidable formulas into a decidable domain, then refinement can be used to attempt to hone in on a valid abstraction or a real counter-example. Such an approach is unlikely to discover any proof that requires induction, but, if it is well designed, it may prove to be very good at finding counterexamples to invalid conjectures.

Some of the techniques in this dissertation may also prove applicable to general-purpose simplification. An important intuition behind SAT solving and SAT-based procedures is to put off any exponential time searching or transformations until the last possible moment. In this manner, easier options are explored first and more information is available when exponential time searching or transformations are finally undertaken. The ACL2 simplifier, and other general-purpose simplifiers, do not always use this strategy. For example, the ACL2 simplifier expands lambda expressions (β reduction) and performs case splitting relatively early in the simplification process, both of which require exponential time. It would be interesting to explore the development of a general-purpose simplifier that puts off β reduction and case splitting until the last possible moment, when the best educated guess can be made as to which one should be performed first.

13.6 Verified SAT-Based Procedure

Some SAT solvers, such as zChaff, can produce a proof of unsatisfiability after determining that a problem is unsatisfiable [93]. One integration of HOL and SAT uses this proof to check results from the SAT solver with the HOL theorem prover [91]. A difficulty, however, is that the HOL theorem prover, like the ACL2 theorem prover, is designed to solve large scale problems through a hierarchy where only a small portion of the problem needs to be considered at once. In SAT solvers, no such hierarchical information is kept. Thus, only

relatively small results from SAT solvers can be checked with HOL. One possible way to address this within ACL2 is to develop a specialized proof checker, prove it correct using the ACL2 theorem prover, and then use ACL2's fast evaluation mechanism to check large problems.

Once the unsatisfiability of a Boolean CNF formula can be verified by ACL2, the translation from the original ACL2 problem to Boolean CNF must also be somehow verified. The SAT-based procedure for solving problems involving the core primitives **if**, **cons**, **car**, **cdr**, **consp**, and **equal** might be verified and introduced as a verified clause processor, as described in Chapter 11. The removal of uninterpreted functions also might be verified in such a manner. The unrolling of functions would require a different approach though since its correctness and termination follows from the (user-extendable) definitional axioms and the proofs of termination of the functions it is unrolling.

Chapter 14

Conclusion

Currently, the floating-point units of many processor designs are formally verified [60, 72], but formal verification is not attempted on most other parts of industrial designs. We believe one cause of this disparity is the relative maturity of formal verification techniques targeted specifically to floating-point units. In this dissertation, we have made advances to the field of formal verification to improve general-purpose formal verification techniques and to develop expertise in formal verification beyond the floating-point unit.

- One method to potentially reduce the cost of scalable hardware verification methodologies, such as those used on the proof of the FM9801 processor [75], is to increase the amount of search used to find proofs automatically during interactive theorem proving. The forward chaining proof strategy has therefore been modified to increase the likelihood that a theorem marked as a forward chaining rule will be used automatically by the ACL2 theorem prover. The modification increases the number of theorems proven automatically at a cost of about 2 percent in the time required to construct a proof. The modified proof strategy is now part of the default forward chaining proof technique used in the ACL2 theorem prover.
- Another method to reduce verification cost is through the integration of fully-

automated formal verification techniques with interactive theorem proving. By identifying the Subclass of Unrollable List Formulas in ACL2 (SULFA) and proving it decidable, we have found a class of problems within an interactive theorem prover on which fully-automated techniques may be applied.

SULFA is an interesting subclass in that it can be recognized efficiently and is built directly from the ACL2 core primitives—the small set of ACL2 primitives listed in Table 3.1, from which all other ACL2 functions are defined. Furthermore, unlike most identified decidable subclasses within first-order logic, SULFA contains a mechanism for extension with new functions and their defining axioms while maintaining decidability. Many useful properties are in SULFA, including 1,249 formulas in the ACL2 regression suite (3.2 percent). Also, finite-state machine models can be constructed such that any property concerning only a finite number of steps of the machine is in SULFA.

We have also developed an efficient recognizer for SULFA formulas and a SULFA solver that proves and disproves SULFA formulas automatically by translating them into the Boolean conjunctive normal form (CNF) required by SAT solvers. This tool is now available as part of the standard distribution of the ACL2 theorem prover.

- The data-tile protocol implementation within the TRIPS processor was verified using a mixture of the SAT-based SULFA solver and interactive theorem proving. The verification strategy reduces safety and liveness properties into finite-step SULFA properties. Significant human effort was avoided by using the SAT-based SULFA solver instead of verifying the SULFA properties using interactive theorem proving. Furthermore, the data-tile protocol is a novel component required by TRIPS multi-tile architecture, which addresses up-coming challenges in computer architecture. By verifying it, we have helped to build formal verification expertise in a new type of hardware design, beyond the floating-point units verified formally in industry today.

- An SMT solver has also been developed for the standard SMT theory of bit vectors by combining the ACL2 simplifier and the SAT-based SULFA solver. This SULFA SMT solver is able to verify all 8,246 problems in the SMT 2006 QF_UFBV32 benchmark suite. Furthermore, the SULFA SMT solver provides a greater degree of flexibility than traditional SMT solvers—the SULFA SMT solver can be extended with new primitives and rewrite rules, which are verified by interactive theorem proving. We believe such flexibility is a key component of developing general-purpose algorithms that can be specialized for different types of hardware units, such as communication protocols, floating-point units, and load store units.

- We have also explored the integration of fully-automated and interactive theorem proving techniques through the development of the ACL2SIX hint, which integrates IBM's SixthSense model checker with the ACL2 theorem prover. The ACL2SIX hint is unique in that not only does it decrease the amount of human guidance required in hardware verification, but it also avoids the need to create and maintain a semantics for VHDL in the ACL2 logic. The ACL2SIX hint has been applied to the formal verification of a high-performance multiplier circuit used in an industrial floating-point multiplier design. The multiplier circuit we verified is beyond the scope of what can be verified by SixthSense alone and the resulting ACL2 proof is much simpler than what would be required without SixthSense.

- Originally, both the SAT-based SULFA solver and the ACL2SIX hint required direct modifications to the ACL2 theorem prover. Discovering how to modify a sophisticated general-purpose interactive theorem prover in a sound manner is a lengthy process. A new mechanism has now been developed, however, for dynamically extending the theorem prover with new proof techniques, called clause processors. Clause processors may be either verified or unverified. Verified clause processors reduce the correctness of the proof technique to the correctness of the underlying ACL2 theorem prover. Unverified clause processors must be declared by their users to be trusted and

no theorem proven by an unverified clause processor will effect the theorem prover of a user that has not declared it. The new unverified clause processor mechanism supports the dynamic extension of ACL2 with either (or both) the SAT-based SULFA solver or the ACL2SIX hint.

- Another method to simplify the formal verification of hardware designs is to develop hardware description languages that are more amenable to formal verification. The DE2 hardware description language has thus been developed with hardware verification in mind. Its novel annotations feature provides the ability to incorporate specification into code, including specifications of non-functional behavior, e.g., power specifications, extended type signatures, and layout information. DE2 also has a simple semantics, making the task of formal verification of DE2 designs easier. The semantics of DE2 are also written in the logic of the ACL2 theorem prover, which provides access to a highly-automated, scalable hardware verification tool.

The main focus of the above contributions has been to decrease the amount of user guidance required by large-scale hardware verification. Our contributions have been divided along a broad front: from developing new hardware description languages to increasing the automation and flexibility of formal verification techniques and developing expertise with those techniques on complex processor designs.

Furthermore, our work points to further possible improvements.

- The DE2 language could be applied to the verification of non-functional design features, such as power, and the verification of programs that operate on circuits, such as timing and power optimization programs.

- The subclass of ACL2 formulas can be expanded to include properties involving arithmetic and constrained functions, as well as some properties that currently require induction.

- The performance of the SAT-based SULFA solver can be improved with improved SAT solvers, improved conversion techniques, and support for abstraction with automated refinement.
- A tighter integration between SixthSense and ACL2 can be created. A definition extension principle can be created to admit functions that are unrollable into bit-vector primitives. Also, multiple clock cycle variables can be supported to admit more sophisticated linear temporal logic properties.

By continuing to improve formal verification techniques and developing expertise over a larger set of hardware designs, we believe that formal verification will become cost-effective over an ever increasing set of hardware designs, eventually leading to the formal verification of entire large-scale industrial processors.

Bibliography

[1] M. Aagaard, R. B. Jones, R. Kaivola, K. R. Kohatsu, and C.-J. H. Seger. Formal Verification of Iterative Algorithms in Microprocessors. In *Proceedings of the* 37th *conference on Design automation (DAC 2000)*, pages 201–206, New York, NY, USA, 2000. ACM.

[2] M. D. Aagaard, R. B. Jones, and C.-J. H. Seger. Lifted-FL: A Pragmatic Implementation of Combined Model Checking and Theorem Proving. In *Proceedings of the* 12th *International Conference on Theorem Proving in Higher Order Logics (TPHOLs 1999)*, volume 1690 of *Lecture Notes in Computer Science*, pages 323–340, London, UK, 1999. Springer-Verlag.

[3] E. Axelsson, K. Claessen, and M. Sheeran. Wired: Wire-aware Circuit Design . In D. Borrione and W. J. Paul, editors, *Proceedings of the* 13th *Advanced Research Working Conference on Correct Hardware Design and Verification Methods (CHARME 2005)*, volume 3725 of *Lecture Notes in Computer Science*, pages 5–19. Springer, 2005.

[4] 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, Feb. 2000.

[5] S. Berezin. *Model Checking and Theorem Proving: A Unified Framework*. PhD thesis, Carnegie Mellon University, 2002.

[6] S. Beyer, C. Jacobi, D. Kröning, D. Leinenbach, and W. J. Paul. Putting it all together — Formal Verification of the VAMP. *International Journal on Software Tools for Technology Transfer (STTT)*, 8(4):411–430, 2006.

[7] R. S. Boyer and J S. Moore. Integrating Decision Procedures into Heuristic Theorem Provers. Technical Report ICSCA-CMP-44, University of Texas at Austin, 1985.

[8] B. Brock and W. A. Hunt, Jr. The Dual-Eval Hardware Description Language. *Formal Methods in Systems Design*, 11(1):71–104, 1997.

[9] D. Burger, S. W. Keckler, K. S. M. 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.

[10] E. M. Clarke and E. A. Emerson. Design and synthesis of synchronization skeletons using branching-time temporal logic. In D. Kozen, editor, *Logics of Programs, Workshop, May 1981*, volume 131 of *Lecture Notes in Computer Science*, pages 52–71, London, UK, 1982. Springer-Verlag.

[11] M. Davis, G. Logemann, and D. Loveland. A Machine Program for Theorem-Proving. *Communications of the ACM*, 5(7):394–397, 1962.

[12] L. M. de Moura, S. Owre, H. Rueß, J. M. Rushby, and N. Shankar. The ICS Decision Procedures for Embedded Deduction. In D. A. Basin and M. Rusinowitch, editors, *Proceedings of the Second International Joint Conference on Automated Reasoning (IJCAR 2004)*, volume 3097 of *Lecture Notes in Computer Science*, pages 218–222. Springer, 2004.

[13] 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 *Lecture Notes in Computer Science*, pages 78–92, Berlin, Germany, 2000. Springer-Verlag.

[14] D. L. Dill and J. Rushby. Acceptance of Formal Methods: Lessons from Hardware Design. *IEEE Computer*, 29(4):23–24, Apr. 1996.

[15] 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, Dec. 1991.

[16] A. Flatau, M. Kaufmann, D. F. Reed, D. Russinoff, E. W. Smith, and R. Sumners. Formal Verification of Microprocessors at AMD. In M. Sheeran and T. F. Melham, editors, *Proceedings of the* 4th *International Workshop on Designing Correct Circuits (DCC 2002)*, Apr. 2002.

[17] V. Ganesh and D. L. Dill. A Decision Procedure for Bit-Vectors and Arrays. In W. Damm and H. Hermanns, editors, *Proceedings of the* 19th *International Conference on Computer Aided Verification (CAV 2007)*, volume 4590 of *Lecture Notes in Computer Science*, pages 519–531. Springer, 2007.

[18] M. J. C. Gordon. Programming Combinations of Deduction and BDD-based Symbolic Calculation. *LMS Journal of Computation and Mathematics*, 5:56–76, 2002.

[19] M. J. C. Gordon, W. A. Hunt, Jr., M. Kaufmann, and J. Reynolds. An Embedding of the ACL2 Logic in HOL. In P. Manolios and M. Wilding, editors, *Proceedings of the* 6th *International Workshop on the ACL2 Theorem Prover and Its Applications (ACL2 2006)*, Aug. 2006.

[20] M. J. C. Gordon and T. F. Melham. *Introduction to HOL: A Theorem-Proving Environment for Higher-Order Logic*. Cambridge University Press, New York, NY, USA, 1993.

[21] D. A. Greve, R. Richards, and M. Wilding. A Summary of Intrinsic Partitioning Verification. In M. Kaufmann and J S. Moore, editors, *Proceedings of the* 5th *International Workshop on the ACL2 Theorem Prover and Its Applications (ACL2 2004)*, Nov. 2004.

[22] E. L. Gunter. Adding External Decision Procedures to HOL90 Securely. In J. Grundy and M. C. Newey, editors, *Proceedings of the* 11th *International Conference on Theorem Proving in Higher Order Logics (TPHOLs 1998)*, volume 1479 of *Lecture Notes in Computer Science*, pages 143–152, London, UK, 1998. Springer-Verlag.

[23] J. D. Guttman. A Proposed Interface Logic for Verification Environments. Technical Report M-91-19, The Mitre Corporation, Mar. 1991.

[24] J. Harrison. Metatheory and Reflection in Theorem Proving: A Survey and Critique. Technical Report CRC-053, SRI International Cambridge Computer Science Research Center, 1995.

[25] J. Harrison. The HOL Light Manual Version 1.1. Technical report, University of Cambridge Computer Laboratory, 2000.

[26] HOL4: The Latest Version of the HOL Automated Proof System for Higher Order Logic. See URL: `http://hol.sourceforge.net/`.

[27] W. A. Hunt, Jr. *FM8501: A Verified Microprocessor*. PhD thesis, Department of Computer Sciences, The University of Texas at Austin, 1985. Also published as Volume 795 of *Lecture Notes in Computer Science*, Springer, 1994.

[28] W. A. Hunt, Jr. The DE Language. In P. Manlolios, M. Kaufmann, and J S. Moore, editors, *Computer-Aided Reasoning: ACL2 Case Studies*, pages 119–131, Boston, MA, USA, June 2000. Kluwer Academic Publishers.

[29] 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 35–47, Upper Saddle River, NJ, USA, 1992. Prentice-Hall.

[30] W. A. Hunt, Jr., M. Kaufmann, R. Krug, J S. Moore, and E. Smith. Meta Reasoning in ACL2. In J. Hurd and T. F. Melham, editors, *Proceedings of the* 18th *International Conference on Theorem Proving in Higher Order Logics (TPHOLs 2005)*, volume 3603 of *Lecture Notes in Computer Science*, pages 163–178. Springer, 2005.

[31] W. A. Hunt, Jr. and E. Reeber. Formalization of the DE2 Language. In D. Borrione and W. J. Paul, editors, *Proceedings of the* 13th *Advanced Research Working Conference on Correct Hardware Design and Verification Methods (CHARME 2005)*, volume 3725 of *Lecture Notes in Computer Science*, pages 20–34. Springer, 2005.

[32] W. A. Hunt, Jr. and E. Reeber. A SAT-Based Procedure for Verifying Finite State Machines in ACL2. In P. Manolios and M. Wilding, editors, *Proceedings of the Sixth International Workshop on the ACL2 Theorem Prover and Its Applications (ACL2 2006)*, pages 127–135, New York, NY, USA, 2006. ACM.

[33] J. Hurd. An LCF-Style Interface between HOL and First-Order Logic. In A. Voronkov, editor, *Proceedings of the* 18th *International Conference on Automated Deduction (CADE 2002)*, volume 2392 of *Lecture Notes in Computer Science*, pages 134–138, London, UK, 2002. Springer-Verlag.

[34] J. Hurd. Fast Normalization in the HOL Theorem Prover. In T. Walsh, editor, *Proceedings of the Ninth Workshop on Automated Reasoning: Bridging the Gap between Theory and Practice*, Imperial College, London, UK, Apr. 2002. The Society for the Study of Artificial Intelligence and Simulation of Behaviour. An extended abstract.

[35] C. Jacobi. Formal Verification of Complex Out-of-Order Pipelines by Combining Model-Checking and Theorem-Proving. In E. Brinksma and K. G. Larsen, editors, *Proceedings of the* 14th *International Conference on Computer Aided Verification*

(CAV 2002), volume 2404 of *Lecture Notes in Computer Science*, pages 211–226, London, UK, 2002. Springer-Verlag.

[36] J. J. Joyce and C.-J. H. Seger. The HOL-Voss System: Model-Checking inside a General-Purpose Theorem-Prover. In T. F. Melham and J. Camilleri, editors, *Proceedings of the 7th International Workshop on Higher Order Logic Theorem Proving and Its Applications (TPHOLs 1994)*, volume 859 of *Lecture Notes in Computer Science*, pages 185–198, London, UK, 1994. Springer-Verlag.

[37] M. Kaufmann and J S. Moore. *ACL2: A Computational Logic for Applicative Common Lisp*. See URL: `http://www.cs.utexas.edu/users/moore/acl2/acl2-doc.html`.

[38] 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.

[39] M. Kaufmann and J S. Moore. Structured Theory Development for a Mechanized Logic. *Journal of Automated Reasoning*, 26(2):161–203, 2001.

[40] M. Kaufmann, J S. Moore, S. Ray, and E. Reeber. Integrating External Deduction Tools with ACL2. *Journal of Applied Logic*, to be published in 2008.

[41] M. Kaufmann, P. Manolios, and J S. Moore. *Computer-Aided Reasoning: An Approach*. Kluwer Academic Publishers, Norwell, MA, USA, 2000.

[42] 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 6th International Workshop on Implementation of Logics (IWIL 2006)*, volume 212 of *CEUR Workshop Proceedings*, pages 7–26, Nov. 2006.

[43] P. Manolios. *Mechanical Verification of Reactive Systems*. PhD thesis, Department of Computer Sciences, The University of Texas at Austin, 2001.

[44] P. Manolios and S. Srinivasan. A Framework for Verifying Bit-Level Pipelined Machines Based on Automated Deduction and Decision Procedures. *Journal of Automated Reasoning*, 37(1-2):93–116, Aug. 2006.

[45] C. McBride. *Dependentrly Typed Functional Programs and their Proofs*. PhD thesis, University of Edinburgh, 1999.

[46] W. McCune and O. Shumsky. Ivy: A Preprocessor and Proof Checker for First-Order Logic. In P. Manolios, M. Kaufmann, and J S. Moore, editors, *Computer-Aided Reasoning: ACL2 Case Studies*, pages 217–230. Kluwer Academic Publishers, Boston, MA, USA, June 2000.

[47] K. McMillan. *Symbolic Model Checking*. Kluwer Academic Publishers, 1993.

[48] K. L. 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* 10th *International Conference on Computer Aided Verification (CAV 1998)*, volume 1427 of *Lecture Notes in Computer Science*, pages 110–121, London, UK, 1998. Springer-Verlag.

[49] K. L. McMillan. A Methodology for Hardware Verification Using Compositional Model Checking. *Science of Computer Programming*, 37(1-3):279–309, 2000.

[50] K. L. McMillan. Parameterized Verification of the FLASH Cache Coherence Protocol by Compositional Model Checking. In T. Margaria and T. F. Melham, editors, *Proceedings of the* 11th *Advanced Research Working Conference on Correct Hardware Design and Verification Methods (CHARME 2001)*, volume 2144 of *Lecture Notes in Computer Science*, pages 179–195, London, UK, 2001. Springer-Verlag.

[51] D. Miller. A Logic Programming Language with Lambda-Abstraction, Function Variables, and Simple Unification. In *Proceedings of the International Workshop on Exten-*

sions of Logic Programming, pages 253–281, New York, NY, USA, 1991. Springer-Verlag New York, Inc.

[52] 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 *Lecture Notes in Computer Science*, pages 217–233, London, UK, 2004. Springer.

[53] J S. Moore. Introduction to the OBDD Algorithm for the ATP Community. *Journal of Automated Reasoning*, 12(1):33–46, 1994.

[54] 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, Sept. 1998.

[55] M. W. Moskewicz, C. F. 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, New York, NY, USA, 2001. ACM.

[56] 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 *Lecture Notes in Computer Science*, pages 1–16, Aarhus, Denmark, May 1995. Springer-Verlag.

[57] G. Nelson and D. C. Oppen. Simplification by Cooperating Decision Procedures. *ACM Transactions on Programming Languages and Systems*, 1(2):245–257, Oct. 1979.

[58] G. Nelson and D. C. Oppen. Fast Decision Procedures Based on Congruence Closure. *Journal of the ACM*, 27(2):356–364, 1980.

[59] T. Nipkow, L. Paulson, and M. Wenzel. *Isabelle/HOL: A Proof Assistant for Higher Order Logics*, volume 2283 of *Lecture Notes in Computer Science*. Springer-Verlag, London, UK, 2002.

[60] J. O'Leary, X. Zhao, R. Gerth, and C.-J. H. Seger. Formally Verifying IEEE Compliance of Floating-Point Hardware. *Intel Technology Journal*, (Q1):147–190, Feb. 1999.

[61] S. Owre, S. Rajan, J. M. Rushby, N. Shankar, and M. K. Srivas. PVS: Combining Specification, Proof Checking, and Model Checking. In R. Alur and T. A. Henzinger, editors, *Proceedings of the* 8th *International Conference on Computer Aided Verification (CAV 1996)*, volume 1102 of *Lecture Notes in Computer Science*, pages 411–414, London, UK, 1996. Springer-Verlag.

[62] S. Owre, J. M. Rushby, and N. Shankar. PVS: A Prototype Verification System. In D. Kapur, editor, *Proceedings of the* 11th *International Conference on Automated Deduction (CADE 1992)*, volume 607 of *Lecture Notes in Artificial Intelligence*, pages 748–752, London, UK, June 1992. Springer-Verlag.

[63] D. Patterson and J. Hennessy. *Computer Organization and Design: The Hardware/Software Interface, Second Edition*. Morgan Kaufmann Publishers, San Francisco, CA, USA, 1998.

[64] V. K. Pisini, S. Tahar, P. Curzon, O. Ait-Mohamed, and X. Song. Formal Hardware Verification by Integrating HOL and MDG. In *Proceedings of the* 10th *Great Lakes symposium on VLSI (GLSVLSI 2000)*, pages 23–28, New York, NY, USA, 2000.

[65] J.-P. Queille and J. Sifakis. Specification and Verification of Concurrent Systems in CESAR. In M. Dezani-Ciancaglini and U. Montanari, editors, *Proceedings of the* 5th *International Symposium on Programming*, volume 137 of *Lecture Notes in Computer Science*, pages 337–351, London, UK, 1982. Springer-Verlag.

[66] R. Boulton, A. Gordon, M.J.C. Gordon, J. Herbert, and J. van Tassel. Experience with Embedding Hardware Description Languages in HOL. In V. Stavridou, T. F. Melham, and R. T. Boute, editors, *Proceedings of the International Conference on Theorem Provers in Circuit Design: Theory, Practice and Experience (TPCD 1992)*, volume A-10 of *IFIP Transactions*, pages 129–156, Nijmegen, The Netherlands, 1992. North-Holland.

[67] 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`.

[68] S. Ray. *Using Theorem Proving and Algorithmic Decision Procedures for Large-Scale System Verification*. PhD thesis, Department of Computer Sciences, The University of Texas at Austin, 2005.

[69] 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, *Proceedings of the* 4th *International Workshop on the ACL2 Theorem Prover and Its Applications (ACL2 2003)*, 2003.

[70] 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 Third International Joint Conference on Automated Reasoning (IJCAR 2006)*, volume 4130 of *Lecture Notes in Computer Science*, pages 453–467. Springer, 2006.

[71] E. Reeber and J. Sawada. Combining ACL2 and an Automated Verification Tool to Verify a Multiplier. In P. Manolios and M. Wilding, editors, *Proceedings of the Sixth International Workshop on the ACL2 Theorem Prover and Its Applications (ACL2 2006)*, pages 63–70, New York, NY, USA, 2006. ACM.

[72] D. M. Russinoff. A mechanically checked proof of IEEE compliance of the floating point multiplication, division and square root algorithms of the AMD-K7 processor. *LMS Journal of Computation and Mathematics*, 1:148–200, 1998. Appendices A and B available to subscribers electronically (http://www.lms.ac.uk/jcm/1/lms98001/appendix-a/ and http://www.lms.ac.uk/jcm/1/lms98001/appendix-b/).

[73] K. Sankaralingam, R. Nagarajan, R. McDonald, R. Desikan, S. Drolia, M. Govindan, P. Gratz, D. Gulati, H. Hanson, C. Kim, H. Liu, N. Ranganathan, S. Sethumadhan, S. Sharif, P. Shivakumar, S. W. Keckler, and D. Burger. Distributed microarchitectural protocols in the TRIPS prototype processor. In *Proceedings of the* 39th *Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-39 2006)*, pages 480–491, Washington, DC, USA, Dec. 2006. IEEE Computer Society.

[74] Satisfiability Suggested Format. see URL: ftp://dimacs.rutgers.edu/pub/challenge/satisfiability/doc/satformat.tex, 1993.

[75] J. Sawada. *Formal Verification of an Advanced Pipelined Machine*. PhD thesis, Department of Computer Sciences, The University of Texas at Austin, 1999.

[76] J. Sawada. ACL2VHDL Translator: A Simple Approach to Fill the Semantic Gap. In M. Kaufmann and J S. Moore, editors, *Proceedings of the* 5th *International Workshop on the ACL2 Theorem Prover and Its Applications (ACL2 2004)*, Nov. 2004.

[77] J. Sawada and E. Reeber. ACL2SIX: A Hint used to Integrate a Theorem Prover and an Automated Verification Tool. In *Proceedings of the* 6th *International Conference on Formal Methods in Computer-Aided Design (FMCAD 2006)*, pages 161–170, Los Alamitos, CA, 2006. IEEE Computer Society.

[78] C.-J. H. Seger, R. B. Jones, J. W. O'Leary, T. F. Melham, M. Aagaard, C. Barrett, and D. Syme. An Industrially Effective Environment for Formal Hardware Verification.

IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, 24(9):1381–1405, Sept. 2005.

[79] S. Sethumadhavan. *Scalable Memory Disambiguation*. PhD thesis, Department of Computer Sceinces, The University of Texas at Austin, 2007.

[80] S. Sethumadhavan, R. McDonald, R. Desikan, D. Burger, and S. W. Keckler. Design and Implementation of the TRIPS Primary Memory System. In *Proceedings of the* 24th *Annual IEEE International Conference on Computer Design (ICCD 2006)*, pages 470–476. IEEE, 2006.

[81] N. Shankar. Using Decision Procedures with Higher Order Logics. In R. J. Boulton and P. B. Jackson, editors, *Proceedings of the* 14th *International Conference on Theorem Proving in Higher Order Logics (TPHOLs 2001)*, volume 2152 of *Lecture Notes in Computer Science*, pages 5–26, London, UK, 2001. Springer-Verlag.

[82] R. E. Shostak. Deciding Combinations of Theories. *Journal of the ACM*, 31(1):1–12, 1984.

[83] K. Slind. *Reasoning about Terminating Functional Programs*. PhD thesis, Technical University of Munich, 1999.

[84] SMT-COMP. See URL: `http://www.csl.sri.com/users/demoura/smt-comp/`.

[85] A. Stump, C. W. Barrett, and D. L. Dill. CVC: A Cooperating Validity Checker. In E. Brinksma and K. G. Larsen, editors, *Proceedings of the* 14th *International Conference on Computer Aided Verification (CAV 2002)*, volume 2404 of *Lecture Notes in Computer Science*, pages 500–504, London, UK, 2002. Springer-Verlag.

[86] The International SAT Competition. See URL: `http://www.satcompetition.org/`.

[87] The Minisat SAT Solver. See URL:
`http://www.cs.chalmers.se/Cs/Research/FormalMethods/MiniSat/`.

[88] The Zchaff SAT Solver. See URL:
`http://www.princeton.edu/~chaff/zchaff.html`.

[89] G. S. Tseitin. On the Complexity of Derivation in the Propositional Calculus. *Zapiski nauchnykh seminarov LOMI*, 8:234–259, 1968. English translation of this volume: Consultants Bureau, N.Y., 1970, pp. 115–125.

[90] M. N. Velev and R. E. Bryant. Effective Use of Boolean Satisfiability Procedures in the Formal Verification of Superscalar and VLIW. In *Proceedings of the* 38th *Conference on Design Automation Conference (DAC 2001)*, pages 226–231, New York, NY, USA, June 2001. ACM.

[91] T. Weber. Integrating a SAT Solver with an LCF-style Theorem Prover. In A. Armando and A. Cimatti, editors, *Proceedings of the Third International Workshop on Pragmatical Aspects of Decision Procedures in Automated Reasoning (PDPAR 2005)*, Edinburgh, UK, July 2005.

[92] Yices: An SMT Solver. See URL: `http://yices.csl.sri.com/`.

[93] L. Zhang and S. Malik. Validating SAT Solvers Using an Independent Resolution-Based Checker: Practical Implementations and Other Applications. In N. Wehn and D. Verkest, editors, *Proceedings of the Conference on Design, Automation and Test in Europe (DATE 2003)*, pages 10880–10885, Washington, DC, USA, 2003. IEEE Computer Society.

Vita

Erik Henry Reeber was born in Santa Cruz, California in 1978 to Henry and Karen Reeber. He graduated from Aptos High School in 1996. Erik went on to major in Electrical Engineering and Computer Sciences at the University of California at Berkeley, where he received a Bachelor of Science in May 2000. Erik joined the University of Texas at Austin in August of 2000, and received a Master's of Science in Computer Sciences in December 2002, before continuing in the Ph.D. program. In May 2008, Erik will marry Carrie Pankrast.

Permanent Address: 25 Crest Lane
Watsonville, CA 95076

This dissertation was typeset with LaTeX 2_ε[1] by the author.

[1] LaTeX 2_ε is an extension of LaTeX. LaTeX is a collection of macros for TeX. TeX is a trademark of the American Mathematical Society. The macros used in formatting this dissertation were written by Dinesh Das, Department of Computer Sciences, The University of Texas at Austin, and extended by Bert Kay and James A. Bednar.