The Dissertation Committee for Sol Otis Swords
certifies that this is the approved version of the following dissertation:

A Verified Framework for Symbolic Execution in the ACL2 Theorem Prover

Committee:

Warren A. Hunt, Jr., Supervisor

Jason Baumgartner

Robert S. Boyer

William Cook

J Strother Moore

A Verified Framework for Symbolic Execution in the ACL2 Theorem Prover

by

Sol Otis Swords, B.A.

DISSERTATION

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

DOCTOR OF PHILOSOPHY

THE UNIVERSITY OF TEXAS AT AUSTIN

December 2010

Acknowledgments

I would like to thank my committee members, Warren Hunt, Jason Baumgartner, Bob Boyer, William Cook, and J Moore, for all of their encouragement, patience, and advice. My advisor, Warren Hunt, and Bob Boyer were particularly helpful in getting me started on this project.

I am grateful to Jared Davis, a colleage at both the University of Texas and Centaur Technology, for his exceptional help in both planning my the project and editing this thesis. I would also like to thank Matt Kaufmann for adding several ACL2 features that made this project possible, and Anna Slobodova for her insights as the main user of the product of this research.

I'd like to thank everyone I have worked with at both UT and Centaur, including David Rager, Sandip Ray, Serita Nelesen, Erik Reeber, Robert Krug, Mark Reitblatt, Terry Parks, Al Sato, Alan Dunn, and Dan Luu, for helping me learn ACL2 and providing interesting problems to work on.

Thanks to my parents, Peter and Diane Swords, and my sister Alicia Swords, for their confidence and support, and to Sarah Swords and Franklin, Miriam, and Sam Kalk for being my family here in Austin. Thanks to my friends and my kickball team for keeping me a sane and happy graduate student. Finally, thanks to my girlfriend Eileen McGinnis for all her wonderful moral, emotional, linguistic, and culinary support.

A Verified Framework for Symbolic Execution in the ACL2 Theorem Prover

Publication No. _____

Sol Otis Swords, Ph.D.
The University of Texas at Austin, 2010

Supervisor: Warren A. Hunt, Jr.

Mechanized theorem proving is a promising means of formally establishing facts about complex systems. However, in applying theorem proving methodologies to industrial-scale hardware and software systems, a large amount of user interaction is required in order to prove useful properties. In practice, the human user tasked with such a verification must gain a deep understanding of the system to be verified, and prove numerous lemmas in order to allow the theorem proving program to approach a proof of the desired fact. Furthermore, proofs that fail during this process are a source of confusion: the proof may either fail because the conjecture was false, or because the prover required more help from the user in order to reach the desired conclusion.

We have implemented a symbolic execution framework inside the ACL2 theorem prover in order to help address these issues on certain problem do-

mains. Our framework introduces a proof strategy that applies bit-level symbolic execution using BDDs to finite-domain problems. This proof strategy is a fully verified decision procedure for such problems, and on many useful problem domains its capacity vastly exceeds that of exhaustive testing. Our framework also produces counterexamples for conjectures that it determines to be false.

Our framework seeks to reduce the amount of necessary user interaction in proving theorems about industrial-scale hardware and software systems. By increasing the automation available in the prover, we allow the user to complete useful proofs while understanding less of the detailed implementation of the system. Furthermore, by producing counterexamples for falsified conjectures, our framework reduces the time spent by the user in trying to determine why a proof failed.

Table of Contents

Chapter 1

Introduction

When using theorem proving to establish correctness properties of industrial scale hardware and software systems, it is necessary to prove numerous lemmas on the way to the desired top-level proof. This process requires the human verifier to obtain a detailed understanding of the program or design being verified, which is time consuming and expensive. Maintenance of a completed proof is also expensive: the proof script must be adjusted over time to accomodate any changes in the implementation of the system under verification, and it is usually necessary for the user to understand these changes in order to make the adjustments. Furthermore, during the process of composing and maintaining proofs, conventional theorem provers are of limited use in debugging failed proofs. A failure may occur either because the goal is not a theorem or because additional lemmas are needed; thus, the user may waste time proving additional lemmas in an attempt to fix the failed proof before realizing that the original conjecture was false.

In order to address these problems, we have implemented a framework that allows symbolic execution of user-defined functions within the ACL2 theorem prover, using a BDD-based symbolic data representation. We have proved

sufficient correctness properties about our framework to allow us to use symbolic execution as a proof strategy. For theorems whose variables are restricted to a finite domain, our framework provides a decision procedure that is fully verified and has a capacity reaching a large set of problems for which exhaustive simulation is infeasible.

We believe that on domains in which it is applicable, our framework helps to alleviate some of the drawbacks of interactive theorem proving. By increasing the prover's capacity to complete proofs automatically, our framework reduces the level of detail required in the user's understanding of the artifact to be verified. Changes in the implementation that are not large enough to affect the statement of a theorem at this reduced level of detail usually will not require any changes in the proof script. Finally, when a user attempts to prove an untrue theorem, our framework provides useful feedback by finding and displaying counterexamples.

Our symbolic execution framework is called GL, short for "G in the Logic." It is a descendent of the G system of Boyer and Hunt [14]. The G system is also a framework for symbolic execution of ACL2 code. However, it is implemented in Common Lisp outside of the ACL2 logic, and as a result its symbolic executions cannot be shown by ACL2 to have any meaning in the ACL2 logic. Furthermore, during the course of its industrial use, several logical bugs affecting soundness were found in the G system, and it is not known whether others remain.

GL addresses both of these problems of the G system. GL is coded in the ACL2 logic and its symbolic executions are mechanically proven to accurately reflect the concrete semantics of ACL2. It also contains a proof procedure that is integrated with the ACL2 system so that it can be used directly to prove theorems. GL's proof procedure is verified in ACL2 by a meta-level proof. Theorems proven using GL therefore rely only on the soundness of ACL2 itself and the Hons system [12], which augments ACL2 with function memoization and unique object representation facilities; no other unverified add-ons are used in proving these theorems.

In the remainder of this introduction, we first offer a motivating example, describing uses of our framework for both semi-formal testing and theorem proving (Section 1.1). We show in this example how our framework can reduce the need for difficult, highly interactive, inductive proofs. We then describe the theoretical basis of the use of symbolic execution for proof (Section 1.2). In that section we give a mathematical definition of symbolic execution and prove a metatheorem that encapsulates our strategy for proof by symbolic execution.

In the rest of this dissertation, we first describe the implementation of symbolic execution in our framework (Chapter 2). Next, we build on the theory described in Section 1.2 to describe how we automate proofs using the symbolic execution core (Chapter 3); we discuss several features of our framework designed to facilitate such proofs. We then describe a hardware modeling framework which we have integrated with GL in order to allow symbolic

execution-based proofs about hardware models (Chapter 4), and illustrate the use of this hardware modeling technology in conjunction with GL in a case study describing a successful industrial application at Centaur Technology (Chapter 5). We review related work, then conclude by discussing possible future improvements to and potential new uses for our framework.

1.1 Motivating Example

We now work through a motivating example showing how GL can be used to prove a theorem that is quite difficult for a human user to prove using traditional theorem proving methods, and which takes significantly longer to prove using exhaustive simulation. In the example, we will study an integer square root algorithm. We will first demonstrate how GL may be used to run symbolic tests of the algorithm, illustrating the use of symbolic execution in our framework (Subsection 1.1.1). We then describe the process of proving the correctness of this algorithm using GL (Subsection 1.1.2).

Our example is a fixed-width integer square root algorithm. Its inputs are natural numbers n and w, where n should have bit width less than or equal to w, and the specification of the algorithm is simply that it should produce the greatest integer whose square is less than or equal to n. This algorithm, described by Crenshaw [27], is of a style that might be implemented in hardware: it avoids multiplications and divisions and its main loop contains only shifting, bit-masking, addition, and comparison operations. Pseudocode for the algorithm is listed in Algorithm 1; this listing uses the symbols $<$

Algorithm 1 Integer square root

```
1:  procedure INT-SQRT(n, w)
2:      k ← ⌈w/2⌉
3:      rem ← 0
4:      root ← 0
5:      for i from 1 to k do
6:          root ← root ≪ 1
7:          rem ← (rem ≪ 2) + (n ≫ (2k − 2))
8:          n ← (n ≪ 2) & ((1 ≪ 2k) − 1)
9:          root ← root + 1
10:         if root ≤ rem then
11:             rem ← rem − root
12:             root ← root + 1
13:         else
14:             root ← root − 1
15:     return root ≫ 1
```

<, ≫, and & for left-shift, right-shift, and bitwise-and, respectively. The corresponding ACL2 function definitions are given as Program 1. Although this algorithm is small, it is not straightforward to prove correct by traditional theorem proving methods. Even with a good understanding of the algorithm, it is challenging to discover the invariant necessary for a proof by induction. In most cases, during the course of proving a theorem like this one, the verifier will attempt to prove several incorrect invariants before finding the correct one. For each failed proof of an invariant, the verifier must examine the output of the failed proof in an attempt to diagnose the problem. A proof might fail because of a flaw in the proposed invariant, but also might fail because the prover's current set of arithmetic reasoning rules was not powerful enough to prove the invariant. Worse, sometimes there is a flaw in the invariant that

Program 1 INT-SQRT ACL2 definitions

```
(defun int-sqrt (n w)
  (let ((k (ceiling w 2)))
    (int-sqrt-loop k n 0 0 (* 2 k))))

(defun int-sqrt-loop (ctr n rem root width)
  (if (zp ctr)
      (>> root 1)
    (let* ((root (<< root 1))
           (rem (+ (<< rem 2) (>> n (- width 2))))
           (n (& (<< n 2) (- (<< 1 width) 1)))
           (root (+ 1 root)))
      (mv-let (root rem)
        (if (<= root rem)
            (mv (+ 1 root) (- rem root))
          (mv (- root 1) rem))
        (int-sqrt-loop (- ctr 1) n rem root width)))))
```

the prover's arithmetic reasoning is not yet strong enough to reveal. Thus a correctness proof for an algorithm such as this may consume many hours of effort even for an experienced user of a theorem prover with an advanced arithmetic library.

The GL framework may be used to verify this algorithm for a fixed bit-width that is sufficiently small. Empirically, the time and memory used by symbolic execution for this proof increases by a factor of about 1.5 for every additional bit. Since this growth is exponential, the scaling is limited. Nevertheless, symbolic execution is significantly faster than exhaustive simulation: at 32 bits, symbolic execution takes about 90 seconds, whereas exhaustive simulation takes about 70 minutes on the same machine. Many other examples

exhibit better scaling than this one; in Chapter 5, for example, we describe floating-point addition verifications in which these symbolic execution techniques scale up to addition of two 80-bit floating-point inputs. We chose the INT-SQRT example for this initial exposition because the statement of the algorithm and its correctness condition are very concise despite the difficulty of the inductive proof.

1.1.1 Testing with Symbolic Execution

GL's symbolic execution capabilities may be used to test the algorithm symbolically, effectively running many tests in a single symbolic execution. To perform such tests, we must provide input in the form of a *symbolic object*, an abstraction of program data that may represent many possible concrete values. A symbolic execution on such an object effectively tests the algorithm on each of the values represented by the input. The output of the symbolic execution will also be a symbolic object; this result object will represent the possible results of the algorithm on the values represented by the symbolic input. This result object may be examined to determine whether any unexpected values occur.

For example, one might test that for some fixed natural number k, INT-SQRT$(n, w) = k$ for all n from k^2 to $k^2 + 2k$, given some w chosen so that $n < 2^w$. Let us take $k = 50$ and $w = 32$. To perform this test, we may symbolically execute INT-SQRT$(n, 32)$ with n assigned a symbolic object whose possible values include all integers in the range from $50^2 = 2500$ to

$50^2 + 2 \cdot 50 = 2600$. The result of this computation will be a symbolic object whose possible values include all of the results of INT-SQRT$(n, 32)$ for n in this range.

GL's symbolic object format primarily uses a bit-level data representation, with BDDs [20] representing Boolean functions. GL symbolic objects consist of BDDs wrapped in tagged structures that describe how the individual bits represented by the BDDs are to be combined into data objects. These tagged structures support symbolic objects ranging over all types of data in the universe of ACL2 objects, with special support for Booleans, numbers, and conses. We describe the symbolic object format in Section 2.1. For this example we will use a symbolic object construct that ranges over a set of integers. This will allow us to construct an object representing n of the range described above.

The following object is a symbolic integer in the GL symbolic object format, where each of the b_i are BDDs:

(:NUM (b_0 b_1 ... b_m))

This symbolic object's possible values consist of some set of integers, depending on the characteristics of the BDDs $b_0, \ldots, b_m$. This object may be evaluated, yielding an integer value, by assigning a Boolean value to each variable in these BDDs. This assignment allows each BDD to evaluate to a Boolean value, and the value of the symbolic object is then the two's-complement integer composed of those bits, with the least significant bits on the left. For example, if b_0 takes

the value *true* and all the other BDDs take the value *false*, then this object's value is 1; if b_m is *true* and all others are *false*, then this object's value is -2^m, negative because of the two's-complement encoding.

To represent the desired range $2500 \leq n \leq 2600$ in this format, we need to construct a symbolic object whose evaluations yield the two's-complement bits of integers in that range and no other values. To do this, we must construct a list of BDDs $b_0, \ldots, b_m$ that always evaluate to a list of Boolean values representing a two's-complement number in that range, and which may evaluate to any number in that range. Constructing such a list of BDDs whose values cover exactly some particular set of values is, in general, quite difficult to do manually, since there may be many interdependencies between the individual bits. GL instead offers an automated process that uses BDD parametrization [5] to reduce a symbolic object that covers a larger set of values than desired to one that covers exactly the desired set. (We describe this process in Section 3.1.) Usually it is easy to construct a symbolic object that covers a superset of the desired values; parametrization will reduce this object to one that covers only the values needed.

The user interface for our automated parametrization process requires the user to provide a *shape specifier* instead of creating a symbolic object directly. A shape specifier is formatted similarly to a symbolic object, but replaces the BDDs contained within the object with indices indicating BDD variables. It therefore gives type and size information about the object as well as a BDD ordering of its bits, but does not allow complex interdependencies

between the BDDs. We discuss the reasons for preferring shape specifiers over symbolic objects in Section 3.2. One appropriate shape specifier for our example is the following object:

```
(:NUM (12 11 ... 1 0)).
```

This shape specifier describes an integer of 13 two's-complement bits, thus ranging from -4096 to 4095; thus, it covers our desired range and is very simple to construct.

We perform BDD parametrization and symbolic execution using the GL-INTERP program, which takes three inputs: the term to be symbolically executed, an association list of bindings giving the shape specifier for each of the free variables of the term, and optionally a hypothesis with which to restrict the inputs by BDD parametrization. For our test, we would call GL-INTERP as follows:

```
(gl-interp (int-sqrt n 32)
           '((n (:NUM (12 11 10 9 8 7 6 5 4 3 2 1 0))))
            :hyp (and (<= 2500 n) (<= n 2600)))
```

The symbolic result yielded by this computation is simply 50, meaning that this is the only possible result of INT-SQRT$(n, 32)$ for all n in the specified range. However, if we run this form with the range extended up to 2700, then the result is a `:NUM` object, reflecting the fact that there is more than one possible value of the function for inputs in this range. In fact, the particular

object returned has a non-constant BDD in its low-order bit, but every other bit is a constant `T` or `NIL`; this reflects the fact that the only two possible values are 50 and 51, which differ only in the least significant bit.

To perform this symbolic execution of the INT-SQRT function, GL-INTERP uses a *symbolic interpreter*. A symbolic interpreter takes a term object (in this case, `(int-sqrt n 32)`) as input along with a table associating each free variable of the term with a symbolic object, and it produces a symbolic result representing the value of that term. It functions by performing a McCarthy-style recursive interpretation of the term, in which it may descend into the bodies of defined functions as well as subterms. The symbolic interpretation algorithm is described in detail in Section 2.5. The symbolic interpreter must also be able to symbolically execute functions with no definitions, such as the ACL2 primitives `+` and `<`. For each one of these primitives, we define a *symbolic counterpart*, a function designed specifically to perform a symbolic execution of that primitive. We describe the definition of symbolic counterparts for primitives in Section 2.3 and automatic generation of symbolic counterparts for other functions in Section 2.4. The symbolic interpreter also handles **If** terms specially so as to avoid symbolic execution of unreachable control branches; this is described in Section 2.2.

1.1.2 Proof by Symbolic Execution

We now sketch the process of proving the correctness of this algorithm on 32-bit inputs using GL. First, we state the theorem we are interested in

proving:

Theorem (Correctness of INT-SQRT for 32-bit inputs)**.** *If n is an integer and $0 \leq n < 2^{32}$, let $root$ = INV-SQRT$(n, 32)$. Then:*

$$root^2 \leq n < (root + 1)^2.$$

To prove this theorem using GL, we separate its statement into hypotheses, which restrict the domain of the free (implicitly universally quantified) variables of the theorem, and conclusion. Here, our hypothesis is that n is a 32-bit natural number, and the conclusion is that INV-SQRT operates as specified on n. Then, we symbolically execute the conclusion using symbolic input data that represents all inputs that satisfy the hypotheses. The conclusion takes the form of a conjunction of inequalities, which always results in a Boolean value. This symbolic execution will therefore result in a symbolic Boolean value. This value represents all of the possible evaluations of the conclusion on the inputs in our domain of interest. If the symbolic result value is constant-true, then we may infer that the conclusion is always true on these inputs, i.e., the theorem holds. We expand on this argument, describing sufficient conditions for proof by symbolic execution, in Section 1.2

This theorem involves one free variable n, which is constrained by the hypotheses to be a 32-bit natural number. Therefore, we must assign to n a shape specifier that represents a superset of the 32-bit naturals. The following shape specifier represents a 33-bit signed integer, which is sufficient to cover

the 32-bit naturals:

```
(:NUM (32 31 ... 1 0)).
```

We have chosen in this case to associate the lowest-numbered variable indices with the most significant bits of this symbolic integer, because empirically this leads to better symbolic execution performance on this algorithm than the reverse order. It is often the case that a good choice of ordering for the BDD variables is crucial for symbolic execution performance.

We may use this binding to perform a simulation using GL-Interp as we did for the test case above, symbolically executing the conclusion with the hypothesis that $0 \leq n < 2^{32}$:

```
(gl-interp (let* ((root (int-sqrt n 32))
                  (sq (* root root)))
             (and (<= sq n)
                  (< n (+ sq (* 2 root) 1))))
           '((n (:NUM (32 31 ... 3 2 1 0))))
           :hyp (and (<= 0 n) (<= n (expt 2 32))))
```

From this symbolic execution we get the result `T`, indicating that this is the only possible value of the conclusion under the assumption of the hypothesis. This provides evidence that our conjecture is a theorem, but does not cause the formula to be accepted as a theorem in ACL2.

For admitting such claims as ACL2 theorems, GL provides a utility called Def-GL-Thm ("define a theorem using GL.") The command used to

Program 2 ACL2 event for INT-SQRT correctness theorem

```
(def-gl-thm 32-bit-int-sqrt-correct
  :hyp (and (integerp n)
            (<= 0 n)
            (< n (expt 2 32)))
  :concl (let* ((root (int-sqrt n 32))
                (sq (* root root)))
           (and (<= sq n)
                (< n (+ sq (* 2 root) 1))))
  :g-bindings `((n ,(g-number
                     '((32 31 30 29 28 27 26 25 24 23 22 21 20
                           19 18 17 16 15 14 13 12 11 10 9 8 7
                           6 5 4 3 2 1 0))))))
```

prove our INT-SQRT correctness theorem is listed as Program 2. This form macroexpands to a DEFTHM command, shown in Program 3, which instructs ACL2 to attempt to prove the conjecture and, if successful, enter it in its database of facts. This DEFTHM command also provides a hint to the prover instructing it to using the GL *clause processor*, a proof procedure that applies symbolic execution in order to show that the conclusion holds when the hypotheses are fulfilled. We discuss the GL clause processor in detail in Section 3.3.

When the DEF-GL-THM form is run, ACL2 processes the resulting DEFTHM event. Because of the hint given, ACL2 immediately calls the GL clause processor on the conjecture. The clause processor proceeds in steps similar to the ones used above to test the algorithm. First it transforms the user-provided shape specifier into a symbolic input object that covers only the values satisfying the hypothesis. Next it symbolically executes the conclusion

Program 3 Macroexpansion of INT-SQRT correctness theorem

```
(defthm 32-bit-int-sqrt-correct
  (implies (and (integerp n)
                (<= 0 n)
                (< n (expt 2 32)))
           (let* ((root (int-sqrt n 32))
                  (sq (* root root)))
             (and (<= sq n)
                  (< n (+ sq (* 2 root) 1)))))
  :hints
  ('(:computed-hint-replacement
     ( ... )
     :clause-processor
     (gl::glcp
      clause
      (list '((n (:NUM
                  (32 31 30 29 28
                      27 26 25 24 23 22 21 20 19 18 17 16 15
                      14 13 12 11 10 9 8 7 6 5 4 3 2 1 0))))
            ... )
      state)))
  :rule-classes :rewrite)
```

with this symbolic object as its input. Finally, it analyzes the symbolic result produced by the symbolic execution of the conclusion. If this result is constant-true, then the symbolic execution was successful. If not, then the clause processor uses this result to construct a counterexample to the claim and displays this counterexample to the user.

When the symbolic execution completes successfully, the clause processor produces one further proof obligation that must be discharged in order to admit the theorem. We must show that the shape specifiers that we provided

for the free variables of the theorem are sufficiently general to cover all assignments of these variables that satisfy the hypotheses. If the shape specifier that we provided for n only represented 10-bit integers, for example, then a successful symbolic execution would not imply anything about inputs greater than $2^9 - 1 = 511$. Our successful symbolic execution only implies the correctness of our theorem if every input satisfying the hypothesis is covered by the shape specifier provided. DEF-GL-THM gives hints to the prover that aid in completing this coverage proof; in this case, these hints are sufficient to complete the proof without user intervention. We discuss the logical necessity for this coverage obligation in Section 1.2 and our proof strategy for this coverage obligation in Section 3.2.

This proof completes in under 90 seconds on an Intel Xeon® E5450 CPU. While the correctness proof of this algorithm does not scale much beyond 32 bits, we have provided this example to show that GL is capable of quickly performing proofs that would be quite time consuming to carry out using conventional theorem proving methods such as rewriting and induction. In Chapter 5 we will discuss an industrial application in which GL is used for several proofs that would have been extremely difficult and time consuming to accomplish using conventional theorem proving, and also far beyond the capacity of exhaustive testing.

1.2 Theory

The GL symbolic execution framework adds to the ACL2 logic a reasoning tool for proving conjectures whose variables range over finite sets of objects. From a logical standpoint, our new reasoning procedure is, in most cases, equivalent to proof by exhaustive simulation: when the hypotheses of a theorem restrict the free variables so that they each range over a finite set, in principle the theorem can be proven simply by evaluating the conclusion for every possible setting of the variables. However, our tool can succeed in many cases where the finite set of inputs is far too large for exhaustive simulation to be practical.

The reasoning engine used in this proof procedure is symbolic execution. Intuitively, to perform a symbolic execution of a program is to perform analogous operations on symbolic data objects to the operations the program itself would perform on concrete data during an ordinary execution. These symbolic data objects may represent many possible concrete data objects, and therefore the result of a symbolic execution may reveal properties common to many concrete executions. Symbolic execution may be used as a reasoning tool by inferring properties of the program under study from properties of the result of the symbolic execution.

Much of the practical usefulness of symbolic execution as a reasoning tool depends on the particular representation of symbolic objects. However, since in this section we are concerned with the theory of symbolic execution rather than its practicality, we do not yet specify a format for symbolic ob-

jects. In this section, we only require that symbolic objects are data objects which may be evaluated, yielding concrete values. They are thus a data representation of functions on some domain. A symbolic execution of a program on some symbolic objects yields a new symbolic object which represents the function composition of this program with the functions represented by the symbolic inputs.

For example, consider this symbolic object from the previous section:

$$\texttt{(:NUM (}b_0\ b_1\ \ldots\ b_n\texttt{))}$$

This object may be evaluated by assigning Boolean values to the variables referenced in the BDDs $b_0, \ldots, b_n$. Under this assignment, each BDD takes a Boolean value and these values are then interpreted as the binary digits of a two's-complement integer. Therefore, this object represents some function whose domain is the set of possible assignments of Boolean values to BDD variables, and whose range is some subset of the integer values between -2^n and 2^n-1. The symbolic execution of the INT-SQRT operation on this symbolic input object yields a new symbolic object which represents the composition of the INT-SQRT function with the function represented by the input object.

We sometimes describe a symbolic object by its *coverage set*, the range of the function it represents. We say a symbolic object *covers* a set if that set is a subset of its coverage set. Note that in the example above, the coverage set depends on the particular BDDs b_i. If each of these BDDs is an independent variable, then the coverage set is the entire integer range from -2^n to $2^n - 1$.

On the other hand, if each is a constant, then the coverage set contains only a single value.

A symbolic object's coverage set is not a complete description of that object. For example, consider the symbolic objects `(:NUM (`v`))` and `(:NUM (`$\neg v$`))`. Both of these objects cover the set $\{0, -1\}$. If we considered only the coverage sets of these objects and imagined including them into an ordered pair, we might expect the coverage of that pair to be

$$\{(0,0),(0,-1),(-1,0),(-1,-1)\},$$

the Cartesian product of their coverage sets. However, because they share a variable in their representations, their pairing would in fact only cover the set $\{(0,-1),(-1,0)\}$. Because of variable interdependencies in the representations of symbolic objects, it is important to think of symbolic objects as representing functions ranging over concrete objects, rather than simply sets of concrete objects.

The set of concrete executions represented by a symbolic execution is determined by the coverage set of the symbolic input vector, and the set of possible values produced by those concrete executions is represented by the coverage set of the symbolic result. This observation can be used to prove conjectures by symbolic execution. Suppose the conjecture has hypotheses which constrain the values of the free variables to some finite set. One might choose an assignment of symbolic inputs to the variables such that the coverage set of these inputs in aggregate covers the set of values allowed by the hypotheses.

A symbolic execution of the conclusion on these same inputs then yields a symbolic object covering the set of possible values of the conclusion on the allowed inputs. In particular, if the symbolic result may only take the value *true*, then the theorem is proven. Otherwise, often the result may be examined to produce specific counterexamples. Our symbolic execution tool automates this proof procedure and counterexample generation for failed proofs.

We now more rigorously describe the logical meaning of symbolic execution, and show that it can be used to prove theorems as described above and as illustrated by example in the previous section. We will first define some terms and then describe a method for performing a proof by symbolic execution. We also prove a metatheorem stating the soundness of this method of proof.

If s is a symbolic object, we write $\langle s \rangle$ for the function represented by that object.[1] We call the inputs to such functions *environments*; for any environment e, $\langle s \rangle(e)$ is the concrete value of s under environment e. The coverage set of s is the range of $\langle s \rangle$.

In the examples of symbolic objects and evaluations we have described so far, the environments have been sequences of Booleans providing values for each of the BDD variables, and the coverage sets have been subsets of the in-

[1] We use similar notation to denote the Boolean function represented by a BDD b, namely $\langle b \rangle_{bdd}$. Because ACL2 is a first-order logic, there can be no operator $\langle\rangle$ that takes a data object as input and produces a function as its result. Instead, we use an evaluator function for symbolic objects that produces the value $\langle s \rangle(e)$ given s and e as inputs, and an evaluator for BDDs that produces the value $\langle b \rangle_{bdd}(e)$ given inputs b and e.

tegers. However, for the purposes of this theoretical exposition, the particular format of symbolic objects and the type and characteristics of environments are unimportant.

A *symbolic execution* of a function f on symbolic input s is a computation that results in a symbolic object representing the functional composition of f with $\langle s \rangle$; that is, a symbolic object s' satisfying

$$\forall e \;.\; \langle s' \rangle(e) = f\left(\langle s \rangle\,(e)\right), \tag{1.1}$$

or equivalently

$$\langle s' \rangle = f \circ \langle s \rangle.$$

We call a function f_{sym} a *symbolic counterpart* for f if it always performs a symbolic execution of f on its argument; that is,

$$\forall s, e \;.\; \langle f_{sym}\,(s) \rangle(e) = f\left(\langle s \rangle\,(e)\right), \tag{1.2}$$

or

$$\forall s \;.\; \langle f_{sym}\,(s) \rangle = f \circ \langle s \rangle.$$

These definitions generalize straightforwardly to higher-arity functions; we use a single variable only for simplicity of presentation.

Suppose we wish to prove a theorem $\forall x \;.\; \textsc{Hyp}(x) \Rightarrow \textsc{Concl}(x)$. The following metatheorem describes a set of conditions under which this conjecture is provable by symbolic execution.

Theorem 1.2.1 (Proof by symbolic execution)**.** *Suppose:*

- *Some symbolic object* $\mathbf{s}$ *covers the set of inputs satisfying* Hyp*:*

$$\forall x \;.\; \textsc{Hyp}(x) \Rightarrow \exists e \;.\; x = \langle \mathbf{s} \rangle (e) \tag{1.3}$$

- $\textsc{Concl}_{sym}$ *is a symbolic counterpart for* Concl*:*

$$\forall s, e \;.\; \langle \textsc{Concl}_{sym}\,(s) \rangle (e) = \textsc{Concl}\,(\langle s \rangle (e)) \tag{1.4}$$

- $\textsc{Concl}_{sym}(\mathbf{s})$ *yields a constant-true symbolic value:*

$$\forall e \;.\; \langle \textsc{Concl}_{sym}\,(\mathbf{s}) \rangle (e) = \mathit{true}. \tag{1.5}$$

Then

$$\forall x \;.\; \textsc{Hyp}(x) \Rightarrow \textsc{Concl}(x).$$

Proof of this theorem is by simple substitutions: given x satisfying $\textsc{Hyp}(x)$, by Equation 1.3 we have some $\mathbf{e}$ satisfying $x = \langle \mathbf{s} \rangle (\mathbf{e})$. Then

$$\begin{aligned} \textsc{Concl}(x) &= \textsc{Concl}\,(\langle \mathbf{s} \rangle\,(\mathbf{e})) \\ &= \langle \textsc{Concl}_{sym}(\mathbf{s}) \rangle (\mathbf{e}) && \text{(by Equation 1.4)} \\ &= \mathit{true} && \text{(by Equation 1.5).} \quad \square \end{aligned}$$

Our framework aids the user in satisfying these three conditions and provides automation for performing proofs in this style. We describe the implementation of symbolic execution and symbolic objects in Chapter 2. This

will clarify the process by which symbolic counterparts are defined and proven correct in order to satisfy Equation 1.4. Additionally, we will describe our symbolic object format and show how such objects may be syntactically analyzed for truth and falsehood, so that Equation 1.5 may be proven. We describe the user-level framework in Chapter 3, including the tools we provide for describing suitable symbolic input objects and proving coverage as required by Equation 1.3.

1.3 Notational Conventions

We use a pseudocode/mathematical notation to describe our algorithms rather than Lisp notation. Function names (INT-SQRT) are printed using small capitals, and non-function symbols (`T`, `NIL`) are printed in a typewriter font.

As introduced in the theory section, we use angle brackets as in $\langle x \rangle(y)$ to denote that we have interpreted the data object x as a function and applied it to argument y. Commonly, x is a symbolic object, y an environment, and $\langle x \rangle(y)$ is the evaluation of x under y. However, we use similar notation $\langle b \rangle_{bdd}(v)$ to denote evaluation of BDD b under an assignment v of Boolean values to BDD variables.

We use the propositional logic connectives $\vee$, $\wedge$, $\neg$, $\oplus$, $\Rightarrow$, and $\Leftrightarrow$ in the conventional manner. However, we also use these symbols to denote BDD operations, in which case we superscript them with an asterisk, as in $\vee^*$, $\neg^*$, and $\Rightarrow^*$.

The symbols `T` and `NIL` are associated with *true* and *false* in ACL2; however, as in Common Lisp, any non-`NIL` object is considered *true* for the purpose of testing in if-then-else contexts. In keeping with this convention, when we say an object is *true*, we mean that it is non-`NIL`; when we speak of a function being constant-*true*, its range may include non-`NIL` objects other than `T`.

Chapter 2

Implementation

In Section 1.2, we described a method for proving theorems using symbolic execution. This method requires some symbolic object format and some mechanism for performing symbolic executions. We describe the syntax and semantics of GL's symbolic objects in Section 2.1, and we present our mechanisms for performing symbolic executions in the remainder of this chapter.

One major challenge of performing symbolic executions of arbitrary user code is in handling control branches. When symbolically executing an if/then/else term, in general one may need to compute a symbolic value for the test and both the then and else branches, and finally merge these three values together into a result which represents the if/then/else term as a whole. However, in some cases the test may result in a constant-true or constant-false value; it is often important for performance to recognize these cases and evaluate only the reachable control branch. We describe our method for handling control branches within symbolic executions in Section 2.2.

In Section 2.3, we describe our method of symbolically executing primitive functions, i.e. functions that are not defined in terms of others but axiomatized to have certain behavior. We handle these by manually defining

symbolic counterparts for each of the ACL2 primitives, and additionally for some other functions for which a manually-defined symbolic counterpart gives a performance benefit. We describe in that section how we have defined these symbolic counterparts and proven them correct.

Given methods for symbolically executing control branches and primitive functions, a symbolic execution of any term may be accomplished. We describe two methods of performing such a symbolic execution; in both of these methods, the symbolic executions provably reflect ACL2 semantics and may therefore be used in proofs. In Section 2.4 we describe the first such method, which uses a code transformation that operates on some user-provided function f, producing a new function f_{sym} that is proven to be a symbolic counterpart for f. In Section 2.5 we describe the second method, in which a symbolic interpreter operates directly on user-provided terms, provably computing symbolic executions of these terms.

Using the techniques in this chapter, we can symbolically execute any recursively-defined function in the ACL2 logic. Furthermore, this symbolic execution is known to produce a correct result, i.e. a symbolic object representing the composition of the function with the symbolic inputs. This allows us to prove theorems using the method described in Section 1.2.

2.1 Symbolic Objects

The GL symbolic object format is optimized to represent object types that we expect to be commonly used in hardware verification, namely integers,

Booleans, and list structures. However, it is also capable of representing any finite function from Boolean variables into the universe of ACL2 data objects. This universe of objects consists of *conses* (ordered pairs) and *atoms*, which include symbols, complex rational numbers, characters, and strings. However, the universe is not assumed to be closed: it is not provable that every ACL2 object is of one of these types.

Well-formed symbolic objects are themselves ACL2 objects. Their allowed syntax is given by a predicate GLOBJECTP ("GL object predicate") and their semantics is defined by a series of evaluator functions. In this section we will describe their syntax and semantics, the necessity of having multiple evaluator functions, and some tools we have developed in order to ease the process of operating on symbolic objects and reasoning about such operations.

2.1.1 Symbolic Object Format

The GL symbolic object format provides structures that may contain both Boolean function objects and unconstrained variable objects. Boolean functions are represented in this format as a form of binary decision diagrams called uBDDs [12]. Evaluation of a symbolic object requires an assignment giving a Boolean value for each uBDD variable and a concrete value for each unconstrained variable. We call this assignment the evaluation environment. Each uBDD within the symbolic object structure is evaluated under the environment, yielding a Boolean value, and each unconstrained variable within the object is simply replaced by its assigned value. The structuring of the sym-

bolic object then determines how these values are combined to form an ACL2 object: bits produced by uBDDs may translate directly to Boolean values or become embedded within a numeric representation, and all symbolic objects may be combined into cons trees or if-then-else constructs.

The primary form of symbolic reasoning used in our system is manipulation of uBDDs. We therefore introduce the syntax of uBDDs before describing that of symbolic objects. The constant-true and constant-false functions are represented by the symbols `T` and `NIL`, respectively, and cons trees with `T` and `NIL` leaves denote non-constant functions. Similar to other BDD representations, the variables on which these functions depend are indexed [20] rather than given symbolic names. The function represented by a uBDD may be evaluated using an ordered list of Boolean values, where the nth value in the list is the assignment for the nth variable. We use the notation $\langle x \rangle_{bdd}(\vec{b})$ for evaluation of a uBDD x by a list of Boolean values $\vec{b}$, and v_i to denote the ith BDD variable, so that $\langle v_i \rangle_{bdd}(\vec{b})$ gives the ith element of the Boolean list $\vec{b}$.

The semantics of symbolic objects, described below, are given by evaluator functions that take a symbolic object s and an environment e. As introduced previously, we use the notation $\langle s \rangle(e)$ to denote evaluation of s under environment e. An environment e contains a list of Booleans, BddVals(e), giving the assignment of values to uBDD variables, and an association list, VarVals(e), giving values (of arbitrary type) to named unconstrained variables. An evaluator function evaluates the uBDDs throughout s to Boolean values, and replaces each unconstrained variable in s with its assigned value.

The format of s then dictates how these values are combined to form a concrete result.

GL allows multiple different evaluator functions to be defined because of the **App** symbolic object construct, which, as described below, represents a function applied to symbolic arguments. Because ACL2 does not allow functions as first-class objects, every evaluator has a fixed set of function names that it recognizes and can apply. If the named function is not in this set, the evaluation of a symbolic object applying that function is undefined. However, new evaluators may be defined that recognize additional function symbols. Most theorems proved about a previous evaluator, including symbolic counterpart correctness conjectures, may be used to prove similar theorems about such an extended evaluator by functional instantiation [11]. Our framework does not allow malformed evaluators, such as ones which apply the wrong function for a given symbol, to be used in proofs; the result produced by any evaluator on a given symbolic object and environment is the same as that produced by any other evaluator when it is defined.

We now list the types of symbolic object constructs, their syntax and their evaluation semantics. The function BitsToNumber, used in evaluating the **Num** construct, is described below.

Atom. *Syntax:* Any non-cons object *obj* other than the symbols `:CONCR`, `:BOOL`, `:NUM`, `:ITE`, `:VAR`, or `:APP`. *Evaluation: obj* itself.

Cons. *Syntax:* `(`x `.` y`)` where x and y are well-formed symbolic objects. *Evaluation:* `(`$\langle x \rangle(e)$ `.` $\langle y \rangle(e)$`)`.

Concr. *Syntax:* `(:CONCR .` *obj*`)` for any object *obj*. *Evaluation:* *obj* itself.

Bool. *Syntax:* `(:BOOL .` b`)` where b is a well-formed uBDD. *Evaluation:* $\langle b \rangle_{bdd}(\textsc{BddVals}(e))$.

Num. *Syntax:* `(:NUM .` *bits*`)`, where *bits* is a list of four or fewer lists of uBDDs. *Evaluation:* $\textsc{BitsToNumber}(bits, e)$.

Ite. *Syntax:* `(:ITE` *test then* `.` *else*`)` where *test*, *then*, and *else* are well-formed symbolic objects. *Evaluation:* If $\langle test \rangle(e)$, then $\langle then \rangle(e)$, else $\langle else \rangle(e)$.

Var. *Syntax:* `(:VAR .` v`)` for any v. *Evaluation:* $\textsc{Lookup}(v, \textsc{VarVals}(e))$.

App. *Syntax:* `(:APP` f `.` *args*`)` where f is a symbol and *args* is a well-formed symbolic object. *Evaluation:* $f(\langle args \rangle(e))$, if the evaluator recognizes f, otherwise undefined. Here $\langle args \rangle(e)$ is expected to be a list of length equal to the arity of f.

The BitsToNumber function evaluates the four or fewer lists of uBDDs from a **Num** form and uses the resulting bits to construct a complex rational number. The four possible lists represent the numerator and denominator of the real part, then the numerator and denominator of the imaginary part. Numerators are interpreted as two's-complement and default

Symbolic Object	Coverage Set
(:BOOL . v_0)	$\{\texttt{T}, \texttt{NIL}\}$
12	$\{12\}$
(:NUM (v_0 NIL))	$\{0, 1\}$
(:NUM (v_0 v_1))	$\{-2, -1, 0, 1\}$
(:NUM (v_0 T v_0))	$\{-1, 2\}$
(:ITE (:BOOL . v_0) A . (:NUM (v_1)))	$\{\texttt{A}, -1, 0\}$
(:CONCR . (:NUM (v_0)))	$\{$(:NUM (v_0))$\}$
((:CONCR . :CONCR) . X)	$\{$(:CONCR . X)$\}$
(:VAR . Z)	U
(:APP BOOLEANP (:VAR . Z))	$\{\texttt{T}, \texttt{NIL}\}$

Table 2.1: Symbolic Object Examples

to 0 when not present; denominators are interpreted as unsigned and default to 1 when not present.

We list some examples of symbolic objects with their coverage sets in Table 2.1. In the final example, U signifies the entire universe of ACL2 objects.

In the symbolic object format, the keyword symbols :CONCR, :BOOL, :NUM, :ITE, :VAR, and :APP act as syntax markers. Due to the **Atom** and **Cons** forms, any object that does not contain one of the syntax marker keywords is a well-formed symbolic object that evaluates to itself. The **Concr** form provides a way to represent objects which do contain these symbols. This together with the **Bool** and **Ite** constructs allow this format to represent any function from finitely many Boolean variables to ACL2 objects. The **Num**

form is redundant in the sense that it does not allow any additional functions to be represented, but it provides a more compact representation for numbers, which is desirable for efficiency of symbolic execution. The **Var** form may represent any object at all and is useful for symbolic executions in which certain inputs are irrelevant. Finally, the **App** form represents a function applied to symbolic arguments, as described above.

In order to prove facts about our core symbolic object system, we define a generic evaluator GEval which does not recognize any function symbols in the **App** form. Because every other symbolic object evaluator is an extension of GEval, theorems about this function may be functionally instantiated to prove similar theorems about other evaluators; it is therefore convenient to prove all necessary lemmas about GEval initially, from which the analogous lemmas about other evaluators may be derived. Despite the possible existence of many evaluators, we will continue to use the notation $\langle s \rangle(e)$; the specific symbolic object evaluator in use is rarely important, since most any fact proven about one evaluator can be proven about another that recognizes a superset of the original evaluator's function symbols.

2.1.2 Constructors, Accessors, and Recognizers

In order to reason about symbolic objects, such as when proving the correctness of symbolic counterpart functions, it is convenient to ignore the concrete syntax of the symbolic objects and instead reason at a more abstract level, using rules regarding constructor, accessor, and recognizer functions. In

this subsection we describe concepts we use in order to reason about symbolic objects at a higher level of abstraction.

We use the *Defaggregate* library of Davis [30] to define constructor, recognizer, and accessor functions for each tagged type of symbolic object. For the **Bool** form, for example, this defines a constructor Bool which takes an input b and creates a tagged object `(:BOOL . b)`, an accessor BoolBdd that extracts b from such an object, and a predicate BoolP that recognizes **Bool** objects. The *Defaggregate* forms introducing the six tagged types produce the following functions:

Constructors: Bool, Num, Concr, Ite, App, Var;

Recognizers: BoolP, NumP, ConcrP, IteP, AppP, VarP;

Accessors: BoolBdd, NumBits, ConcrObj, IteTest, IteThen, IteElse, AppFn, AppArgs, VarName.

The *Defaggregate* tool also automatically introduces various rewrite rules for reasoning about the functions it introduces. For example, the following rule is proven about the constructor function Ite and the accessor IteTest:

$$\textsc{IteTest}(\textsc{Ite}(\mathit{test}, \mathit{then}, \mathit{else})) = \mathit{test}.$$

We also separately prove theorems regarding the evaluation of these forms by GEval. For example, the evaluation theorem for the Ite constructor

is as follows:

$$\langle \textsc{Ite}(test, then, else)\rangle(env) = \textbf{if } \langle test\rangle(env) \\ \textbf{then } \langle then\rangle(env) \\ \textbf{else } \langle else\rangle(env).$$

These constructors, accessors, and recognizers are strictly syntax oriented functions; it is often more convenient to use functions whose behavior is less rigid with regard to syntax. For example, when constructing a new symbolic object, most often we want to create an object with certain semantics, rather than a particular syntax; we would prefer the resulting syntactic representation to be as simple as possible. For example, we would prefer to create the atom 1 rather than the number `(:NUM (T NIL))`. We implement simplifying constructors to achieve this. It is also often the case that we want to know what type of concrete object a given symbolic object may represent, rather than its syntactic type. We therefore define extended recognizers for certain types of symbolic objects. Finally, we define generalized accessors so that when the syntactic type of an object is not known, we may still perform operations semantically analogous to extracting fields of these objects.

We define four simplifying constructors:

- MkBool(*bdd*): if *bdd* is a constant `T` or `NIL`, then *bdd*, otherwise Bool(*bdd*);

- MkNum(*bits*): if *bits* is a list of lists of Booleans (that is, constant-valued BDDs), then the concrete number described by those bits; otherwise Num(*bits*);

- MkIte(*test*, *then*, *else*): *then* or *else* as appropriate if either the truth value of *test* may be determined syntactically, or if *then* and *else* are syntactically equal; Ite(*test*, *then*, *else*) otherwise;

- MkConcr(*obj*): Concr(*obj*) if *obj* contains a syntax marker keyword, otherwise *obj* itself.

Each simplifying constructor is proven to be equivalent under evaluation to the corresponding simple syntactic constructor. For example,

$$\begin{aligned}\langle \textsc{MkBool}(b)\rangle(env) &= \langle b\rangle_{bdd}(env)\\ &= \langle \textsc{Bool}(b)\rangle(env).\end{aligned}$$

We define generalized recognizers for the following four categories of symbolic objects:

- *General Booleans*: **Bool** and Boolean-valued **Atom** and **Concr** objects, recognized by GenBoolP;

- *General Numbers*: **Num** and numeric-valued **Atom** and **Concr** objects, recognized by GenNumP;

- *General Conses*: **Cons** and cons-valued **Concr** objects, recognized by GenConsP;

- *General Concretes*: **Atom** objects, **Concr** objects, and cons trees of **Atom** and **Concr** objects, recognized by GenConcrP.

The last category is somewhat unlike the first three. The first three categories each contain symbolic objects that must evaluate to a particular type. On the other hand, the distinguishing feature of general concrete objects is that they may only evaluate to one possible value, which may be determined syntactically. This category also overlaps with the other three categories, which are themselves disjoint.

These extended recognizers are not intended to recognize all symbolic objects that always evaluate to the appropriate type, but rather a useful subset of symbolic objects based on their evaluation semantics. For example, an **Ite** object will always evaluate to a Boolean if both of its branches always evaluate to Booleans, but such an **Ite** object is not considered a general Boolean.

We define the following extended accessors that act like the corresponding simple syntactic accessors but may operate on symbolic objects within the general category, rather than a single syntactic type:

- GenBoolBdd returns the BDD reflecting the value of a generalized Boolean: the bdd field of a **Bool** object, or the (constant) value of a Boolean atom, which is a syntactically valid BDD itself.

- GenNumBits returns four values, each a list of BDDs reflecting, respectively, the bits of the numerator and denominator of the real and

imaginary parts of a generalized number. When the argument is a **Num** object, this function simply extracts the BITS field; otherwise the argument must be a numeric atom, which is exploded into lists of Booleans.

- GENCONSCAR returns a symbolic object representing the CAR of a generalized cons. This is CAR itself on the **Cons** symbolic object type, or a concrete object constructed from the CAR of the CONCROBJ of a cons-valued **Concr** object.

- GENCONSCDR returns a symbolic object representing the CDR of a generalized cons, analogously to GENCONSCAR.

- GENCONCROBJ returns the unique possible value of a generalized concrete object. This may be obtained by traversing the outer cons structure of the object, extracting the value from each **Concr** object within the tree.

Each of these generalized accessors has a corresponding theorem showing its relation to the evaluation semantics of its argument. For example,

$$\textsc{GenBoolP}(s) \Rightarrow \langle s \rangle(env) = \left\langle \textsc{GenBoolBdd}(s) \right\rangle_{bdd}(env).$$

2.2 Control Flow

In general, a symbolic execution progresses in much the same way as a concrete execution. ACL2 uses an eager evaluation strategy: to execute a function composition term $f(g(x))$, it first evaluates $g(x)$ and subsequently f

of the result. Similarly, to symbolically execute the term, GL first symbolically executes $g(x)$ and then symbolically executes f on the result.

However, in both ACL2's concrete execution and our symbolic execution strategy, **if** is evaluated lazily as a special case. To concretely evaluate an **if** term

if *test* **then** *then* **else** *else*,

ACL2 first evaluates *test*, yielding a value which determines which of the branches to execute: if the result is the special symbol `NIL` (signifying *false*), *else* is evaluated; otherwise, *then*. In symbolic execution of the same term, GL similarly first symbolically executes *test*. However, the resulting value may be a symbolic object that may be `NIL` (*false*) under some environments and non-`NIL` (*true*) under others. Therefore, GL may need to symbolically execute both branches. To make this determination, the GL control-flow algorithm first analyzes the result of the test to determine whether it is constant-*true*, constant-*false*, or may take either value. If it is constant, then only one branch needs to be symbolically executed. Otherwise, both branches are symbolically executed and their results are merged into a single symbolic value representing the symbolic result of the **if** term as a whole.

2.2.1 Path Conditions

Analyzing the test term in isolation may not be enough to avoid executing branches that are unreachable, because the context in which the term occurs affects the possible values of the test. For example, suppose that x may

take the values 1, 2, or 3, and we are interested in the symbolic execution of the following program:

```
1: if x ≥ 2 then
2:     return 0;
3: else
4:     if x is even then
5:         return f(x);
6:     else
7:         return g(x);
```

Analyzing the test on line 4 in isolation does not reveal that only one of its branches is possible; indeed, the test is true for $x = 2$ and false for $x = 1$. However, x is constrained by the previous test to be less than 2 at this point; thus the control flow cannot reach that block with an even value of x. So, to avoid needlessly symbolically executing $f(x)$, we must consider the context in which the test occurs.

To do this, we use a scheme similar to that described by King [53]. For a particular execution path to be realizable in a concrete execution, the values taken by the tests in each **if** term along that path must be consistent. Therefore, when analyzing a test to determine its possible truth or falsehood, GL checks the test values for consistency with a *path condition*, represented as a uBDD, which is accumulated from previous tests along the current execution path. Whenever symbolic execution follows a **then** or **else** branch, the path condition is updated by conjoining it with the test or its negation, respectively. This prevents symbolic executions from following execution paths containing

contradictory **if** tests.

The path condition represents the portion of the input space that is relevant to the current symbolic execution. Earlier, when we described the correctness condition of symbolic executions, we omitted the path condition as an input parameter for simplicity; our symbolic executions still conform to this condition when the path condition is *true*. To be more precise, we only expect the correctness condition to hold for environments on which the path condition holds. Therefore, recalling the definition of symbolic execution in Section 1.2, we define a symbolic execution of function f on s under path condition pc as a computation producing s', where

$$\forall e \;.\; \langle pc \rangle_{bdd}(e) \Rightarrow \langle s' \rangle(e) = f\left(\langle s \rangle\,(e)\right).$$

2.2.2 Analyzing If Tests

GL's analysis of **if** tests aims to construct a BDD which reflects the truth or falsehood of the symbolic value of the test. This is complicated by the possible presence of **App** and **Var** objects in that symbolic value. In absence of these forms, a symbolic object can be syntactically transformed into a BDD that has the same truth value, under all assignments, as the original object. However, the **Var** form does not permit this because its truth value is independent of a BDD variable assignment, and the **App** form does not permit this because the semantics of applied functions are not considered during this analysis. Therefore, the analysis algorithm instead produces a pair of BDDs: the first, k ("known"), represents the domain on which the analysis

was able to determine the truth value, and the second, v ("value"), represents the truth value when it is determined.

This analysis is implemented by the function GTESTS, which takes the symbolic object s to be analyzed, and the path condition pc (represented as a BDD). We have proven in ACL2 the following correctness theorem which describes the intended operation of GTESTS:

Theorem 2.2.1 (Correctness of GTESTS)**.** *Let*

$$(k, v) = \text{GTESTS}(s, pc).$$

Then for all environments e satisfying $\langle pc \rangle_{bdd}(e)$ and $\langle k \rangle_{bdd}(e)$,

$$\langle s \rangle(e) \Leftrightarrow \langle v \rangle_{bdd}(e).$$

The GTESTS algorithm is shown in Algorithm 2. (Recall that we use $\neg^*$, $\wedge^*$, etc. to denote BDD operations.) This algorithm analyzes the input object s by cases on its type. For most of the symbolic object types, the analysis is trivial. For all types other than **App**, **Var**, and **Ite**, the truth value is known. The **Cons** and **Num** types are unconditionally known true, since all concrete values represented by these objects are non-`NIL`, and for the **Atom** and **Concr** types the truth value is simply determined by fixing the represented object to a Boolean. The truth value for a **Bool** object is its BDD field. APP and VAR objects are unknown, so that the truth value returned is irrelevant (we arbitrarily choose `NIL`).

Algorithm 2

```
1: procedure GTESTS(s, pc)
2:     if s is a Num or Cons then
3:         return (T, T)
4:     else if s is a Bool then
5:         return (T, BOOLBDD(s))
6:     else if s is an Atom then
7:         return (T, if s then T else NIL)
8:     else if s is a Concr then
9:         return (T, if CONCROBJ(s) then T else NIL)
10:    else if s is an App or Var then
11:        return (NIL, NIL)
12:    else                                              ▷ s is an Ite
13:        test ← ITETEST(s)
14:        (k_test, v_test) ← GTESTS(test, pc)
15:        allk_test ← (pc ⇒* k_test) = T
16:        if allk_test ∧ (pc ⇒* v_test) = T then
17:            return GTESTS(ITETHEN(s), pc)
18:        else if allk_test ∧ (pc ⇒* ¬* v_test) = T then
19:            return GTESTS(ITEELSE(s), pc)
20:        pc_then ← pc ∧* (v_test ∨* ¬* k_test)
21:        pc_else ← pc ∧* (¬* v_test ∨* ¬* k_test)
22:        (k_then, v_then) ← GTESTS(ITETHEN(s), pc_then)
23:        (k_else, v_else) ← GTESTS(ITEELSE(s), pc_else)
24:        v ← BDDITE(v_test, v_then, v_else)
25:        k_branches ← BDDITE(v_test, k_then, k_else)
26:        same ← v_then ⇔* v_else
27:        k ← BDDITE(k_test, k_branches, same ∧* k_then ∧* k_else)
28:        return (k, v)
```

The **Ite** case is the only complicated one. First the **Ite** object's test is analyzed in a recursive call. In lines 15 through 19, GTESTS checks whether the test is either known to be constant true under the *pc*, in which case GTESTS recurs only on the ITETHEN field, or constant false, in which case GTESTS recurs only on the ITEELSE. In both cases the appropriate *pc* for the recursive call is the current one, since that branch is followed under all the cases that the current path is followed.

Otherwise, in lines 20–21 the *pc* is adjusted for each of the branches. The computation of the *pc* values for the two branches reflects the conditions under which each branch might be reachable. There are two conditions under which a branch is determined to be unreachable: if its containing ITE expression is itself unreachable ($\neg^* pc$), or if the test's value is known to be the one under which the other branch is taken ($\neg^* v_{test} \wedge^* k_{test}$ for the **then** branch, $v_{test} \wedge^* k_{test}$ for the **else** branch). When the test value is not known, it does not rule out the reachability of either branch. The new *pc* values for the branches represent the case where neither of these unreachability conditions hold.

Wherever the truth value of the **Ite** object is known, it is simply the BDD if-then-else of its test and branches, computed on line 24. The condition under which this value is known is more complicated. If the test is known, then the object's truth value is known if the relevant branch is known. If the test is unknown, then the object's truth value is only known if both branches are known and their truth value is the same. The BDD combining these cases is computed on line 27.

Analysis of an **if** test using GTESTS allows us to determine a new path condition for each branch; this, in turn, determines whether both branches or just one must be symbolically executed. A given branch must be executed under conditions where the current path condition holds and either the test's truth value is unknown or it corresponds to the direction of that branch: *true* for the **then** branch, *false* for the **else** branch. Therefore, given the current path condition *pc* and values

$$(k_{test}, v_{test}) = \text{GTESTS}(test, pc),$$

the new path conditions for the **then** and **else** branches are $pc \wedge^* (\neg^* k_{test} \vee^* v_{test})$ and $pc \wedge^* (\neg^* k_{test} \vee^* \neg^* v_{test})$, respectively. If one of these path conditions is `NIL` (constant-*false*), then that branch is skipped. The path condition given at the top level of a symbolic execution should be either *true* or non-constant; therefore it is an invariant that the current path condition is never constant-*false*. Under this invariant, at least one of the branches always has a satisfiable path condition and therefore one or both of the branches is always run.

2.2.3 Merging Branches

If one of the branches has a constant-*false* path condition, then the symbolic result of the if term is simply the result of symbolically executing the other branch.[1] Otherwise, GL symbolically executes both branches and must merge both symbolic results into a single symbolic value. This result must

[1] On the other hand, if one of the branches has a constant-*true* path condition, the other branch's must be constant-*false*.

take the value of the **then** branch when the test evaluates to *true* and of the **else** branch when the test evaluates to *false*. This object is constructed using the results (k_{test}, v_{test}) from the GTESTS analysis and the results s_{then} and s_{else} from the symbolic executions of the two branches. This takes the following form:

$$\begin{aligned}&\textsc{MkIte}(\textsc{MkBool}(k_{test}),\\ &\qquad \textsc{IteMerge}(v_{test}, s_{then}, s_{else}, pc),\\ &\qquad \textsc{MkIte}(s_{test}, s_{then}, s_{else})).\end{aligned}$$

Here, ITEMERGE is a function that constructs a new symbolic object representing an if-then-else among the BDD v_{test} and the two symbolic objects s_{then} and s_{else}, under the assumption of the pc. Under this assumption and the correctness statement of GTESTS above, this term reduces to the if-then-else of the symbolic results from the test, then, and else branches, as desired.

The correctness condition required of ITEMERGE is described by the following theorem, which we have proved in ACL2:

Theorem. *For all environments e such that* $\langle pc \rangle_{bdd}(e)$*,*

$$\begin{aligned}\big\langle \textsc{IteMerge}(v_{test}, s_{then}, s_{else}, pc)\big\rangle(e) = \ &\textbf{if } \langle v_{test}\rangle_{bdd}(e)\\ &\textbf{then } \langle s_{then}\rangle(e)\\ &\textbf{else } \langle s_{else}\rangle(e).\end{aligned}$$

To satisfy this correctness requirement, it would suffice to produce a new **Ite** object as follows:

$$\textsc{MkIte}(\textsc{MkBool}(v_{test}), s_{then}, s_{else}).$$

However, rather than creating a new **Ite** object, it is often possible to merge the two branches into a simpler structure, which is generally preferable. For example, when the two branch results are both **Bool** objects, the if-then-else may be merged into a single **Bool** whose value is the BDD if-then-else of v_{test} and the BDD values of the two branch objects. Our implementation of IteMerge merges the two branches whenever they conform to one of the following five cases:

- They are equal. Their merge is then their identical value.
- They are both general numbers (as defined in Subsection 2.1.2): either **Num** objects, **Concr** objects whose values are numeric, or numeric atoms. Their merge is then the numeric object formed from the BDD if-then-else of v_{test} with each of their respective component bits:

$$\begin{aligned}\textsc{MkNum}\big(\textsc{BddIteListList}(&v_{test},\\ &\textsc{GenNumBits}(s_{then}),\\ &\textsc{GenNumBits}(s_{else}))\big).\end{aligned}$$

- They are both general Booleans: either **Bool** objects, **Concr** objects whose values are Boolean, or Boolean atoms. Their merge is the Boolean

symbolic object formed from the BDD if-then-else of v_{test} with their respective BDD values:

$$\textsc{MkBool}\big(\textsc{BddIte}(v_{test}, \textsc{GenBoolBdd}(s_{then}), \textsc{GenBoolBdd}(s_{else}))\big).$$

- They are both general conses: either **Cons** objects or **Concr** objects whose values are conses. Their merge is the recursive IteMerge of their Cars and Cdrs:

$$\begin{aligned}
&\textsc{Cons}\big(\textsc{IteMerge}(v_{test},\\
&\qquad\qquad \textsc{GenConsCar}(s_{then}),\\
&\qquad\qquad \textsc{GenConsCar}(s_{else}),\\
&\qquad\qquad pc),\\
&\qquad \textsc{IteMerge}(v_{test},\\
&\qquad\qquad \textsc{GenConsCdr}(s_{then}),\\
&\qquad\qquad \textsc{GenConsCdr}(s_{else}),\\
&\qquad\qquad pc)\big).
\end{aligned}$$

- They are both **App** objects applying the same function. Their merge is the App of that function with the recursive IteMerge of their arguments:

$$\begin{aligned}
&\textsc{App}\big(\textsc{AppFn}(s_{then}),\\
&\qquad \textsc{IteMerge}(v_{test}, \textsc{AppArgs}(s_{then}), \textsc{AppArgs}(s_{else}), pc)\big).
\end{aligned}$$

In addition to cases in which s_{then} and s_{else} conform to one of the above conditions, ITEMERGE also specially handles cases where one or both are **Ite** objects whose ITETEST fields are of the **Bool** type. In this case, ITEMERGE walks over the **Ite** structures of s_{then} and s_{else} to find pairs of sub-branches, one within s_{then} and one within s_{else}, which may be merged.

To simplify the process of finding mergeable pairs of sub-branches, ITEMERGE maintains an invariant on the **Ite** structures of symbolic objects:

- **Ite** nests are structured such that the ITETHEN branches are never themselves **Ite** objects. Therefore, iterating over the **Ite** structure of an object involves a list-like traversal, recurring on the ITEELSE branch, rather than a tree-like traversal in which ITEMERGE would need to recur on both branches.

- Non-**Ite** leaves are ordered within each object. This ordering gives some precedence to every symbolic object, where the precedence for two objects is equal if and only if those two objects are mergeable, meeting one of the five conditions described above. In a non-nested **Ite** object, the ITETHEN branch must have higher precedence than the ITEELSE branch, and in any nested **Ite**, the outermost ITETHEN branch must have higher precedence than the ITETHEN of the ITEELSE branch.

Note that this invariant implies that there is at most one branch of such an **Ite** structure that is a general Boolean, at most one branch that is a general number, etc. Because **Ite** branches are strictly ordered by precedence and

mergeable objects have equal precedence, an **Ite** may not contain two branches that are mergeable.

ITEMERGE operates under the assumption that the s_{then} and s_{else} inputs obey this invariant. Under this assumption a linear traversal of the **Ite** structures of these objects suffices to complete the merge, producing a result that also obeys the invariant. This linear traversal, similar to that of a mergesort, ensures that if the **Ite** branches in s_{then} and s_{else} are sorted, then the merged result will also be sorted.

The linear traversal of the branches' **Ite** structures operates by separating the two input objects each into a *head* and *tail*. The head of an object is the ITETHEN branch if the object is a **Ite** with **Bool** test, and the whole object itself if not. The tail is the ITEELSE branch or else, arbitrarily, `NIL`; the tail is irrelevant in the non-**Ite** case. ITEMERGE checks the relative precedence of the two head objects. If they have equal precedence, then they are merged and ITEMERGE is called recursively on the two tails, composing the merged heads and tails into a final **Ite** object when necessary. Otherwise, as in a mergesort, ITEMERGE recurs by merging the tail of the object whose head took precedence with the entirety of the other object. Again, ITEMERGE then creates an **Ite** node combining the selected head with the merged tails.

While the sorting and structure invariant is algorithmically convenient and helps to find mergeable objects, it is only heuristic. If either or both of the branches do not obey the invariant, the ITEMERGE algorithm still conforms

to its correctness claim, though the object produced may not be sorted or maximally merged.

2.2.4 Full Control Flow Algorithm

The full algorithm for handling of a control branch is described in pseudocode in Algorithm 3. In this listing, inputs *test*, *then*, and *else* are the subterms of an **if** block. Rather than implementing this algorithm as a function, we instead define it as a macro, effectively replicating this code structure wherever we require a symbolic if-then-else expression. This structuring allows us to evaluate the branches lazily. In this pseudocode listing, we use the notation SExec($term, pc$) to denote symbolic execution of the given term under the given path condition.

To evaluate an if-then-else expression, we first symbolically execute the test under the current path condition and analyze the result using Gtests. The results k_{test} and v_{test} describe the Boolean conditions under which the test is known and true, respectively. These allow us to compute the appropriate new path conditions for each of the branches. A given branch is relevant (its path condition is true) only if the top-level path condition is true, and only if the test either produced the value indicating that branch or else an unknown result. New path conditions representing these conditions are computed in lines 4–5.

The path conditions for the branches indicate whether or not they must be symbolically executed: if the path condition for a branch is `NIL`, then there

Algorithm 3

```
1: function SExecIf(test, then, else, pc)
2:     s_test ← SExec(test, pc)
3:     (k_test, v_test) ← GTests(s_test, pc)
4:     pc_then ← pc ∧* (¬* k_test ∨* v_test)
5:     pc_else ← pc ∧* (¬* k_test ∨* ¬* v_test)
6:     if pc_then ≠ NIL then
7:         s_then ← SExec(then, pc_then)
8:     else
9:         s_then ← NIL
10:    if pc_else ≠ NIL then
11:        s_else ← SExec(else, pc_else)
12:    else
13:        s_else ← NIL
14:    pc_merge ← pc ∧* k_test
15:    s_merge ← IteMerge(v_test, s_then, s_else, pc_merge)
16:    s_unknown ← MkIte (s_test, s_then, s_else)
17:    return MkIte (MkBool(k_test), s_merge, s_unknown)
```

is no environment under which that branch is reachable, and we may skip its evaluation. We arbitrarily assign `NIL` as the symbolic result of an unreachable branch.

After computing the symbolic values of the branches, these values must be merged into a single result. As described in Subsection 2.2.3, we use separate merging strategies for the situations where the test's value is known or unknown. When the test is unknown, we simply produce an **Ite** of the three symbolic results. When the test is known, we call IteMerge in order to simplify the merger of the two branches. Since this merge is only relevant when the test is known, we perform the IteMerge under a new path condition which conjoins k_{test} with the original path condition.

To ensure the correct operation of our control flow algorithm, we have proved a metatheorem stating the following correctness result. This uses the correctness theorems of GTESTS and ITEMERGE along with the evaluation properties of the **Ite** and **Bool** types.

Theorem. *For all environments e satisfying $\langle pc \rangle_{bdd}(e)$, and letting pc_{then} and pc_{else} be defined as in the definition of* SEXECIF*,*

$$\begin{aligned}\big\langle \text{SEXECIF}(test, then, else, pc)\big\rangle(e) = \ & \textbf{if } \langle \text{SEXEC}(test, pc)\rangle(e) \\ & \textbf{then } \langle \text{SEXEC}(then, pc_{then})\rangle(e) \\ & \textbf{else } \langle \text{SEXEC}(else, pc_{else})\rangle(e).\end{aligned}$$

2.3 Symbolic Primitives

The ACL2 logic includes several primitive functions whose behavior is constrained by axioms, such as +, CONS, CAR, and STRINGP. Other functions, whether user-defined or built-in, either are recursively defined in terms of these primitives or, like the primitives, are constrained by axioms but not defined. Our code transformer and symbolic interpreter both permit symbolic execution of any functions that are recursively defined in terms of functions with defined symbolic counterparts. However, we have no automated mechanism for generating symbolic counterparts for functions that are constrained rather than defined, so we have manually defined symbolic counterpart functions for the primitives and proven their correctness by conventional theorem proving.

We follow certain conventions when defining symbolic counterparts. For each function f of arity n, we define its symbolic counterpart f_{sym} so that the following properties hold, and prove them where applicable.

Arity. f_{sym} has arity $n+2$; the first n formals correspond to the formals of f, and the additional two are the path condition *pc* and a natural number *sdepth* measuring the stack depth of the symbolic execution in order to ensure termination.

Guards. f_{sym} is Common Lisp compliant under the guard that its first n formals are well-formed symbolic objects, *pc* is a well-formed BDD, and *sdepth* a natural number.[2]

Well-formedness. f_{sym} produces a well-formed symbolic object.

Correctness. f_{sym} satisfies the symbolic counterpart correctness criterion: for all environments e satisfying $\langle pc \rangle_{bdd}(e)$,

$$\left\langle f_{sym}(s_1, \ldots, s_n, pc, sdepth) \right\rangle(e) = f\left(\langle s_1 \rangle(e), \ldots, \langle s_n \rangle(e)\right).$$

When our code transform generates symbolic counterpart functions from an existing function definition, the control structure mirrors that of the

[2]ACL2 features a mechanism called guards, which allow ACL2 functions to be soundly executed by the underlying Lisp. A function's guard determines which inputs to that function are well-formed. One may perform a certain proof to verify the guard of a function, which shows that execution of that function in Common Lisp behaves as ACL2's logical semantics dictate provided its inputs are well-formed. If a function's guard is not verified, then a ACL2 instead executes a "safe" version of its definition which is faithful to the ACL2 logic even in the case of badly-typed arguments [48]. In certain cases the safe version may be drastically slower than raw Lisp execution.

original function. In contrast, most manually defined symbolic counterparts split into cases based on the symbolic object types of the inputs. First, as a special case, when all of the symbolic inputs are general-concrete objects (satisfying GenConcrP), then the original function is called on their GenConcrObj values. This allows symbolic executions to short-circuit into concrete executions when all of the inputs to the function are concrete. Then, if any input is of the **Ite** type, the symbolic counterpart recurs on one or more of the branches, depending on the analysis of the test object under the path condition; this process is similar to the SExecIf algorithm. If any inputs are **App** and **Var** forms, these are usually handled by creating a new **App** wrapping the symbolic arguments.

Handling the **Ite**, **App**, and **Var** cases eliminates three of the eight symbolic object types. The inputs are then known to be of one of the types **Atom**, **Cons**, **Concr**, **Bool**, or **Num**. Usually, these are regrouped into categories according to the type of object represented rather than the type of the symbolic object; these categories are delineated using the generalized recognizers GenBoolP, GenNumP, etc., described in Subsection 2.1.2.

ACL2 primitives often are intended to only operate on a certain type; inputs of the wrong type are typically fixed to some value. For example, arithmetic operations usually consider non-numeric inputs equivalent to zero. This helps to reduce the number of cases that must be considered in each symbolic counterpart definition; all inputs not in the general number category may be fixed to zero, after which all operands are known to be general numbers.

Once a symbolic counterpart has focused on a case in which operands are known to be a particular type, the implementation either simply performs syntactic manipulations of the input objects or else uses uBDD operations to implement some symbolic manipulation. Most of the situations in which uBDD manipulation is required occur within arithmetic-related functions. For these we use helper functions that simulate bit-level arithmetic implementations on lists of uBDDs; for example, we use a ripple-carry algorithm to implement symbolic additions. We prove these algorithms correct and use these results to prove the symbolic counterpart correctness criterion for each symbolic primitive.

In certain cases, a primitive symbolic counterpart may produce an **App** object representing the application of itself to its symbolic arguments rather than performing a more explicit symbolic computation. For example, our symbolic implementations of most arithmetic operations will only perform uBDD-based arithmetic on symbolic integers, since symbolic arithmetic on rationals and complex numbers tends to be impractically expensive. These operations produce an **App** object when given a symbolic input which could represent a non-integer.

When necessary for performance, the user may hand-code a symbolic counterpart rather than allowing it to be generated automatically. However, this generally requires a high degree of proof effort, so automatic generation is usually preferred. We have hand-coded and proven the necessary properties of symbolic counterparts for several non-primitive functions, such as Common

Lisp's bitwise logical operations; these have ACL2 function definitions that are very inefficient to symbolically execute directly. We have also done this for an and-inverter graph evaluation routine that is used in our hardware verification efforts; this is described in Section 4.2.

2.4 Symbolic Counterpart Generation

To generate symbolic counterparts for a set of non-primitive functions, GL includes a code transformation routine that may be applied to each function definition. This code transformation replaces each term in the function body with a new term that carries out a symbolic execution of the original term. In this section we describe the code transformation algorithm and our strategy for automating the correctness proofs of the resulting symbolic counterparts.

All steps of the code transformation operate on ACL2 *translated* terms. In this subset of the ACL2 language, there are only four syntactic forms: variables, constants, lambda applications, and function applications. In particular, this subset does not include macros or the **let** form. Every well-formed ACL2 term may be translated by expanding away macro calls and replacing **let** forms with equivalent lambda applications. The syntactic forms are structured as Lisp S-expressions, as follows:

Variable. v, a legal variable symbol.

Constant. `(QUOTE` *obj*`)`, where *obj* may be any ACL2 object.

Lambda application. `((LAMBDA (`$v_1 \ldots v_n$`)` *body*`)` $a_1 \ldots a_n$`)`, where *body* is a subterm, $v_1 \ldots v_n$ is a duplicate-free list of legal variable symbols which includes all free variables in *body*, and $a_1 \ldots a_n$ is a list of subterms. In mathematical notation, we write

$$(\lambda v_1, \ldots, v_n \ .\ body)(a_1, \ldots, a_n).$$

Function application. `(`$f\ a_1 \ldots a_n$`)`, where f is a function symbol of arity n and $a_1 \ldots a_n$ is a list of subterms. In mathematical notation, we write $f(a_1, \ldots, a_n)$. As a special case, f may be **if**, in which case there are three arguments and we instead write **if** a_1 **then** a_2 **else** a_3.

Our code transformation generates a symbolic counterpart for a function essentially by replicating the structure of its definition, but substituting a symbolic counterpart call for each function call and a control flow code structure for each **if** term. In order to do this, every function called in the definition must already have a symbolic counterpart defined. Therefore, as the first step to generating a symbolic counterpart for a given set of functions, the code transformation algorithm first constructs their complete call graph and obtains a topological ordering of all functions in that graph, then generates their symbolic counterparts in that order.

The code transformation routine also proves that the functions generated are indeed symbolic counterparts, showing that for f_{sym} generated from original function f,

$$\langle pc \rangle_{bdd}(e) \Rightarrow \left\langle f_{sym}(s_1, \ldots, s_n, pc, sdepth) \right\rangle(e) = f\left(\langle s_1 \rangle\,(e), \ldots, \langle s_n \rangle(e)\right).$$

These proofs require a suitable symbolic object evaluator $\textsc{eval}(s, e) = \langle s \rangle(e)$. As described in Section 2.1.1, evaluators differ in the set of functions they recognize in **App** objects. A symbolic counterpart generated for a function f may create **App** objects representing calls of f — in particular, when *sdepth* is exhausted, it produces such an **App** object in order to terminate immediately while still ensuring that the above equation holds. Therefore, the code transformation introduces a new evaluator that recognizes every function for which a symbolic counterpart is to be created. It additionally includes every function that any previous evaluator recognizes, so that the new evaluator will be backwards compatible with previously defined evaluators. For previously introduced symbolic counterparts, this backward compatibility allows existing symbolic counterpart correctness theorems to be updated to ones formulated in terms of the new evaluator.

To streamline the proofs done in the course of symbolic counterpart generation, it is helpful to ensure that these proofs involve only small terms. To do this, the code transform algorithm generates a *factored* version of each function for which a symbolic counterpart is to be generated, replacing certain subterms of its translated body by calls of newly introduced subfunctions. In particular, each **if** term and lambda application in each function is replaced with an equivalent term that is a call of a newly-introduced function. In practice, this keeps the size of each generated function small enough that proofs involving these functions are feasible. After defining each such new function, the code transform algorithm proves that it is equal to the corresponding

subterm of the original function.

For **if** terms, factorization works as follows. Suppose the term in question is

$$\textbf{if } \mathit{test} \textbf{ then } \mathit{then} \textbf{ else } \mathit{else}.$$

The factorization algorithm first recurs on the three subterms, producing new terms test', then', and else', along with theorems showing that they are equal to the corresponding original subterms. Suppose that the collective free variables of *then* and *else* are $v_1, \ldots, v_n$, and that x is a variable that is not among these. (Factorization preserves the free variables of the term, so these are also the free variables of then' and else'.) The algorithm then defines a new function equivalent to the original **if** term. Taking AnonIfFn to be some symbol not previously defined as a function, the factorization routine defines

$$\textsc{AnonIfFn}(x, v_1, \ldots, v_n) = \textbf{if } x \textbf{ then } \mathit{then}' \textbf{ else } \mathit{else}'$$

and proves the theorem

$$\textsc{AnonIfFn}(x, v_1, \ldots, v_n) = \textbf{if } x \textbf{ then } \mathit{then} \textbf{ else } \mathit{else}.$$

Finally, the algorithm returns

$$\textsc{AnonIfFn}(\mathit{test}', v_1, \ldots, v_n),$$

which is equivalent to the original **if** term.

The factorization algorithm replaces lambda applications by a call of a new function that is defined as the body of the lambda. Suppose the term is

$$(\lambda v_1, \ldots, v_n \ .\ \mathit{body})(a_1, \ldots, a_n).$$

As with **if** terms, our factorization algorithm first recurs on each subterm, obtaining $body'$, $a_1', \ldots, a_n'$, and theorems showing their equivalence to the original subterms. It then generates a function to replace the lambda body; here suppose AnonLambdaFn is not a previously defined function and that $v_1, \ldots, v_n$ are the free variables of *body* (and therefore of $body'$). The algorithm defines

$$\textsc{AnonLambdaFn}(v_1, \ldots, v_n) = body'$$

and proves

$$\textsc{AnonLambdaFn}(v_1, \ldots, v_n) = body.$$

(There are no unbound variables in this function definition, since every lambda in a translated ACL2 term binds all free variables of its body.) Finally, the algorithm returns

$$\textsc{AnonLambdaFn}(a_1', \ldots, a_n'),$$

which is equivalent to the original lambda application.

The factorization algorithm introduces a top-level factored function that takes the same formals as the original function and whose body is the factored body of the original function. Given the theorems produced during factorization, it is trivial to prove this function equal to the original function.

The code transformation that creates symbolic counterparts operates on the factored functions. In topological order according to the call graph, the code transform creates symbolic versions of each factored subfunction. Since lambda applications are eliminated by factorization, the code transformation

needs to act only on the other three syntactic types of term. It operates as follows:

Variable. Unchanged.

Constant. Most constants are well-formed symbolic objects representing themselves, and are unchanged; other constants are replaced by a **Concr** object representation.

Function application. The code transform is recursively applied to the function's arguments. Then:

- If the function is not **if**, apply the function's symbolic counterpart to the transformed arguments and the current *pc* and *sdepth*.
- When the function is **if**, replace the **if** term by a term implementing the SExecIf algorithm of Section 2.2. The calls to SExec in that listing are each replaced by the appropriate transformed argument, under a rebinding of *pc* when necessary.

The transformation of the top-level factored function also inserts a preamble before the transformed body. First, this preamble fixes each symbolic argument to a well-formed symbolic object and *pc* to a well-formed uBDD. It uses ACL2's MBE (for "must be equal") facility [36] to do this without affecting execution speed: the symbolic counterpart's guard requires each argument to be well-formed; under this guard, the fixing operation is the identity, so when executing this function with guards satisfied, this operation may be

soundly skipped. Next, the preamble checks whether all symbolic arguments are general-concrete, and if so, calls the original function on their concrete values and returns this concrete result. Finally, the preamble addresses termination, checking whether the stack depth is exhausted; if so, an **App** object is returned representing the application of the original function to the symbolic arguments. Otherwise, the transformed body is called under a binding of *sdepth* to one less than the input *sdepth*.

Termination of the generated symbolic counterparts is proven by an argument about *sdepth*. For any recursive function or mutually-recursive clique, the symbolic counterparts of its factored subfunctions form a mutually recursive clique: the generated symbolic counterpart of the top-level factored functions, which include preambles, are considered to be the symbolic counterparts of the original functions, and calls of the original functions within the factored bodies are translated to calls of these symbolic counterparts. ACL2 requires that termination be proved by specifying an ϵ_0 ordinal-valued measure function for each function in a mutually-recursive clique. The proof obligation given such a set of measure functions is that for every recursive call in the clique, the caller's measure applied to its arguments is strictly greater than the callee's measure applied to its arguments [50]. The measure we choose for symbolic counterparts is $sdepth \cdot \omega + idx$. Here idx is a constant natural number chosen for each factored function; it is zero for top-level symbolic counterparts (those corresponding to the original functions of the mutually-recursive clique) and one plus the maximum idx of a callee for all other functions in the clique.

This ensures that for a call from a top-level function to a subfunction, the *idx* may increase but the *sdepth* decreases due to its binding in the preamble; and for any other recursive call, the *sdepth* is preserved and the *idx* decreases. In both cases, the ordinal measure decreases.

Once a symbolic counterpart function is defined, it remains to show that it obeys the guard, well-formedness, and correctness conventions listed in Section 2.3. (The arity convention is satisfied by construction.) While each of these proofs is conceptually straightforward, they must be managed carefully to ensure that they are fully automatic and their performance is acceptable. Factorization helps to reduce the size of terms that must be considered in each proof. Additionally, we select a minimal set of rewrite rules to make available to the prover and we give prover hints to control induction and function expansion.

The proofs that symbolic counterparts produce well-formed symbolic objects and that they obey the symbolic counterpart correctness constraints are straightforward for non-recursive symbolic counterparts, using the correctness and well-formedness conditions of each other symbolic counterpart called in the body. For (mutually) recursive symbolic counterparts, these proofs are completed by induction. In the typical case where the symbolic counterpart is mutually recursive, the induction scheme is generated as a "flag function" using the MakeFlag utility of Davis and the author [32].

The well-formedness and correctness conditions are not predicated upon the well-formedness of the inputs: a symbolic counterpart must produce a well-

formed and correct output regardless of the well-formedness of the inputs. This is supported because the preamble fixes the inputs to well-formed ones. This discipline makes the symbolic counterpart correctness and output well-formedness proofs faster by reducing the number of hypotheses in the rewrite rules that are applied during and produced by these proofs. The guard verifications that are necessitated by this discipline are themselves quite simple since they primarily rely on rewrite rules produced from the well-formedness proofs, which are hypothesis-free.

The reasoning for the well-formedness and correctness proofs primarily uses the rewrite rules generated by the well-formedness and correctness theorems of previously-introduced symbolic counterparts. A different approach is preferred for proving the type and correctness conjectures of transformed **if** terms. Each **if** of the original function is transformed into a complex structure of **if** terms and lambda applications that implements the SExecIf algorithm; the ACL2 rewriter will generally beta-reduce lambdas and case-split on **if** tests, causing our proofs to grow substantially. Instead, we use a strategy in which we prevent the rewriter from descending into this structure and use a meta rule [40] to reduce the type and correctness conjectures to the respective conjectures about the test and branch subterms. To prevent the rewriter from descending into the structure, we use ACL2's Hide function, an identity function that is specially recognized by ACL2's rewriter and prevents the rewriter from operating upon its argument.

Our efforts to make the introduction of symbolic counterparts and the

associated proofs automatic and fast have made it practical to introduce symbolic counterparts for moderately large programs; for proofs at Centaur Technology, we have generated symbolic counterparts for programs of around 1000 lines of code, implementing floating-point addition, among others. For this program, 39 new symbolic counterpart functions were introduced, incorporating 424 factored functions. On an Intel Xeon E5450 processor, it takes 37 seconds to introduce the new evaluator, factored functions, and symbolic counterparts and prove their guards, well-formedness, and correctness properties. However, in the next section we will discuss symbolic interpretation, which offers a way to perform symbolic execution of functions for which no symbolic counterparts have been generated.

2.5 Symbolic Interpretation

In addition to our code transform which generates symbolic counterparts, we have implemented symbolic interpretation as a second symbolic execution strategy. This approach avoids introducing symbolic counterparts and performing the associated guard and correctness proofs; newly introduced functions may immediately be symbolically interpreted without first generating symbolic counterparts and performing the associated proofs. In the course of proof development, it is common to repeatedly change certain function definitions, and repeatedly regenerating the symbolic counterparts for these functions often slows down the development process.

Interpreter-based symbolic execution is somewhat slower than execu-

tion using symbolic counterparts: while the speed of BDD operations and other symbolic object manipulation is the same between the two methods, each interpreted function call imposes some overhead. This slowdown is especially bad in cases where the bulk of the cost is from deep recursion and a relatively small portion is due to symbolic manipulation. The worst such case is when the interpreter is operating on concrete values, and therefore there is no symbolic manipulation. To alleviate this slowdown, GL allows new symbolic interpreters to be defined which are capable of directly calling a fixed set of functions on concrete arguments. Each new symbolic interpreter may also directly call a fixed set of symbolic counterparts, typically all those that were defined at the time of its definition. Thus symbolic interpreters are parametrized by two sets of functions: those that the interpreter may call on concrete arguments, and those for which the interpreter may call a symbolic counterpart.

A symbolic interpreter operates at the meta level: the main input is a quoted ACL2 term x, and the interpreter symbolically executes x on a binding b of the free variables of x to some symbolic inputs, yielding a symbolic object *res*. In stating and proving the correctness of a symbolic interpreter, we use a term evaluator [15, 40, 51] to compute the meanings of quoted terms. This allows us to integrate the interpreter into a verified clause processor [51], a procedure that can be called to relieve goals within ACL2 proofs; we describe this clause processor in Section 3.3. In the remainder of this section, we describe the interface of the interpreter, its correctness claim, and the algorithm

implementing symbolic interpretation.

2.5.1 Interface and Correctness Claim

We previously defined a symbolic execution of a (unary) function f as a computation that produces a symbolic object s', satisfying

$$\forall e \;.\; \langle s' \rangle(e) = f\left(\langle s \rangle\,(e)\right).$$

This generalizes trivially to a function of n inputs:

$$\forall e \;.\; \langle s' \rangle(e) = f\left(\langle s_1 \rangle\,(e), \ldots, \langle s_n \rangle\,(e)\right).$$

We wish to show that symbolic interpretation is a method of performing such a symbolic execution. That is, if symbolic interpretation of a term x yields a result *res*, we wish to claim that *res* satisfies the above condition. This requires a generalization of our view of symbolic execution since x is a data object (a quoted term) rather than a function. The semantics of such terms are given by a term evaluator $\textsc{Ev}(x, a)$, which produces the value of x given an assignment (association list) a of values to its free variables. If the free variables of x are $v_1, \ldots, v_n$, we then regard x as representing the function

$$f_x = \lambda i_1, \ldots, i_n \;.\; \textsc{Ev}\left(x, [(v_1, i_1), \ldots, (v_n, i_n)]\right).$$

The correctness claim for the symbolic interpreter is that its result *res* is a symbolic execution of f_x on the symbolic objects bound in the binding b. If b

has the form $[(v_1, s_1), \ldots, (v_n, s_n)]$, then the correctness claim is:

$$\begin{aligned}\forall e \;.\; \langle res \rangle(e) &= f_x\left(\langle s_1 \rangle(e), \ldots, \langle s_n \rangle(e)\right) \\ &= \textsc{Ev}\left(x, [(v_1, \langle s_1 \rangle(e)), \ldots, (v_n, \langle s_n \rangle(e))]\right).\end{aligned}$$

We use the notation $\langle b \rangle_{alist}(e)$ to denote symbolic object evaluation of the bound values in an association list such as b; the same correctness claim may then be expressed as

$$\forall e \;.\; \langle res \rangle(e) = \textsc{Ev}(x, \langle b \rangle_{alist}(e)).$$

Note, however, that symbolic interpretation is not guaranteed to produce a valid symbolic execution result in all cases; the correctness theorem proven of each symbolic interpreter has some hypotheses guarding the above correctness claim.

A symbolic interpreter INTERP takes a total of five arguments including the term x and symbolic bindings b, and returns three values including the symbolic execution result *res*. The arguments are $(x, b, pc, sdepth, world)$, with the following meanings:

- x: The term to be symbolically executed.
- b: The symbolic bindings; an association list giving a symbolic value for each free variable of x.
- *pc*: The current path condition of the symbolic execution.
- *sdepth*: The stack depth limit.

- *world*: ACL2's current database of axioms, definitions, and theorems.

The three return values are $(err, res, defs)$, described as follows:

- *err*: An error message, or *false* if there was no error.
- *res*: The result of the symbolic execution.
- *defs*: A collection of all definitions used during the symbolic interpretation.

The arguments *pc* and *sdepth* provide the same functionality as they do in symbolic counterparts: *pc* is a uBDD that tracks the control conditions under which the current path is executed, and *sdepth* forces termination. Unlike in symbolic counterpart definitions, however, when the stack depth is exhausted by the symbolic interpreter, an error is returned instead of a **App** object. The *world* argument is used for looking up definitions of functions. This allows an interpreter to symbolically execute functions that did not have symbolic counterparts or did not exist at the time the interpreter was defined: to symbolically execute a call of a such a function, the interpreter looks up its definition and recursively symbolically interprets the definition body. This implementation is described in the next subsection.

As mentioned above, a symbolic interpretation may encounter an error if the stack depth runs out or, additionally, if it encounters a function that is not defined. In such cases, a message describing the error condition is stored

in the *err* return value; if an error message is returned, the symbolic execution result *res* is considered invalid.

The final return value *defs* holds a collection of terms, representing all definitional equations obtained from *world* that were used during the symbolic execution. These definitions must be accurate in order for the symbolic execution to be correct. Typically, the *world* passed to an interpreter is ACL2's internally-maintained database and is known from an implementation perspective to contain correct definitions; however, there is no axiom reflecting this implementation reality, and additionally, an interpreter could be called with some other *world* containing incorrect definitions. Therefore, the interpreter records in *defs* each definition used during the course of each symbolic execution; the correctness of the result of that symbolic execution is then predicated on the correctness of these function definitions.

We are now ready to state the correctness theorem for symbolic interpreters. Essentially, the claim is that *res* is a symbolic execution of term x under symbolic bindings b as described above, provided:

1. the interpreter does not produce an error,
2. all definitions used are correct: that is, each definitional equation $eq \in$ *defs* evaluates to *true* under any assignment a,
3. the evaluation environment satisfies the path condition *pc*.

Theorem (Symbolic interpreter correctness claim)**.** *Let*

$$(err, res, defs) = \textsc{Interp}(x, b, pc, sdepth, world).$$

Then for any environment e*,*

$$\neg err \ \wedge \ \langle pc \rangle_{bdd}(e) \ \wedge \ \forall a \ . \bigwedge_{eq \in defs} \textsc{Ev}(eq, a) \\ \Rightarrow \langle res \rangle(e) = \textsc{Ev}(x, \langle b \rangle_{alist}(e)).$$

We prove this claim in ACL2 initially about a generic symbolic interpreter that does not recognize any functions, even the primitives for which symbolic counterparts are predefined. In order to create useful symbolic interpreters, we provide a utility that defines a new interpreter capable of calling any symbolic counterparts that are available at definition time, and a user-provided set of functions for concrete evaluation. This utility functionally instantiates the correctness theorem of the generic symbolic interpreter to show that this also holds of the new interpreter.

2.5.2 Implementation

A symbolic interpreter recursively walks over its input term x symbolically executing its subterms, then combines these results. As discussed previously, terms may be of four types: variables, constants, lambda applications, and function applications, of which **if** applications are a special case. An interpreter call $\textsc{Interp}(x, b, pc, sdepth, world)$ first checks that *sdepth* is not exhausted, returning an error message if it is, then splits into cases on the type of x, as follows:

Variable. Look up and return the binding of the variable name x in the symbolic object bindings b.

Constant. Return the constant value of x, wrapped in a **Concr** form if necessary.

Lambda application. Recursively interpret the terms given as actual parameters of the lambda application. Create a new association of the lambda's formals to the actuals, and interpret the lambda body under this set of bindings.

If. As in the SExecIf algorithm of Section 2.2, interpret the test term, then analyze its result to determine whether both or only one of the branches must be taken. Interpret the branches as necessary, then return the merged results.

Function application. Recursively symbolically interpret the terms given as actual parameters of the function application. Then:

1. If the function has a symbolic counterpart known to this interpreter, call that symbolic counterpart on the actuals.
2. Otherwise, if the results from the actuals are all concrete values and the function may be directly executed by this interpreter, call the function on the actuals.
3. Otherwise, look up the function's body and formals in *world*; if the function is not defined, return an error message. Add the function's

definitional equation to *defs*. Create a new association of the function's formals to the actuals, and symbolically interpret the body of the function under this set of bindings.

Our symbolic interpretation scheme allows us to flexibly balance the cost of automatically introducing symbolic counterparts with the interpreter overhead. In practice, we have found that interpretation overhead is almost never problematic except in cases where a function is interpreted on concrete inputs, so that there is no symbolic manipulation. In this case, it suffices to define an interpreter that has the problematic function among its set of concretely executable functions. Furthermore, usually the culprits in these cases are built-in ACL2 functions that are defined recursively in terms of the ACL2 primitives, but which have Common Lisp implementations that have different performance characteristics than their ACL2 definitions. For example, the function CHAR, which finds the character at a given index in a string, is defined in ACL2 as

$$\textsc{Char}(idx, str) = \textsc{Nth}(idx, \textsc{Coerce}(str, \textsc{list})).$$

This definition is linear-time, whereas Common Lisp implementations of this function use constant-time array indexing.

Chapter 3

Proof Automation

We obtain proofs by symbolic execution by following the steps outlined in Section 1.2: design a symbolic input vector that covers the set satisfying the hypotheses and symbolically execute the theorem's conclusion on these inputs. If the symbolic execution yields a constant-true result and the coverage claim can be proven, this suffices to prove that the hypotheses imply the conclusion. We have developed considerable automation to ease the process of proving theorems via these steps.

First, we implemented a procedure based on BDD parametrization which automatically narrows the coverage sets of symbolic inputs based on a hypothesis; we describe this in Section 3.1. This procedure makes it easier to construct a symbolic input vector that covers the inputs satisfying a hypothesis but that does not cover extraneous inputs that, since they do not satisfy the hypothesis, likely also do not satisfy the conclusion. For example, consider the theorem "all non-prime natural numbers between 2 and 16 are divisible by 2 or 3." It is relatively simple, in our symbolic object format, to manually create a symbolic object that covers the set of naturals less than 16; it is much more involved to create one that covers only the non-primes greater

than 1. But symbolically executing the conclusion "x is divisible by 2 or 3" on the former will yield the counterexamples 1, 5, 7, 11, and 13. A simple solution is to rephrase the theorem as "for all natural numbers x less than 16, x is prime, or is less than 2, or is divisible by 2 or 3." However, it is often the case that it is slower to symbolically execute a function on inputs covering a large set than a small one. Our parametrization procedure solves this problem by automatically narrowing a symbolic object covering more inputs than necessary to one that only covers those satisfying the hypothesis.

In Section 3.2 we describe a methodology, including supporting lemmas and proof automation techniques, that reduces the user effort required to prove that a symbolic input vector truly covers all inputs satisfying the hypotheses. While such coverage proofs are in general undecidable, in practice our methodology completes them automatically with at most a few simple hints from the user.

Finally, we have implemented a verified clause processor that automates the process of introducing theorems by symbolic execution, as we will describe in Section 3.3. Our clause processor supports both the basic steps described in Section 1.2 as well as a case-splitting mode in which the top-level theorem is proven using several symbolic executions which each cover a subset of the relevant input space. The clause processor performs the symbolic executions and obtains the required side conditions to complete the proof.

3.1 Symbolic Input Generation

To prove a theorem by symbolic execution following the method described in Section 1.2, one must provide a symbolic object for each free variable of the theorem. These symbolic objects must cover every possible setting of these free variables satisfying the hypotheses of the theorem. For performance, it is usually best to construct these objects so that their coverage set is as small as possible while meeting this requirement.

Constructing symbolic objects with a specific coverage set by hand is a challenging and error-prone process. For example, suppose the hypotheses to a theorem are as follows:

$$a, b, c \in \mathbb{N}, \quad a < 64, \quad b < 64, \quad c < 16, \quad -c < a - b < c. \tag{3.1}$$

It is easy to create a symbolic input vector that covers the set satisfying these hypotheses. It suffices that a, b, and c be symbolic integers of the appropriate bit lengths with each bit an independent BDD variable. It is also easy to restrict each two's-complement sign bit to *false* so that the coverage set includes only nonnegative integers. However, it is much more difficult to restrict the coverage set to only those input vectors satisfying $-c < a - b < c$. Constructing the component BDDs for the bits of these numbers would require a great deal of effort without algorithmic support.

3.1.1 Support for Narrowing of Coverage

BDD parametrization [5, 26] offers a method of automatically generating symbolic inputs with coverage restricted to those values which satisfy a particular predicate. BDD parametrization effectively substitutes specially constructed expressions for the variables of the Boolean formula represented by each BDD. These expressions have the property that their possible values correspond to exactly those settings of the variables that cause the parametrization predicate to be satisfied. There may be many possible such parametrizing substitutions for a given predicate; the BDD parametrization algorithm finds one such substitution. For example, suppose the predicate is $v_1 \oplus (v_2 \vee v_3)$. One possible parametrizing substitution would be

$$v_1 \leftarrow \neg(w_1 \vee w_2)$$

$$v_2 \leftarrow w_1$$

$$v_3 \leftarrow w_2.$$

This is an acceptable parametrizing substitution because the substitution applied to the predicate yields $\neg(w_1 \vee w_2) \oplus (w_1 \vee w_2)$, which is a tautology, and of the eight possible settings of v_1, v_2, v_3, the four that satisfy the predicate are achievable by some setting of w_1, w_2. In general, a parametrizing substitution for a predicate p is of the form

$$v_1 \leftarrow f_1^p(w_1, \ldots, w_m)$$

$$\vdots$$

$$v_n \leftarrow f_n^p(w_1, \ldots, w_m)$$

which we abbreviate using vector notation:

$$\vec{v} \leftarrow \vec{f^p}(\vec{w}).$$

It is generally desirable that m be smaller than n when possible, since having fewer variables speeds up BDD computations. The parametrization procedure we use does not always minimize m, but it never produces a substitution in which m exceeds n.

The substitution must satisfy the following two conditions with respect to the parametrization predicate p; we consider these to be the definition of a parametrizing substitution.

- **Precision.** For all settings of the parametrized variables $\vec{w}$, the predicate holds of the parametrizing expressions:

  $$\forall \vec{w} \; . \; p(\vec{f^p}(\vec{w})). \tag{3.2}$$

  Because the parametrizing expressions never produce extraneous values that do not satisfy the predicate, we say they are precise.

- **Generality.** For every setting of the original variables $\vec{v}$ satisfying the predicate, there is a setting of the parametrized variables such that the parametrizing expressions equal these values:

  $$\forall \vec{v} \; . \; p(\vec{v}) \Rightarrow \exists \vec{w} \; . \vec{f^p}(\vec{w}) = \vec{v}. \tag{3.3}$$

  Because the parametrizing expressions may produce any value satisfying the predicate, we say they are general.

Our BDD parametrization procedure is adapted from the *Cnst* algorithm of Coudert and Madre [26] to apply to uBDDs. The algorithm applies the parametrization of a predicate p to an individual BDD x, effectively applying the parametrizing substitution without computing it explicitly. (The substitution may be computed by applying the algorithm to each BDD variable.) Our algorithm, PARAMETRIZE(p, x), is shown in Algorithm 4. Here the predicate p, which must not be constant-*false*, is represented as a uBDD, and x is the uBDD to which we are applying the parametrizing substitution. In the uBDD representation, the depth within the tree implicitly indicates the variable of each node; each non-constant node only contains "then" and "else" subtrees. The constructor MK-UBDD(a, b) creates a node in which a is the THEN and b is the ELSE branch. In this algorithm, for the cases where p and x are non-constant, the variable of the top node of both uBDDs is implicitly the same.

Algorithm 4

```
1:  function PARAMETRIZE(p, x)
2:      if p = false then
3:          return error
4:      else if p = true or x is constant then
5:          return x
6:      else if THEN(p) = false then
7:          return PARAMETRIZE(ELSE(p), ELSE(x))
8:      else if ELSE(p) = false then
9:          return PARAMETRIZE(THEN(p), THEN(x))
10:     else
11:         a ← PARAMETRIZE(THEN(p), THEN(x))
12:         b ← PARAMETRIZE(ELSE(p), ELSE(x))
13:         return MK-UBDD(a, b)
```

The critical property of this algorithm is that its result represents the composition of the substitution with the input BDD:

$$\big\langle \textsc{Parametrize}(p, x) \big\rangle_{bdd}(\vec{y}) = \langle x \rangle_{bdd}(\vec{f^p}(\vec{y})). \tag{3.4}$$

The particular substitution that is used is not important for our purposes, aside from the fact that it obeys the precision and generality conditions listed above. However, it can be computed by applying Parametrize to BDD variables v_i:

$$f_i^p = \big\langle \textsc{Parametrize}(p, v_i) \big\rangle_{bdd}.$$

An important consideration about the parametrization process is the sensitivity of BDDs to variable ordering. While parametrization by a complicated predicate may disrupt the variable ordering chosen, in practice the algorithm tends to preserve the ordering relationship among the variables.

Suppose s is a symbolic object that covers a superset of the values recognized by a hypothesis $\textsc{Hyp}(x)$. We wish to obtain a new symbolic object s' which covers only those values satisfying $\textsc{Hyp}(x)$. To do this, we first symbolically execute Hyp on s. This yields a symbolic object $s_p = \textsc{Hyp}_{sym}(s)$ whose truth value is the predicate that we will use for parametrization. We extract a BDD representing this truth value using the Gtests function of Section 2.2. Recall from Section 2.2 that Gtests produces two BDD values k, v, giving the conditions under which the truth value of the input can be determined, and the conditions under which it is true. Let

$$(k_p, v_p) = \textsc{Gtests}(s_p, \mathtt{T}).$$

If s_p contains **App** or **Var** forms, then its truth value cannot be fully determined and k_p will be falsifiable. Some of these unknown values may reflect cases where the hypothesis is satisfied; therefore, we include these unknown values in our parametrization predicate. Specifically, we assign

$$p = v_p \vee^* \neg^* k_p.$$

We then parametrize each BDD in the representation of s using this predicate, yielding a new symbolic object s'; the function performing this transformation is called Parametrize-Gobj.

The following two theorems express the result of this transformation, stated in terms of the variables introduced above. Theorem 3.1.1 proves that if s covers the set recognized by Hyp, then so does

$$s' = \textsc{Parametrize-Gobj}(p, s);$$

this will be key to our coverage proof strategy in Section 3.2. Theorem 3.1.2 is not important for coverage proofs or any soundness claim, but assures us that, at least in the case where the symbolic execution of Hyp produced a known result, s' covers exactly the desired input set.

Theorem 3.1.1. *For all environments e satisfying* $\textsc{Hyp}(\langle s \rangle(e))$*, there exists e' for which $\langle s \rangle(e) = \langle s' \rangle(e')$.*

Theorem 3.1.2. *If $k_p = \mathtt{T}$, then* $\textsc{Hyp}(\langle s' \rangle(e))$ *holds of all environments e.*

To prove these theorems, recall that each BDD b' in the representation of s' is the parametrization by p of the corresponding BDD b in the representation of s. So for any environment e, by the correctness of PARAMETRIZE,

$$\begin{aligned}\langle b' \rangle_{bdd}(e) &= \big\langle \textsc{Parametrize}(p, b) \big\rangle_{bdd}(e) \\ &= \langle b \rangle_{bdd}(\vec{f}^p(e)) \qquad \text{(by Equation 3.4)}\end{aligned}$$

and therefore

$$\langle s' \rangle(e) = \langle s \rangle(\vec{f}^p(e)). \tag{3.5}$$

Here, as a notational convenience, we leave implicit the non-BDD variable bindings (supporting the **Var** form) in symbolic object environments. When we apply such an environment to a BDD, we mean to strip these bindings out, and when we apply an operator such as $\vec{f}^p$ we mean for it to simply preserve these bindings.

For the proof of Theorem 3.1.1, note that $\textsc{Hyp}(\langle s \rangle(e))$ implies $\langle p \rangle_{bdd}(e)$; this follows from Theorem 2.2.1 regarding GTESTS and the definition of symbolic execution, Equation 1.1. Therefore, using the generality property of the parametrizing substitution (Equation 3.3), there exists e' for which $\vec{f}^p(e') = e$. Then, using Equation 3.5,

$$\langle s' \rangle(e') = \langle s \rangle(\vec{f}^p(e')) = \langle s \rangle(e). \quad \square$$

We have proved both of these theorems in ACL2; a sketch of their proofs follows. For Theorem 3.1.2, by the precision property of the parametrizing substitution, we have $\langle p \rangle_{bdd}(\vec{f}^p(e))$, and since $k_p = \texttt{T}$, $\langle v_p \rangle_{bdd}(\vec{f}^p(e))$.

By the correctness of GTESTS and the symbolic execution, this implies that $\text{HYP}(\langle s \rangle(\vec{f}^p(e)))$, and therefore $\text{HYP}(\langle s' \rangle(e))$. $\square$

Recall the example of Equation 3.1, in which a, b, c are hypothesized to be natural numbers of a certain bit-length, but additionally restricted by the condition $-c < a - b < c$. Using our parametrization scheme, a user could simply provide symbolic objects covering the full possible range of each variable. By symbolically executing the hypothesis on these inputs and then parametrizing them using the resulting predicate, a new set of symbolic objects corresponding to a, b, and c is created which covers only the values allowed by the hypothesis.

3.1.2 Support for Case Splitting

In addition to narrowing a symbolic input to one of minimal or near-minimal coverage, parametrization may also be used to split a proof by symbolic execution into subcases. In some cases, it takes less time and memory to run multiple symbolic executions on small coverage sets whose union is the complete set of interest than to run a single symbolic execution covering the whole set. Of course, the reverse is also often true; otherwise, exhaustive concrete simulation would be preferable to symbolic execution.

Our system provides automation for case-splitting and allows the user a high degree of flexibility in specifying the cases and the symbolic inputs to be used in each case. Because different cases may benefit from different BDD orderings, our system allows the original, unparametrized symbolic input

vector to be chosen individually for each case to be considered. The case split is specified by an additional hypothesis that restricts coverage to a subset of the full input space; we symbolically execute the conjunction of the top-level hypothesis and the case hypothesis in order to obtain the parametrization predicate. The theorem's conclusion is symbolically executed separately on the parametrized symbolic inputs resulting from each case, and each case generates a corresponding coverage obligation.

When using case-splitting, a basic soundness requirement is that the cases considered must cover the space recognized by the top-level hypothesis. That is, it must be proven that for every input vector satisfying the top-level hypothesis, the additional hypotheses of at least one of the cases must be satisfied. This proof may itself usually be completed by symbolic execution without case-splitting. Full automation for proof by case splitting, including this additional proof, is built into the GL clause processor, as described in Section 3.3.

3.2 Coverage Proofs

Proof of coverage is a requirement for any proof using symbolic execution; a symbolic execution produces no information regarding concrete inputs that are not in its coverage set. The coverage theorem states that if an input vector is of interest in the top-level theorem (i.e., it satisfies the hypotheses), then it is in the coverage set of the symbolic inputs to be used in the symbolic execution. For the case of a single variable and a choice **s** of the symbolic

input object, the necessary theorem takes the form of Equation 1.3:

$$\forall x \;.\; \textsc{Hyp}(x) \Rightarrow \exists e \;.\; x = \langle \mathbf{s} \rangle(e).$$

Proof of coverage is undecidable, in general, due to the fact that the hypothesis may be arbitrarily complicated. In fact, any ACL2 theorem may be phrased as a coverage requirement with an appropriate choice of the hypothesis function and symbolic object. Given any conjecture $\mathrm{F}(x)$, let $\mathbf{s}$ be the constant `NIL`, and define

$$\textsc{Hyp}(x) = \neg(\mathrm{F}(\texttt{NIL}) \wedge \mathrm{F}(x)).$$

The coverage conjecture then reduces to

$$\forall x \;.\; \big((\mathrm{F}(\texttt{NIL}) \wedge \mathrm{F}(x)) \vee x = \texttt{NIL}\big)$$

which, given the existence of non-`NIL` objects is:

$$\mathrm{F}(\texttt{NIL}) \wedge \forall x \;.\; (\mathrm{F}(x) \vee x = \texttt{NIL})$$

or equivalently

$$\forall x \;.\; \mathrm{F}(x).$$

A separate potential difficulty in coverage proofs, independent of the complexity of the hypotheses, is the possibility of interdependent BDDs present in the symbolic objects. To compute the joint coverage of several BDDs is, in general, a difficult computational problem. Even resolving the question of whether a single concrete value is covered is NP-complete; for example, any CNF satisfiability problem can be rephrased in polynomial time as the question

of whether a set of BDDs each encoding a clause may all be simultaneously true. Therefore, coverage proofs are, at least in principle, difficult on two fronts: undecidable if the hypotheses are complicated, and NP-complete if the symbolic input in question is complicated.

However, it is rarely the case that the user supplies an input vector for symbolic simulation without knowing what coverage is desired. In general, the user has a coverage set in mind and designs the symbolic inputs to fit it, aided by techniques such as the parametrization method discussed in Section 3.1. Furthermore, the hypotheses of theorems amenable to symbolic execution tend to include simple restrictions of the types and sizes of inputs; otherwise, it would be unclear to the user how to approach creating a suitable symbolic input.

To heuristically aid in proving these conjectures, we have developed a methodology in which coverage proofs may often be completed automatically, and usually require only a small amount of user input. This methodology makes use of BDD parametrization, depending in particular on Theorem 3.1.1, which states that if some symbolic input vector s covers the set recognized by a hypothesis HYP, then the result of parametrizing s using HYP is another symbolic input s' which also covers that set.

In our methodology, we begin with a symbolic input vector s that obeys certain restrictions that make its coverage easy to compute; s must cover the set recognized by the hypotheses, but it typically covers a superset. Applying parametrization to s yields a new input vector, s', whose coverage is smaller,

but by Theorem 3.1.1 is still sufficient; often, the coverage of s' is exactly the set recognized by the hypotheses.

The restrictions we place on the initial, unparametrized symbolic input vector are as follows.

1. Each BDD appearing in a **Bool** or **Num** object must be a unique BDD variable.

2. Each **Var** form must have a unique name.

3. No **App** forms are allowed.

The first restriction eliminates interdependencies between the BDDs present in the object, making each bit in the symbolic representation a free, independent variable. The second restriction similarly eliminates interdependencies between **Var** forms. The third restriction is necessary since there is no syntactic method available to determine the coverage of an **App** form.

3.2.1 Shape Specifiers

To enforce and clarify these restrictions, our user interface requires descriptions of the symbolic inputs in a format that is distinct from, albeit similar to, that of symbolic objects. We call objects of this format *shape specifiers*. The shape specifier format has the same syntactic types as the symbolic object format, except for **App**, which is not allowed. In the **Num** and **Bool** types, which in the symbolic object format require BDD fields,

these BDDs are replaced by natural numbers indicating the indices of BDD variables. A further restriction on valid shape specifiers is that the BDD variable indices and the **Var** names appearing throughout the shape specifier must be duplicate-free. To transform a shape specifier into the unparametrized symbolic input vector, we define a function SSpec-To-SObj that simply replaces each BDD variable index i with the corresponding BDD variable v_i.

We define the coverage set of a shape specifier as the coverage set of the symbolic object derived from that shape specifier using SSpec-To-SObj. Due to the unique-variable restriction, computing the coverage set of a shape specifier is much simpler than the NP-complete problem of computing the coverage set of an arbitrary symbolic object. In Algorithm 5 we define a function SSpec-Covers(s, x) that determines whether a given concrete value x is covered by the shape specifier s, as shown by its correctness theorem:

$$\forall x, s \;.\; \text{SSpec-Covers}(s, x) \Leftrightarrow \exists e \;.\; x = \langle \text{SSpec-To-SObj}(s) \rangle (e).$$

This allows us to restate coverage obligations without an existential quantifier:

$$\forall x \;.\; \text{Hyp}(x) \Rightarrow \text{SSpec-Covers}(s, x). \tag{3.6}$$

In the coverage proofs seen in symbolic executions, the shape specifier s is a constant given by the user, and because SSpec-Covers is a recursion over s it readily expands and reduces to a series of type and size requirements on components of x.

The definition of SSpec-Covers is listed as Algorithm 5. In the **Atom** and **Concr** cases, s only covers its single value, so x is only covered if it

Algorithm 5 Computing coverage of shape specifiers

```
1: function SSpec-Covers(s, x)
2:     if s is an Atom then
3:         return x = s
4:     else if s is a Concr then
5:         return x = Concr-Obj(s)
6:     else if s is a Bool then
7:         return Booleanp(x)
8:     else if s is a Num then
9:         return Numberp(x) ∧ NumSpec-Covers(Num-Bits(s), x)
10:    else if s is s Var then
11:        return true
12:    else if s is an Ite then
13:        test-true ← SSpec-Covers-Iff(Ite-Test(s), true)
14:        test-false ← SSpec-Covers-Iff(Ite-Test(s), false)
15:        then-cov ← SSpec-Covers(Ite-Then(s), x)
16:        else-cov ← SSpec-Covers(Ite-Else(s), x)
17:        return (test-true ∧ then-cov) ∨ (test-false ∧ else-cov)
18:    else                                              ▷ s is a Cons
19:        if Consp(x) then
20:            car-cov ← SSpec-Covers(Car(s), Car(x))
21:            cdr-cov ← SSpec-Covers(Cdr(s), Cdr(x))
22:            return car-cov ∧ cdr-cov
23:        else
24:            return false
```

equals that value. In the **Bool** case, either Boolean value may be covered by assigning it to the variable index of s. If s is a **Var**, then it covers every possible object. If s is a **Num**, we call an auxiliary function NumSpec-Covers which checks whether each component of x — the real and imaginary numerator and denominator — are within the range of the lists of indices in s. If s is an **Ite**, we first check the possible truth values of the test subfield, using the auxiliary function SSpec-Covers-Iff. This function is similar to SSpec-Covers, but checks only whether an object of the same truth value as x is covered. Then it must be the case that either the test may be *true* and the **then** branch covers x, or that the test may be *false* and the **else** branch covers x. Finally, if s is a **Cons** specifier, then x must be a cons and the corresponding components must recursively satisfy SSpec-Covers.

3.2.2 Proving Coverage with Shape Specifiers

We provide rewrite rules to open calls of SSpec-Covers and compute the coverage set of the supplied shape specifier. These rewrite rules do not support **Ite** forms or **Num** forms that may represent non-integers, since in the usage modes we have seen so far, these forms have not been used in inputs to symbolic executions. On the other hand, these rules give special support to lists of Booleans, which are used frequently. The supplied set of rewrite rules may be extended by the user. Examples of these rules are listed in Figure 3.1.

The following example illustrates the operation of these rewrite rules on a coverage obligation. Let Hyps(a, b, c) be the formula listed in Equation

$$\textsc{SSpec-Covers}(\mathbf{Num}(bits), x) \rightsquigarrow \textsc{Integerp}(x) \wedge -2^{|bits|-1} \leq x < 2^{|bits|-1} \quad \text{(SSpec-Covers-Int)}$$

$$\frac{s \text{ is a } \mathbf{Cons}}{\textsc{SSpec-Covers}(s, \textsc{Cons}(x_1, x_2)) \rightsquigarrow \textsc{SSpec-Covers}(\textsc{Car}(s), x_1) \wedge \textsc{SSpec-Covers}(\textsc{Cdr}(s), x_2)} \quad \text{(SSpec-Covers-Cons)}$$

$$\textsc{SSpec-Covers}(\mathbf{Bool}(b), x) \rightsquigarrow \textsc{Booleanp}(x) \quad \text{(SSpec-Covers-Bool)}$$

$$\frac{a \text{ is an } \mathbf{Atom}}{\textsc{SSpec-Covers}(a, x) \rightsquigarrow x = a} \quad \text{(SSpec-Covers-Atom)}$$

$$\textsc{SSpec-Covers}(\mathbf{Concr}(v), x) \rightsquigarrow x = v \quad \text{(SSpec-Covers-Concr)}$$

$$\frac{\textsc{List-Of-Bools}(s)}{\textsc{SSpec-Covers}(s, x) \rightsquigarrow \textsc{Boolean-Listp}(x) \wedge |x| = |s|} \quad \text{(SSpec-Covers-Bool-List)}$$

Figure 3.1: Rewrite rules for SSpec-Covers

3.1, and let us assign the following shape specifiers to a, b, and c, respectively:

$$s_a = \textsc{Num}([0, 1, 2, 3, 4, 5, 6, 7])$$
$$s_b = \textsc{Num}([8, 9, 10, 11, 12, 13, 14])$$
$$s_c = \textsc{Num}([15, 16, 17, 18, 19]).$$

The following coverage obligation, derived from Equation 3.6, is generated by this assignment and hypotheses: 3.1:

$$\forall a, b, c \;.\; \textsc{Hyps}(a, b, c) \Rightarrow \textsc{SSpec-Covers}([s_a, s_b, s_c], [a, b, c]).$$

Three applications of the rewrite rule SSpec-Covers-Cons reduce this to

$$\begin{aligned} \forall a, b, c \;.\; \textsc{Hyps}(a, b, c) \Rightarrow \quad & \textsc{SSpec-Covers}(s_a, a) \\ & \wedge \textsc{SSpec-Covers}(s_b, b) \\ & \wedge \textsc{SSpec-Covers}(s_c, c) \\ & \wedge \textsc{SSpec-Covers}(\texttt{NIL}, \texttt{NIL}). \end{aligned}$$

The final conjunct is eliminated using SSpec-Covers-Atom, and the others are reduced by SSpec-Covers-Int, yielding

$$\begin{aligned} \forall a, b, c \;.\; \textsc{Hyps}(a, b, c) \Rightarrow \quad & \textsc{Integerp}(a) \wedge -2^6 \leq a < 2^6 \\ & \wedge \textsc{Integerp}(b) \wedge -2^6 \leq b < 2^6 \\ & \wedge \textsc{Integerp}(c) \wedge -2^4 \leq c < 2^4. \end{aligned}$$

Each of these conjuncts is implied by the hypotheses, so the coverage obligation is met.

In addition to these rewrite rules, we supply a computed hint scheme that is designed to heuristically expand functions within the hypotheses in a well-organized manner. This scheme is based on the observation that a hypothesis is often a conjunction of predicates about separate components of the inputs. For example, often an input variable is required to be a tuple, each element of which must have a particular type. These computed hints focus on one component of the tuple at a time, expanding predicates that seem relevant to the proof obligation for that component and throwing out any hypotheses that do not concern that component to avoid slowing down the prover.

These hints and rewrite rules automate coverage proofs in many common cases. However, the user still may need to intervene in the proof depending on the nature of the symbolic inputs and the hypotheses. Our user interface supports such intervention. The user may provide additional rewrite rules to increase the reasoning capabilities used during the proof. It is also often the case that some subfunctions within the hypotheses are unnecessary for the coverage proof, and expanding these will only slow down the proof; we therefore allow the user to specify a list of functions that will not be expanded by our computed-hint mechanism. Finally, in the worst case, the user may disable our built-in hints for coverage proofs and replace them with customized ones.

3.3 Clause Processor

We provide automation for proof by symbolic execution using verified clause processors [51], which are custom proof procedures for ACL2 that are proven (via an ACL2 proof) to be sound. In this section we first give a high-level description of clause processors and how they are verified, then describe the clause processors that we provide in support of proof by symbolic execution.

3.3.1 Verified Clause Processors

A clause processor is a custom proof procedure for ACL2 that operates on the level of proof obligations (goals). Each such obligation is represented as a *clause*, a list of terms whose disjunction is the goal to be established. A clause processor takes a clause as input and produces a list of clauses as output. ACL2 uses a clause processor by applying it to a prover goal, creating a new prover subgoal for each output clause returned. If each such subgoal is proved, this is assumed to be sufficient to prove the original subgoal.

A clause processor may be *verified* by an ACL2 proof showing that whenever each output clause is a theorem, the input clause is also a theorem. Once a clause processor is verified, ACL2 may use it to simplify proof goals without introducing unsoundness. The correctness claim for a clause processor verification may be stated using Ev to denote the function that, given an ACL2 term and an assignment of values to its free variables, produces the value of the term under that assignment. "Clause c is a theorem" may then be phrased as

$\forall a$. Ev(Disjoin(c), a), where Disjoin creates a term that is the disjunction of the members of a clause. The clause processor correctness claim is then as follows:

$$\begin{aligned} \forall c, h \;.\; \big(\forall a', c' \;.\; c' \in f(c,h) \Rightarrow \textsc{Ev}(\textsc{Disjoin}(c'), a')\big) \\ \Rightarrow \forall a \;.\; \textsc{Ev}(\textsc{Disjoin}(c), a). \end{aligned} \tag{3.7}$$

Here f is the clause processor function in question, c and c' are understood as clauses, a and a' are understood as association lists giving value assignments for the variables of the clauses, and h is an arbitrary input "hint" for f.

This correctness claim cannot be stated in the first-order ACL2 logic because the function Ev cannot be defined. However, one can instead state and prove a first-order theorem that implies this correctness claim by replacing the fully general evaluator with one that has defined behavior on only a fixed set of functions [15, 51]. Intuitively, every fact that is known about this evaluator is also true of the universal evaluator; therefore, if one can prove the correctness theorem about the limited evaluator, then it must hold in general. Therefore, to verify a clause processor in ACL2, one is required to prove a theorem similar to the claim above, but where Ev is replaced by such a partially-defined evaluator function and the inner quantifier is eliminated by Skolemization. The form of the theorem proved is as follows, where g may be any function of c, h, and a, and Conjoin-Clauses takes a list of clauses and returns a term representing the conjunction of the disjunctions of the terms of each clause:

$$\begin{aligned} \forall c, h, a \;.\; \textsc{Ev}\big(\textsc{Conjoin-Clauses}(f(c,h)), g(c,h,a)\big) \\ \Rightarrow \textsc{Ev}(\textsc{Disjoin}(c), a). \end{aligned} \tag{3.8}$$

A variation on this scheme is also allowed: the clause processor function may take the ACL2 state[1] as an additional argument and produce a triple of values consisting of a flag indicating an error, the result clauses, and the modified state; in this case, the same proof obligation is required to hold under the condition that there was no error.

3.3.2 GL Clause Processors

The clause processors used for proof by symbolic execution are paired with corresponding symbolic interpreters; recall from Section 2.5 that the user may define a new symbolic interpreter in order to support direct execution of a particular set of functions. A macro is provided that creates both a new symbolic interpreter and a verified clause processor that calls that interpreter to performs symbolic execution. These clause processors take the ACL2 state, primarily for access to function definitions, and may produce errors.

Each such clause processor has a basic mode of operation in which the proof is attempted with a single symbolic execution, and a case-splitting mode in which multiple symbolic executions are used to cover the space of possibilities in pieces. We first describe the simpler, non-case-splitting mode.

[1]The ACL2 state is a special object that allows access to, among other things, the ACL2 logical world, a structure which records all of the function definitions and theorems currently known to the logic. The state also contains a *globals table* for data storage, and additionally provides access to functionality such as file system access and random number generation, which cannot be soundly described in the logic as functions of typical objects. Instead, the logical story supposes that each access causes an irreversible change to the state, so that the same function may not be applied to the "same" state more than once.

3.3.2.1 Non-Case-Splitting Mode

In addition to the clause itself, the non-case-splitting mode requires additional arguments. First, an argument *bindings* assigns shape specifiers to each free variable of the theorem. These are used as a basis for generating symbolic inputs through the parametrization process described in Section 3.1. Next, arguments *hypothesis* and *conclusion* describe the division of the clause into hypothesis and conclusion. These are necessary because in clausal form, the terms forming these portions of the theorem are indistinguishable, but they play very different roles in proof by symbolic execution. The clause processor uses the given hypothesis and conclusion terms, rather than the clause itself, as the basis for the symbolic execution. To justify this, it creates a proof obligation stating that if the hypothesis implies the conclusion, then the clause itself holds. Since the hypothesis and conclusion should be a simple division of the clause into two parts, relieving this proof obligation is usually a matter of simple tautology checking.

The clause processor checks several error conditions. The hypothesis and conclusion terms must be syntactically well-formed terms. The bindings must contain a shape specifier for each free variable of the theorem, and the shape specifiers must be well-formed, with no duplicate BDD variable indices or **Var** names among them; this is required for our coverage methodology as discussed in Section 3.2.

After checking for these error conditions, the clause processor performs the parametrization procedure of Section 3.1. Assuming that the user-provided

shape specifiers cover the set of inputs satisfying the hypothesis, this process creates symbolic inputs s' that more narrowly cover this set; typically s' covers only and exactly those inputs satisfying the hypothesis. The conclusion is then symbolically executed on the generated symbolic inputs s', yielding a symbolic result $s_{res} = \textsc{Concl}(s')$. To determine whether the result is always true, s_{res} is checked using GTESTS, yielding $(k_{res}, v_{res}) = \textsc{Gtests}(s_{res}, \mathtt{T})$. Then,

- If the k_{res} and v_{res} BDDs produced by GTESTS are both constant *true*, then we conclude that, for covered inputs, the theorem's conclusion is always true.

- If $k_{res} \wedge^* \neg^* v_{res}$ is satisfiable, then we have a definite counterexample; the clause processor may generate several satisfying assignments from this BDD and evaluate the generated symbolic inputs under these assignments in order to display these counterexamples to the user.

- Otherwise, the clause processor performs a similar procedure for $\neg^* k_{res}$. The resulting assignments may or may not be true counterexamples, but hopefully will reveal to the user why the symbolic execution yielded a result containing **App** or **Var** forms.

3.3.2.2 Case-splitting Mode

When case splitting is desired, the clause processor requires additional input from the user to characterize the set of subcases to be considered and, for each subcase, the input object shapes and BDD orderings to be used in

the symbolic execution. To characterize the subset of the input space to be considered in each subcase, the user provides an additional case hypothesis that does not appear in the final theorem. The case hypothesis involves additional variables that are not present in the theorem; we will refer to these as case parameters. At a given setting of the case parameters, the case hypothesis restricts the values of the free variables of the theorem to some subset of their possible values. To specify the case-split to be used, the user provides a list of assignments of concrete values to the case parameters; the case hypothesis under each assignment determines that subcase's restriction on the theorem's variables.

For illustration, suppose our theorem's free variable was an integer x, and that we wished to split into cases based on whether x is even or odd. To specify this case split, one might supply the case hypothesis

$$\textbf{if } \textit{parity} \textbf{ then } \textsc{Oddp}(x) \textbf{ else } \textsc{Evenp}(x),$$

where *parity* is a case parameter, and the two assignments *parity* = `T` and *parity* = `NIL`.

In proofs with case-splitting, it is often important for performance to use a different BDD ordering for each subcase. Therefore, each case parameter assignment is accompanied by a binding list of the theorem variables to shape specifiers, to be used for generating the symbolic inputs for that subcase. Each subcase therefore incurs its own coverage proof obligation: the provided shape specifiers must be shown to cover the space recognized by the conjunction of

the main theorem hypothesis and the case hypothesis with the given parameter assignment.

Case-splitting additionally incurs an obligation to show that the subcases are sufficient to cover the entire input space recognized by the theorem's hypothesis. Specifically, if the theorem's hypothesis is satisfied by an assignment of values to the theorem's free variables, then for at least one of the listed case parameter assignments, the case hypothesis is also satisfied. This may itself be proved by symbolic execution; the user provides an additional set of shape specifier bindings specifically for this proof obligation. These shape specifiers must in turn be shown to cover the entire input space recognized by the main hypothesis.

Chapter 4

Hardware Modeling

The GL framework is designed to provide symbolic execution capabilities for arbitrary ACL2 functions, but it is particularly useful in hardware verification. In such applications, typically a property or specification is written as an ACL2 function, and the goal is to prove that a hardware model satisfies this property or meets this specification. There are many possible ways to model hardware. In this chapter we describe an approach to hardware modeling that has been used at Centaur Technology for several successful verifications; a case study of one such verification is described in Chapter 5.

We model hardware in the E hardware description language [13], a formally defined HDL that is deeply embedded [57] in ACL2. Each module of the hardware design is represented as an ACL2 constant object called an E module. The operation of these modules may be simulated by an ACL2-based HDL simulator called Emod. This simulator may be used to run concrete test cases, and may also symbolically simulate the modules in order to extract Boolean formulas representing the circuit logic of blocks of interest.

In our verification process, we receive the design as Verilog RTL files [1]. We mechanically translate these Verilog sources to E modules [44]; this trans-

lation step is unverified. We can compare our E translations to the original Verilog by running tests on our translations using EMOD and on the Verilog sources using commercial Verilog simulators. To perform proofs about a translated hardware module, we use EMOD to symbolically simulate the module using and/inverter graphs (AIGs) [56] as the Boolean function representation. This yields a set of AIGs[1] giving the outputs and next-state functions of the hardware module in terms of the inputs and previous states. We have proven that evaluating these AIGs produces results that are equal to those produced by concrete simulation with EMOD.

Figure 4.1 shows the progression of models in this verification process. Here the dotted arrows indicate steps that are unverified or extralogical, and solid arrows indicate steps that we have verified with respect to one another.

In this chapter we will describe our strategy for proving theorems about our hardware models. In Section 4.1 we discuss the E hardware description language and the EMOD simulator. Then in Section 4.2 we will describe the AIG-based method by which we symbolically execute our hardware models inside GL proofs.

[1] In most AIG implementations, all AIG nodes are stored in one network that may have many output and/or latch nodes. In our ACL2 implementation, each AIG is a rooted tree where the root node is the output value. Therefore, we tend to speak of multiple AIGs rather than a single AIG with multiple outputs. The common structures of all AIGs are shared via structural hashing in both cases.

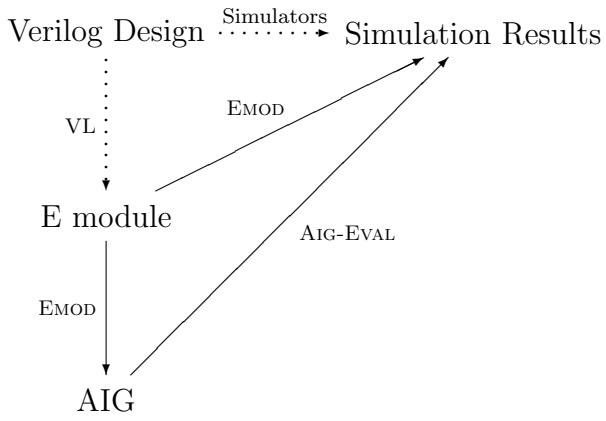

Figure 4.1: Hardware Modeling and Simulation

4.1 The E Hardware Description Language

The E hardware description language, developed by Boyer and Hunt [13, 43], is an occurrence-based, hierarchical HDL that is deeply embedded in the ACL2 logic. It is related to other formal hardware description languages such as HEVAL [17], DUAL-EVAL [18], and DE2 [42]. Each hardware unit modeled in the E language is a data object, called an E module, defined as a constant in ACL2. The semantics of these hardware objects is given by an interpreter function, EMOD, written in ACL2.

The EMOD interpreter runs in several modes. It can perform symbolic simulations in either two- or four-valued logics (four-valued simulations are necessary to model constructs such as bidirectional buses and tristate drivers). In these symbolic simulations, one may use either BDDs or AIGs as the Boolean function representation. A concrete simulation is performed by running a symbolic simulation with concrete input and state values. EMOD

can also perform conservative dependency analysis and delay estimation.

The E language is a hierarchical, gate-level language. Each module is either a predefined primitive, such as a basic single-bit gate or latch, or a hierarchical module containing a netlist of submodules. The semantics of primitive modules are built into EMOD, whereas hierarchical modules are simulated by recursively simulating their submodules.

The submodules within each hierarchical module are listed in a certain order that may include duplicates. To simulate a hierarchical module, EMOD recursively simulates its submodules in the given order. At each submodule occurrence m in this order, EMOD extracts the inputs of m from the current settings of the signals, and recursively simulates m on those inputs. The signal settings are then updated by setting the signals connected to the outputs of m to the values produced by the recursive simulation. In order for this simulation to yield realistic results, the occurrences should be ordered so that the final duplication of each occurrence occurs after its inputs have settled to their final values.

The E language has a phase-based timing model. A module is simulated with a given setting of its input signals, and it is assumed that these input values are held long enough for all wires in the circuit to settle to new steady-state values. An EMOD simulation operates on an input and initial state vector. It produces an output and final state vector, signifying the values produced by the module outputs and stored in the state-holding elements,

```
(defm *half-adder*
  '(:i (a b)
    :o (sum carry)
    :occs
    ((:u o0 :o (sum)   :op *xor2* :i (a b))
     (:u o1 :o (carry) :op *and2* :i (a b)))))

(defm  *counter*
  (:i (c-in reset-)
    :o (out c)
    :occs
    ((:u o2 :o out         :op *ff* :i (sum-reset))
     (:u o0 :o (sum c)     :op *half-adder* :i (c-in out))
     (:u o1 :o (sum-reset) :op *and2* :i (sum reset-))
     (:u o2 :o out         :op *ff* :i (sum-reset)))))
```

Figure 4.2: EMOD examples

respectively, after each wire has reached a new steady state.[2] There is no explicit unit of delay in this timing model. Thus, certain constructs may not be accurately modeled by E modules generated from the Verilog translator; however, the Centaur design is largely restricted to a subset of Verilog that may be modeled in the E language.

A simple example of the E language syntax is shown in Figure 4.2. In this example, we define a half-adder and a one-bit counter in terms of the primitives ***xor2***, ***and2***, and ***ff***.

Each module and occurrence definition consists of several fields labeled by keyword symbols `:i`, `:o`, `:occs`, etc. The list of occurrences is given by

[2]These "vectors" are not necessarily just lists; E allows input, output, and state vectors to be organized in more complex ways.

`:occs`, and the connectivity among the module occurrences is given by the `:i` (input) and `:o` (output) fields of the modules and occurrences. Each occurrence has a name given by its `:u` field. Note that the ***counter*** module's occurrence list contains a duplication of instance `o2`. The first occurrence is necessary to set its output signal `out` according to the stored state value, and the second is necessary in order to update the state with the newly calculated value `sum-reset`.

While simulations run using EMOD are symbolic in general, useful properties about hardware modules are most clearly expressed as statements about concrete-value simulations. For example, the correctness of the ***counter*** module might be stated as follows:

Theorem. *Let c_{in}, $\overline{reset}$, and st be Booleans, and let*

$$(nextst, out) = \text{EMOD}(\texttt{'TWO}, \textbf{\texttt{*counter*}}, \text{LIST}(c_{in}, \overline{reset}), \text{LIST}(st)).$$

Then

$$nextst = out = \overline{reset} \wedge (st \oplus c_{in}).$$

In this example, the `TWO` flag determines that EMOD will run a two-valued, BDD-based symbolic simulation; however, for any instance of this theorem in which the hypotheses are satisfied, the inputs to this EMOD simulation are concrete, Boolean values.

Symbolic simulation can be used to prove such theorems because we have shown that EMOD symbolic simulation is faithful to concrete simulation. The commutative diagram in Figure 4.3 illustrates the meaning of this

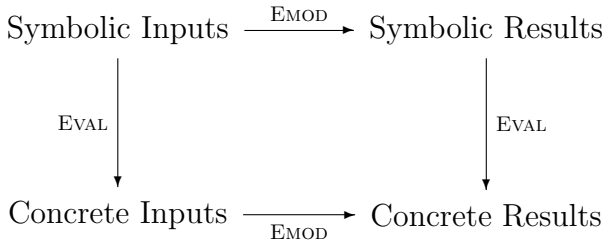

Figure 4.3: EMOD commutativity with evaluation

correspondence: the same concrete results may be obtained either by first evaluating some symbolic inputs and running a concrete simulation using EMOD, or by running a symbolic simulation using EMOD and subsequently evaluating the symbolic results. The following theorem, which has been proved in ACL2, states the correspondence for the two-valued AIG mode; similar theorems hold for the four-valued AIG and two/four-valued BDD modes.

Theorem 4.1.1 (EMOD commutes with AIG-EVAL)**.** *Let m be a well-formed module, i_{sym} a well-formed two-valued AIG input vector for m, s_{sym} a well-formed two-valued AIG state vector for m, and env an assignment of Boolean values to the variables of i and s, and let:*

$$i_{concr} = \text{AIG-EVAL-VEC}(i_{sym}, env),$$
$$s_{concr} = \text{AIG-EVAL-VEC}(s_{sym}, env),$$
$$(s'_{sym}, o_{sym}) = \text{EMOD}(\texttt{'AIG}, m, i_{sym}, s_{sym}),$$
$$(s'_{concr}, o_{concr}) = \text{EMOD}(\texttt{'AIG}, m, i_{concr}, s_{concr}).$$

Then

$$s'_{concr} = \text{AIG-EVAL-VEC}(s'_{sym}, env), \ \ and$$
$$o_{concr} = \text{AIG-EVAL-VEC}(o_{sym}, env).$$

Here the well-formedness conditions are syntactic checks on the module and the symbolic input/state vectors, requiring, for example, that they contain the right number of AIGs. AIG-EVAL is a function that determines the Boolean value of an AIG given an assignment *env* of Boolean values for each of its variables, and AIG-EVAL-VEC(v, env) is a function that applies AIG-EVAL(a, env) to each AIG a in the vector v.

These theorems can be used to define a GL symbolic counterpart for EMOD. A fully general symbolic counterpart for EMOD would allow symbolic executions in which the (GL object) symbolic inputs to EMOD themselves represented (BDD or AIG) symbolic values. Instead, we wish to run symbolic executions in which the (GL object) symbolic inputs to EMOD represent concrete (Boolean or four-valued) values. To do this, we define a custom symbolic counterpart for EMOD which uses EMOD itself as the underlying symbolic simulation engine. This symbolic counterpart checks that the mode flag and module inputs are concrete GL objects, the module is well-formed, and that the provided input and state vectors are symbolic (GL) objects that always represent well-formed, concrete (Boolean or four-valued) input/state vectors for the module. That is, while these may be non-concrete symbolic objects, each of their possible values is an input vector consisting only of constants,

rather than non-constant BDDs or AIGs. We extract from these GL symbolic objects a vector of BDDs, each representing the conditions under which the corresponding input bit is true or false, and we run a BDD-based EMOD symbolic simulation on these vectors. The result is an output and next-state vector of BDDs; we wrap each such BDD in a **Bool** object to create the output of the EMOD symbolic counterpart. Thus, our GL symbolic counterpart for EMOD uses EMOD itself as its own symbolic simulator.

A major disadvantage of this strategy is that BDD-based symbolic simulations may be extremely inefficient. It is often the case that many internal signals of the hardware module are irrelevant to the desired result; for example, many arithmetic-related units perform several operations in parallel but produce only the result of one of them, based on some data-dependent logic. In such cases, EMOD will symbolically simulate all of these operations even though it may be that only one is relevant to the desired result. This may be especially problematic when different internal signals of the hardware module require different BDD orderings to efficiently represent their values as a function of the inputs. These problems are remedied by the AIG-based methodology described in the following section.

4.2 Symbolic AIG Evaluation

In order to avoid computing BDDs for irrelevant internal signals of a hardware model, we perform EMOD symbolic simulations using AIGs and subsequently convert the resulting AIGs into BDDs. This is an effective strategy

because it is relatively inexpensive to build AIGs representing the outputs of a hardware module, and because it is possible to program algorithms that avoid irrelevant and overly expensive computations while operating on AIGs. In this section we will describe the algorithm we use to produce a BDD representation from a set of AIGs representing the results of a hardware simulation, and how this fits into our hardware verification methodology.

Suppose we wish to show that the result given by a specification function is equivalent to the result computed by a concrete EMOD simulation of a hardware model. As stated in Theorem 4.1.1, this EMOD simulation is equivalent to an evaluation of AIGs resulting from a sufficiently general symbolic EMOD simulation. We therefore may instead, equivalently, prove that the specification function is equivalent to that AIG evaluation. To prove this theorem using GL, we need a symbolic counterpart for our AIG evaluation function.

The AIG evaluation function, AIG-EVAL(x, *env*), is a simple recursive evaluator for AIGs, given as Algorithm 6; it computes the Boolean value of x, an AIG, given an environment *env* assigning Boolean values to each of the inputs of x. We use the ACL2(h) memoization capability [12] to memoize this function so that when it recurs on an AIG node whose value has been previously computed for the current environment, it returns the recorded value instead of recomputing it, thus ensuring that this function takes time linear in the size of the AIG.

For hardware-related theorems, we wish to symbolically execute AIG-EVAL. To do this, we use a hand-coded symbolic counterpart rather than

Algorithm 6 AIG evaluation

```
function AIG-EVAL(x, env)
    if x is Boolean then
        return x
    else if x is a variable then
        return LOOKUP(x, env)
    else if x is a negation then
        return ¬AIG-EVAL(x.child, env)
    else                                              ▷ x is an AND node
        return AIG-EVAL(x.child1, env) ∧ AIG-EVAL(x.child2, env)
```

relying on the symbolic interpreter or the automated symbolic counterpart generation. These automated methods perform poorly for two reasons. First, the symbolic evaluation of a particular AIG subgraph is recomputed at every visit to its root node; memoization avoids this problem in concrete executions of AIG-EVAL. Second, even if we were to memoize these computations, we would still face a problem similar to that of BDD-based symbolic simulation in EMOD: often the values of many internal nodes of the AIG are irrelevant to the final result, and computing their BDDs often causes blowups.

Our hand-coded symbolic counterpart for AIG-EVAL is tailored to the type of symbolic inputs we expect to occur in proving hardware-related theorems. The AIG input x itself will be a (GL) concrete object, the result of an EMOD symbolic simulation. In the environment *env*, the variables of x may be bound to (GL) symbolic Booleans. The result computed by our symbolic counterpart will also be a symbolic Boolean expressing the function composition of the AIG with the variable bindings. Because GL's symbolic Booleans are expressed using BDDs, we compute this by producing a BDD representing

Algorithm 7 AIG symbolic evaluation (naive algorithm)

```
function A2B(x, env)
    if x is Boolean then
        return x
    else if x is a variable then
        return LOOKUP(x, env)
    else if x is a negation then
        return ¬* A2B(x.child, env)
    else                                          ▷ x is an AND node
        a ← A2B(x.child1, env)
        b ← A2B(x.child2, env)
        return a ∧* b
```

the function composition of the AIG x with the BDDs bound in *env*.

The core algorithm used by our hand-coded symbolic counterpart is called AIG2BDD and computes the AIG/BDD function composition for the primary output node of the AIG [76]. A specification for this algorithm, called A2B, is described in Algorithm 7; this specification simply computes the AIG/BDD function composition for each node in the AIG. This specification is amenable to memoization, but it performs poorly in practice because it computes full BDD representations for many irrelevant AIG nodes. We describe certain cases where these irrelevant nodes occur and how the AIG2BDD algorithm identifies and prunes them.

As we mentioned previously, hardware modules are commonly designed to internally compute several different functions, but produce as output the result of only one of these functions based on an opcode or other control signals. In the AIG representations of such designs, the discarded function results occur

as subgraphs in the AIGs of the outputs. Often, applying constant propagation based on a setting of control signals for the operation in question will prune out the irrelevant functionality from the AIGs. This is a very simple step that could be applied to the AIGs as pre-processing before computing the full AIG/BDD function composition.

However, in some cases the multiplexing is data-dependent. For example, in some implementations of floating-point addition, two or more computations are performed in parallel, of which only one is expected to produce the correct result. The result from the appropriate computation is chosen based on the relative values of the signs and exponents of the two operands. In this case, even when the coverage of a symbolic execution is limited so that all covered inputs result in a particular path being chosen, constant propagation is not sufficient to remove the irrelevant portions; stronger Boolean reasoning is required. Furthermore, it is often prohibitively expensive to symbolically execute the irrelevant paths, since a BDD ordering that works well for one path may be inefficient for another.

We prune out these irrelevant AIG branches using methods similar to the dynamic weakening strategy of Seger et al [74] and the incremental symbolic simulation strategy of Paruthi et al [63]. Our AIG2BDD algorithm uses two distinct methods to prune irrelevant branches from the AIG. We call these the *substitution* and *bounding* methods. Both methods associate AIG nodes with approximate BDD representations conforming to a certain BDD size bound. These approximate representations show that certain branches

may be ignored, and are then iteratively tightened until an exact BDD result is obtained. Generally, bounding method is weaker but faster than the substitution method.

In an AIG AND node with two children a and b, b is irrelevant if it can be shown that $a \Rightarrow b$; in this case, the AND node may be replaced by a. (One common situation is that a is constant-*false*.) The Aig2Bdd algorithm successively applies the two approximation methods, each of which can detect certain such implications without necessarily computing a full BDD representation of a or b. In both methods, we begin at the leaves of the AIG and build toward the root, computing exact BDD translations for nodes until some node's translation exceeds a size limit. We replace the over-sized BDD with a new representation that loses some information but allows the computation to continue. After the AIG's output node is translated, we check to see whether its BDD result is exact. If so, we are done; otherwise, we increase the size limit and try again. During this translation process, we check each AND node for an irrelevant branch and remove such branches from the AIG so that it will be ignored in subsequent iterations. Heuristically, we use the (weaker) bounding method first with small size limits, then switch to the substitution method at a larger size limit.

In the bounding method, the translated value of each AIG node is two BDDs that are upper and lower bounds for the exact BDD function composition, in the sense that the lower-bound BDD implies the exact BDD and the exact BDD implies the upper-bound BDD. If the upper and lower bound

BDDs for a node are equal, then they both represent the exact BDD translation for the node. When a BDD larger than the size limit is produced, it is thrown away and the constant-$true$ and constant-$false$ BDDs are instead used for its upper and lower bounds. If an AND node $a \wedge b$ is encountered for which the upper bound for a implies the lower bound for b, then we have $a \Rightarrow b$; therefore we may replace the AND node with a. Thus, using the weak method we can, for example, replace an AIG representing $a \wedge (a \vee b)$ with a whenever the BDD translation of a is known exactly, without computing the exact translation for b.

In the substitution method, instead of approximating BDDs by an upper and lower bound, fresh BDD variables are introduced to replace over-sized BDDs. We ensure that these variables are not reused. The BDD associated with a node is its exact translation if it references only the variables used in the primary input assignments. This catches certain additional pruning opportunities that the weaker method would miss; for example, it can replace $b \neq (a \neq b)$ with a.

We have proven in ACL2 that the Aig2Bdd algorithm is correct. It may not always produce an exact BDD result if the maximum BDD size limit is set too low; however, when it signals that an exact result has been produced, that BDD result represents the function composition of the AIG and the BDD inputs. We also prove that this BDD result is equal to the result of the simpler A2B function [76].

We use the AIG2BDD algorithm as the core of our hand-coded symbolic counterpart for the AIGEVAL function. In this symbolic counterpart, we require the symbolic input AIG x to be a general-concrete object (as defined in Subsection 2.1.2, so that it has only one possible concrete value, which we may determine syntactically. We also require that the symbolic environment, *env*, be concrete in its representation except in its bound values: the keys must be of **Atom** type and the alist structure in which the values are bound must be composed of simple **Cons** constructs, without, for example, any **Ite**-type components. Furthermore, we must be able to extract a BDD representing the unambiguous truth value of each bound value using GTESTS; that is, the bound values may not contain **App** or **Var** forms. When the inputs meet these syntactic restrictions, we may extract a concrete AIG from the general-concrete input x and an alist mapping atoms to BDDs from the symbolic environment *env*; these are suitable inputs for AIG2BDD. Running AIG2BDD on these inputs results in a BDD that, when inserted into a **Bool** construct, is the correct symbolic result. If either of the syntactic requirements on the inputs are not met, we return an **App** object representing the call of AIGEVAL.

We prove the usual theorem (see Section 2.3) stating that this function satisfies the definition of a symbolic counterpart for AIG-EVAL. This proof makes use of the correctness of the AIG2BDD algorithm and also addresses the extraction of the AIG and association list inputs.

In our current hardware verification methodology, the theorems we produce concern evaluation of the AIGs resulting from a symbolic Emod simulation of a hardware model. A more clear and elegant theorem would instead state that some property holds of the Boolean results of all concrete Emod simulations of the hardware model. Theorem 4.1.1 shows that symbolic simulation is faithful to concrete simulation; therefore, our AIG-related theorems imply theorems of the second type. However, in practice it is tedious and confusing to complete these proofs. The steps in such a proof are similar to those described in Theorem 1.2.1, and we do not yet have a framework in place to automate these steps. The coverage step, in particular, should be automated to make this kind of proof practical.

As future work, we intend to explore approaches that ease the process of introducing theorems based on Emod concrete simulations. One such approach is to design a GL symbolic counterpart for Emod which performs an AIG-based symbolic simulation of the given module and then converts the results to BDDs in order to produce a GL-formatted symbolic result. However, this lacks the capability to skip the BDD conversion for results that will not be used; furthermore, when the theorem involves multiple phases of Emod simulations, such a symbolic counterpart would convert all the intermediate states and outputs to BDDs, even though some of them might not be relevant to the final results.

Chapter 5

Case Study: Floating-point Addition

We have used the GL framework at Centaur Technology to verify the correctness of several operations in the execution cluster of the VIA Nano™ processor, a commercial 64-bit x86 processor. The verified operations include many media unit instructions, several multiplications, and the main iteration of a divider. As a case study, this chapter describes the verification of the floating-point addition instructions of the media unit, and focuses on the role of the GL framework; other parts of the process are discussed in greater detail elsewhere [43, 44].

We would like to thank Terry Parks for several contributions to this work, including an ACL2 specification for floating-point addition and a Verilog parser written in Lisp.

5.1 Overview of Verification Methodology

The verification methodology used for the *fadd* unit is illustrated in Figure 5.1. The hardware model to be verified is based on Verilog files developed by logic designers. In Chapter 4, we described how Verilog RTL descriptions are captured as E modules and symbolically simulated. We use the four-valued

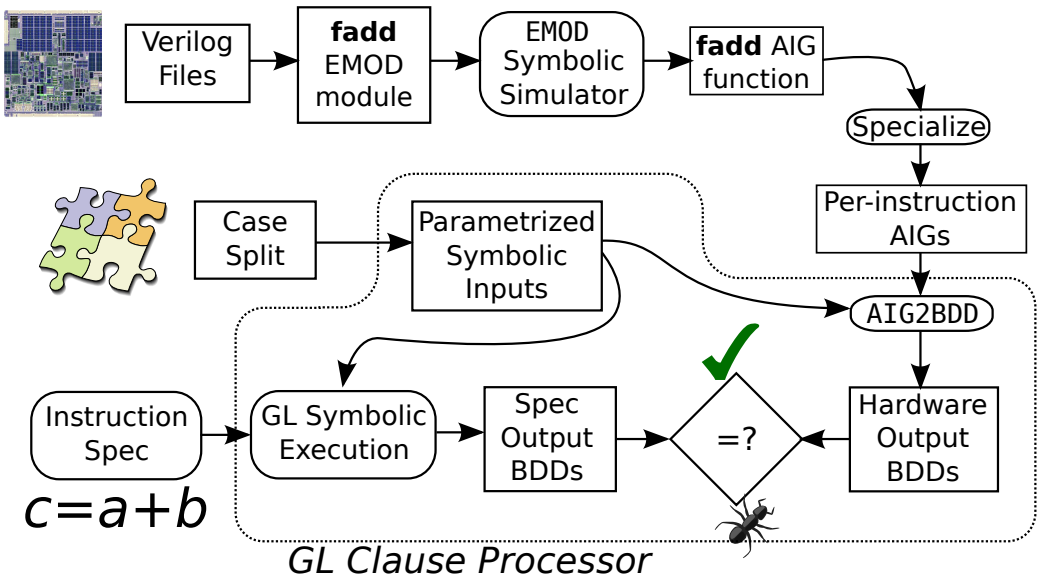

Figure 5.1: Centaur Verification Methodology

AIG-based symbolic simulation mode to produce formulas describing the outputs and next-states of our hardware modules as a function of the inputs and previous states. We initially perform a very general simulation that covers a large class of instructions. We then specialize the resulting AIGs by constant-propagating the control signals for a particular instruction, generating a set of AIGs specific to that instruction. In this constant-propagation process, we also assign X (unknown) values to all signals that we expect to be irrelevant to the operation of that instruction; this yields a conservative simplification of the resulting AIGs. We also restrict relevant inputs to symbolic Boolean (non-X) values, because we usually do not specify desired behavior of an instruction

when any of the relevant inputs are X.

We use GL to prove that the AIG formulas derived from the hardware simulation conform to some specification as described by an ACL2 function. For floating-point addition operations, we use the case-splitting feature of the GL clause processor to avoid BDD size explosions. Parametrization, AIG to BDD conversion, symbolic execution of the specification, and checking the implementation for compliance with the specification are handled by the GL clause processor. The result of this process is either a completed proof that the hardware model meets our specification, or a counterexample showing that it does not.

5.2 The *fadd* unit

Within the media unit of the Nano CPU, the *fadd* unit handles floating-point addition and subtraction. This unit is highly optimized for low latency arithmetic operations. It implements SIMD 32- and 64-bit floating-point additions as well as scalar x87 80-bit floating point additions and many of the other operations from the SSE and SSE2 extensions to the x86 instruction set, such as bitwise logical operations, shuffles, and min/max operations. All floating-point addition operations are performed with a two-cycle latency; the *fadd* unit can also forward results internally so that operations may be chained.

The *fadd* unit's RTL-level design is 33,700 lines of Verilog composing 680 modules. It has 374 output signals and 1074 inputs including 26 clocks; multiple clocks are used for power management. Its physical implementation

contains 432,322 transistors, about equally split between PMOS and NMOS devices.

The *fadd* unit contains four adders: two 32-bit units, one 64-bit unit, and one 80-bit unit. When a 32-bit packed addition is requested, all four units are used, and the 64-bit and 80-bit adders each take 32-bit operands and produce a 32-bit results. When a 64-bit packed addition is requested, the 64-bit and 80-bit adders each take 64-bit operands and produce a 64-bit result. The *fadd* unit can only add one pair 80-bit operands per clock cycle. Other combinations are possible when a memory-resident operand is added to a register-resident, X87-style, 80-bit operand; the *fadd* unit also manages such X87-mode, mixed-size addition requests.

For each addition, multiple datapaths through the addition logic operate in parallel, and for most sets of operands only one such path is expected to compute the correct result. The relevant path for a particular pair of operands is determined by characteristics such as the operand types (NaN, zero, denormal, etc.) and their sign and exponent bits, and the result from the correct path is selected by a multiplexor as the result of the addition.

5.3 Floating-point Addition Verification Process

To show that the floating-point addition instructions are correct, we must prove that the AIG pair giving the four-valued formula for each data output represents a Boolean value under any assignment of the input variables, and that these values match our specification for floating-point addition. Our

floating-point addition specification is an ACL2 function that operates on two integers representing the bit-vector encoding of floating-point operands a and b, a third integer representing the SSE MXCSR or x87 FPCW register that specifies the rounding mode and exception masks, and two flags *apre* and *bpre*, determining whether the operands are to be considered register-format float values or values of a different "prenormalized" form that is used internally during certain operations. The specification function produces integer outputs representing *res*, the resulting sum of the operands, the new settings for the exception flags, and certain implementation-specific flags about the operation.

We attempt to ensure that our specifications are themselves correct. The specification functions are executable and may be tested against existing floating-point implementations. We have run millions of tests of our floating-point addition specification against Intel hardware implementations. Our specifications are also much more compact than the hardware implementations; for example, our SSE floating-point addition specification is about 525 lines of ACL2 code, which may be reviewed much more easily than the 33,000 lines of Verilog composing the FADD unit. For better symbolic execution performance, our floating-point addition specification is written at a level primarily involving bitwise operations on integers. However, our specification has also been proven to match an even higher-level, rational number-based function that carefully implements the IEEE floating point specification without optimization [54].

We use our GL tools to prove by symbolic execution that the evalua-

tion of the AIGs yields the same result as the ACL2 floating-point addition specification. This proof makes use of case-splitting based on the difference in the exponents of the two operands. Similar case-splitting schemes have been used in other floating-point addition verifications [5, 25]. Without such case-splitting, the mantissas of the two operands are shifted a variable amount relative to each other and then added together, and the BDD representation of such a function is exponential in size. In addition to the exponent difference, we consider whether the signs of the operands are the same (effective addition) or opposite (effective subtraction), and whether each of the operands is a normalized floating-point number or a special kind, either denormal, zero, infinite, indefinite, or signaling or quiet not-a-number (SNaN/QNaN). The cases we examine are outlined in Table 5.1. For the main cases of normal and prenormalized floats and zeros, we separately consider each exponent difference from -70 to 70 for double- and extended-precision addition and from -30 to 30 for single-precision, with signs opposite or equal also considered separately for each difference. Since an absolute exponent difference greater than 70 (30) is beyond the point where the mantissas no longer overlap during the addition, we consider all exponent differences greater than 70 (30) and less than -70 (-30) each as a block.

To implement this case splitting scheme, we define a function called Operand-Case-Okp$(a, b, case)$ that recognizes whether a pair of operands is appropriate for a particular subcase. This function takes the operand pair a, b and *case*, a list of parameters that describe the subcase of interest. This

Op1 Types	Op2 Types	Subdivisions
Normal Prenorm Denorm	Normal Prenorm Denorm	Equal/opposite sign, exponent difference
Normal Prenorm Denorm Zero	Zero	Equal/opposite sign
Zero	Normal Prenorm Denorm Zero	Equal/opposite sign
QNaN SNaN Indefinite Infinity	QNaN SNaN Indefinite Infinity	Equal/opposite sign
Normal Prenorm Denorm Zero	QNaN SNaN Indefinite Infinity	Equal/opposite sign
QNaN SNaN Indefinite Infinity	Normal Prenorm Denorm Zero	Equal/opposite sign

Table 5.1: Subcases for Floating-point Addition

parameter list includes the exponent difference, a flag indicating whether that exponent difference must be exact or whether any absolute difference greater than or equal to that difference is acceptable, a flag indicating whether the signs must be equal or opposite, and lists indicating the acceptable types of each operand. We supply Operand-Case-Okp as the case hypothesis (see Section 3.3) and provide a list of settings of the parameter list for each of the cases listed in Table 5.1. The GL clause processor then checks that this list of cases is exhaustive and splits the symbolic executions into these cases.

For symbolic execution to perform acceptably, the BDD ordering must be selected appropriately for each case. In particular, when adding or subtracting two normal numbers of a given exponent difference, we ensure that the BDD indices of the mantissa bits of the operands are arranged so that after shifting by the appropriate exponent difference, they will be interleaved to make the symbolic addition fast. Because we split into hundreds of cases for each addition instruction, we generate this list programmatically: we define a function that produces the appropriate shape specifiers (Section 3.1) when given *case*, the parameter list describing the subcase.

5.4 Results

Our verification efforts began after the relevant instructions had been thoroughly checked using a testing-based methodology. Even so, we discovered two bugs. The first bug was a timing anomaly affecting SSE addition instructions, which we found during our initial investigation of the media unit. Later,

a bug in the extended precision instruction was detected by symbolic simulation. This bug affected a total of four pairs of input operands out of the 2^{160} possible, producing a denormal result of twice the correct magnitude. Because of the small number of affected inputs, it is unlikely that random testing would have encountered the bug; directed testing had also not detected it. Both bugs have now been fixed.

Our proofs are collected in a regression suite that runs some proofs nightly and other, more expensive ones weekly. The instructions verified in these regressions include floating-point additions, comparisons, min/max functions, format conversions, logical operations, shuffles, and integer and floating-point multiplications. Our proofs range in speed from a few seconds (logical operations, comparisons) to several hours (multiplications). Single-precision floating-point addition takes 8 minutes, double-precision 40 minutes, and extended-precision 70 minutes.

Since these proofs were originally completed, there have been several design changes in the *fadd* unit. Usually, these changes do not break our proof scripts; in a few cases, input or output signals have been renamed and this requires simple changes in the scripts. In rare cases, the interface for certain instructions has significantly changed so that the specification for that instruction must be updated. So far, however, we have not needed to change our proof strategy or to prove additional lemmas in order to fix our verification scripts in response to a design change. Altogether, our verifications seem to be very robust with respect to design changes.

Chapter 6

Related Work

Several verification systems use symbolic execution as a core reasoning procedure in hardware and software verification. In this chapter, we outline the origins of symbolic execution in the software verification domain and its use in theorem proving as a replacement for verification condition generation. We also describe the use of symbolic simulation in hardware verification, particularly the development of symbolic trajectory evaluation for datapath verification. Because our work integrates a symbolic execution engine into a theorem proving framework, we also discuss the integration of automated verification routines into theorem provers.

6.1 Symbolic Execution in Software Verification

The first symbolic execution software was intended for program verification. The EFFIGY and SELECT systems were developed at IBM and Stanford Research Institute, respectively [10, 52, 53]. Both used symbolic expressions described at a level similar to the program code, with named variables standing for unconstrained inputs, and used a theorem prover to algebraically simplify these expressions. Both systems also tracked path conditions accu-

mulated from IF tests along each execution path in order to prune the set of possible execution paths, similar to our use of BDD path conditions for the same purpose (Section 2.2). The reductions used in these systems were necessarily incomplete, although SELECT implemented a complete decision procedure for systems of linear inequalities. Because of this incompleteness, loop structures in the targeted programs could cause the symbolic executions to fail to terminate even when the number of iterations was bounded; these systems therefore allowed the user to set an iteration limit.

More recently, Matthews et al. [58] used symbolic execution driven by a theorem proving engine as the basis of a software verification tool. The user provides a machine-code program annotated with assertions at cutpoints. The tool symbolically executes the program from each cutpoint, stopping when another cutpoint is reached, and attempts to prove that the assertion at the initial cutpoint implies the assertion at the next cutpoint reached during execution. This system avoids termination checking for looping control structures because the user is required to provide enough cutpoints to break each loop. This methodology is independent of the symbolic execution technology used; it only requires some way of symbolically executing non-looping programs. A possible area of future work is to apply this methodology to our symbolic execution platform and explore its suitability as a software verification tool.

GL uses a bit-based rather than term-based representation for symbolic objects, so our symbolic execution engine is more often able to decide control questions that arise during symbolic execution, such as whether there

is a possible execution path along a conditional branch and whether a loop has completed. In term-based engines, these questions are undecidable in general, whereas in our tool, they can be answered by a Boolean satisfiability check. However, our bit-based representation requires the range of the inputs to the symbolic execution to be restricted to a finite set. Because of this, our system is appropriate for a different class of problems than term-based symbolic execution engines. When a term-based symbolic execution engine is instrumented with a powerful enough set of rules for simplifying terms in a particular domain, it may be able to solve problems that cannot be approached by our Boolean reasoning-based engine. Our engine, on the other hand, is fully automatic without the addition of domain-specific rules if the inputs are restricted to a finite set.

6.2 Symbolic Simulation in Hardware Verification

Following the application of symbolic execution strategies to program verification, Darringer [29] suggested applying similar strategies to hardware verification. Bryant [20,21] proposed the use of ROBDDs as the representation for wire values in hardware symbolic simulation. Extending this strategy from Boolean logic to a ternary logic with X acting as an unknown value [19,22] led to the development of symbolic trajectory evaluation, a form of model-checking based on ternary symbolic simulation [39, 73].

Symbolic trajectory evaluation (STE) is a model-checking procedure for a simple linear temporal logic. In its usual form, the only temporal operator

is the next-time operator, but the STE algorithm has also been extended to handle other temporal operators [77]. This form of model-checking has been used at Intel as the basis for the Forte formal verification tool; among other efforts, this tool has recently been used to formally verify the execution cluster of the Core i7 processor [47]. Forte is an LCF-style theorem prover with a built-in STE engine called Voss; Forte's logic includes several STE-related inference rules that allow STE results to be soundly composed and shifted in time [3, 38, 59, 74].

Both our work and Intel's integrate a bit-level symbolic reasoning procedure into a theorem-proving framework. However, Forte is narrowly targeted to work with hardware designs whereas GL handles the complete general-purpose logic of ACL2. We also have built our symbolic execution engine in a verified manner so that a proof using our tool has the same level of trust as the ACL2 base system with the Hons extension, whereas Voss must be trusted in order to believe Forte's verification results. Through the process of proving GL's correctness, we found several bugs in our implementation that might have gone undetected if we had integrated it as an unverified tool.

Our approach to integrating symbolic execution into a theorem proving framework also differs from Intel's. We began with a full-fledged, general-purpose theorem prover and designed a symbolic execution engine for its logic. Intel began with the Voss STE engine, a symbolic simulation tool geared specifically toward hardware, and built a lightweight theorem prover around it, with the specific purpose of driving STE. As a result, our symbolic execu-

tion engine is applicable to a variety of non-hardware-related tasks. However, Intel's close integration of its theorem prover with its STE engine makes interaction with hardware models relatively user-friendly, e.g., "wiggling" and concrete simulations as well as symbolic simulations and STE-based proofs are all available through high-level commands [4, 74]. The integration of our symbolic execution engine with E hardware models (Chapter 4) requires more user direction to obtain theorems about the hardware model via symbolic simulation: the hardware's E representation is symbolically simulated in AIG mode, and the resulting AIGs are the verification targets. Composing proofs is labor-intensive; we lack an automated way to decompose hardware modules, prove lemmas about the pieces, then compose these proofs into one about the top-level module. As one area of future research, we intend to address these practical impediments by more tightly integrating our symbolic execution engine with our hardware modeling infrastructure, and by providing a framework in which hardware-related proofs may be spatially decomposed without much additional user effort.

Besides variations of symbolic trajectory evaluation, other forms of hardware verification range from traditional theorem proving approaches (requiring extensive user interaction) to fully-automatic model- and equivalence-checking based approaches. AMD has performed several verifications using traditional interactive theorem proving in ACL2, including floating-point addition, multiplication, and division hardware as well as microcode for the square root operation [67–69]. On the other hand, IBM uses a fully-automated ap-

proach for sequential equivalence and model checking using the SixthSense tool [6]. In addition to property and equivalence-checking, this tool has also been used for functional correctness verification; for example, it has been used to verify a floating-point fused multiply-add unit [45]. SixthSense has also been used as an external proof tool within ACL2 [71]. While equivalence- and model-checking based approaches offer an impressive degree of automation, we believe our approach offers more opportunity for manual composition of proofs, so that they might scale beyond the capacity of fully automated methods. We hope in future work to extend GL to take advantage of highly automated external tools.

6.3 Automatic Proof Routines in Theorem Provers

In interactive theorem proving, the most time-consuming portion of a verification task is usually the user interaction. A major thrust of theorem proving research is to integrate special-purpose routines that offer greater automation for certain kinds of problems, so the user can concentrate on higher-level problems. The goal is to allow the user to provide a high-level set of instructions, and let the automated routines fill in the details of the proof. Our work on GL is a step toward this same goal.

Many theorem provers include some built-in decision procedures that are useful in a wide variety of reasoning domains. For example, ACL2 and PVS both include decision procedures for equality-based reasoning and linear arithmetic [16, 28, 41, 62]. However, in many cases more domain-specific automatic

procedures are useful. As one example, we have discussed Intel's integration of the Voss symbolic trajectory evaluation engine in the Forte theorem prover. As another example, PVS contains a BDD-based symbolic model checker for the finite μ-calculus [75]. Many theorem provers allow the user to add new reasoning procedures.

The integration of user-defined reasoning methods into existing theorem provers takes many forms. In some cases, the extensions are designed so that the theorem prover with the new procedure still maintains the soundness of the unextended prover. Harrison [37] categorizes systems allowing such extensions as either *fully expansive* or *reflective*. Fully expansive extension systems include LCF-style provers such as HOL; in such systems, the new reasoning procedures must ultimately yield a proof described in terms of basic inference rules in the prover's logic. Reflective systems provide a metareasoning system by which the user-defined reasoning procedure may be proven to be sound. ACL2 implements a reflective extension system; it allows the user to define meta rules and clause processors which must be proven to operate soundly on (quoted) expressions [40, 51]. Additionally, the Milawa prover [31] uses a reflection mechanism to prove the correctness of its own rewriting engine.

In both the fully expansive and reflective extension mechanisms, the user is restricted in the programming language and style that may be used to program the external reasoning procedure. In many cases, it may be preferable to instead use an unverified external reasoning procedure. For example, it may be advantageous to write the reasoning procedure in a different programming

language than the one supported by the extension mechanism, particularly if the performance of the reasoning algorithm is critical. Pre-existing verification tools are also unlikely to be verifiable in either the fully-expansive or reflective setting. However, some such systems may create a proof certificate checkable in the host theorem proving system.

To integrate an unverified reasoning tool with a theorem prover, a user may reprogram the prover's internals; in principle, this is no more unsound than relying on the external tool. However, ACL2 provides a mechanism by which an unverified ("trusted") clause processor may be installed as a reasoning procedure [51]; this ameliorates the soundness risk of modifying the prover itself, although there is no guarantee of the external procedure's soundness.

Certain unverified extensions to ACL2 are notable. Reeber and Hunt [66] described a decidable subset of ACL2 called SULFA and provided a SAT-based decision procedure for that subset. This extension appeals to an external SAT-solver and thus could not be verified completely within ACL2's metatheory; it is implemented as a trusted clause processor. Sawada and Reeber [71] also designed an unverified procedure that calls IBM's SixthSense formal verification tool to perform hardware-related proofs.

6.4 Our Contribution

We have contributed a reasoning procedure that is fully automated, verified, and targets the entire logic of a general-purpose theorem prover, ACL2.

That is, any ACL2 theorem in which the free variables are restricted by hypotheses to a finite set may, in principle, be proven by our symbolic execution-based proof engine. Our proof engine is fully verified in ACL2, so that a proof completed using our procedure is as trustworthy as if it used only the built-in routines of the theorem prover itself. We believe this distinguishes our work from previous efforts in both the domains of interactive theorem proving and fully automatic verification methods such as are often used in hardware verification.

While the conjectures that may be proven by our framework may in principle also be solved by exhaustive testing, we have successfully applied this procedure to many problems that were beyond the capacity of exhaustive testing. For example, we have verified several functions of a commercial x86 processor design, including extended-precision floating point addition, for which the set of possible inputs is over 2^{160}.

The full codebase for the GL symbolic execution framework will be available in the distributed books of future ACL2 releases [49] and from the ACL2 books repository [2]. The GL release includes automated utilities for:

- running and examining output of symbolic executions
- proving theorems by symbolic execution, optionally with case-splitting
- defining symbolic counterparts for user-defined functions

- defining extended symbolic interpreters which may directly call a user-provided set of functions.

The GL codebase contains purely user-level ACL2 code, loadable using the standard ACL2 `INCLUDE-BOOK` command. The codebase contains over 20,000 lines of code among 66 ACL2 event files (books). These books contain over 700 function definitions and 1200 lemmas. This includes:

- definitions and correctness proofs of manually-generated symbolic counterparts (such as for ACL2 primitives): 100 definitions, 300 lemmas;
- the GL symbolic interpreter and clause processor and their correctness proofs: 190 definitions, 370 lemmas;
- the program transformation to automate the creation of symbolic counterparts for user-defined functions: 180 definitions, 150 lemmas;
- symbolic if-then-else implementation and correctness proofs: 40 definitions, 85 lemmas;
- basic definitions concerning the symbolic object format: 180 definitions, 290 lemmas.

This framework is currently in use at Centaur Technology for execution unit verification. It has been used to verify the implementations of many x86 instructions in the VIA Nano$^{\text{TM}}$ processor design, including floating-point additions and multiplications.

Chapter 7

Conclusions

We have implemented a symbolic execution framework for ACL2 that is integrated with the theorem prover and mechanically verified. This framework includes a proof procedure that produces theorems with the same level of trust as the ACL2 system with the HONS extension. This framework has become a core tool in the hardware verification flow at Centaur Technology, a designer of commercial x86-compatible processors.

Our symbolic execution engine operates primarily on the bit level, using BDDs as our Boolean function representation. Our bit-level representation allows our framework to prove many finite-domain conjectures automatically, scaling beyond the capacity of exhaustive testing on many problems. The proof procedure we have built around the symbolic execution engine also produces counterexamples in cases where it fails due to a false conjecture. Because of its automation and counterexample-generation capabilities, the need for the user to interactively guide the prover is reduced, and the user may prove or disprove desirable properties of the hardware or software under investigation without first obtaining a deep understanding of the details of the design.

In order to facilitate proofs using our symbolic execution engine, we pro-

vide a framework that automates the steps of such proofs, including coverage proofs, while still giving the user fine-grained control over performance-critical details such as case-splitting and BDD variable orderings. The interface to this framework is designed so that the user never needs to directly construct or reason about BDDs: the case split is given by an ACL2 function, and the symbolic inputs are constructed automatically from a shape specifier. This high-level interface also facilitates the automation of coverage proofs, which reason about the shape specifiers so as to avoid having to reason about the more complicated BDD-based symbolic inputs.

This framework is in use as a core formal verification tool at Centaur Technology. We have used it to verify many instructions of the VIA Nano™ processor, and verification work using this tool is ongoing. As we discuss in the following sections, we intend in future work to improve our framework so that it may be successfully applied to larger and more varied problems.

7.1 Future Work

We have many ideas for extending this work. Two possible directions were mentioned in the previous section: building a cutpoint-based software verification framework around our symbolic execution engine, and improving the integration between our symbolic execution framework and our hardware models. Another direction for future research is to allow more flexibility in the symbolic object format and symbolic reasoning procedures of our tool, which may increase the scalability of the system. For example, it may be helpful

to use an alternative Boolean function representation such as AIGs rather than BDDs as the basis of our bit-level representation (Subsection 7.1.1). For some problems it may also be helpful to replace the bits in our representation by ternary values such as are used in STE; in cases where certain input bits are irrelevant to the result, this may provide up to exponential speedup by replacing variables with X values (Subsection 7.1.2). A more extensive possible modification to our symbolic reasoning procedures would be to integrate term-based methods such as SMT, so that problems could be approached at a higher level of abstraction when possible.

Another direction of future research is toward finding a wider range of applications of our framework. As we discussed in the previous section, our symbolic execution engine might be usefully integrated into a cutpoint-based software verification framework. Another application is in bridging the semantic gap between the ACL2 logic and the languages of automated methods such as model checking (Subsection 7.1.3).

7.1.1 AIG-based symbolic execution

In our framework, each symbolic bit is expressed as a BDD. As future work we plan to explore using AIGs as the Boolean function representation and SAT-based reasoning. These two representations and reasoning strategies have complementary strengths: BDDs usually outperform SAT on regular, arithmetic-style reasoning when a good variable ordering exists, but SAT is more scalable on less-regular problems where there may be no efficient BDD

ordering. Because of this, we hope that allowing GL to use AIG/SAT based reasoning will allow us to solve a large class of problems that could not be handled by the BDD-based implementation alone. In recent work we have added to GL an AIG-based operating mode, so that the user may choose between using the AIG- and BDD-based representations for each proof; this AIG-based mode uses an external tool based on MiniSat [34] to decide the satisfiability of queries expressed as AIGs. Beyond this, we will consider methods by which the two strategies might be combined so that BDDs and AIGs/SAT might both be used within a single symbolic execution.

AIGs offer much more flexibility than BDDs in choosing how much reasoning to attempt at a given point in the computation. For example, when analyzing an IF test as in Section 2.2, we obtain BDDs describing the cases under which the test is known true, known false, and unknown. In our BDD-based system, this allows us to immediately tell which branches must be taken. In contrast, with an AIG representation, this question becomes a satisfiability query, but an optional one: we might instead choose to symbolically execute both branches. Although it is not clear when we should make this choice, it seems many optimizations are possible. For example, we would likely wish to perform some random simulation of the AIG in question, since this might immediately show that the AIG took both true and false values. Various simplification strategies may also be applied before attempting to check SAT directly. For example, DAG-aware rewriting [60] and cut-sweeping [33] offer the capability of substantially reducing the number of AIG nodes before SAT

Value	Boolean possibilities
$\top$	$\emptyset$
1	$\{1\}$
0	$\{0\}$
X	$\{0, 1\}$

Table 7.1: Four-valued logic

needs to be called. FRAIGing [61] and SAT-sweeping [55], though more expensive, achieve greater simplification by proving equivalences between AIG nodes using SAT.

7.1.2 Ternary symbolic execution

In symbolic trajectory evaluation, the values assigned to each wire in a hardware model are abstracted in two crucial ways. First, like the symbolic objects used in GL, they may be functions of Boolean variables. Second, the values taken by these functions are not simply the Boolean values 1 and 0, but may also be the value X, sometimes called $\bot$, representing an unknown value. Because there are now three values possible for each wire, this is termed ternary simulation [19, 23]. A fourth value, called $\top$, is also often added; this is an overconstrained value meaning that the wire is logically excluded from having either 1 or 0 as its value. These four values correspond to sets of simple Boolean values that the wire might take, described in Table 7.1. These four values form a complete lattice partially ordered by the subset relation, and the three-valued subset forms a semilattice.

In this section we focus on ternary simulation, since the primary advantage of either a three- or four-valued logic over a pure Boolean logic is the presence of the X value. Most of our comments are equally applicable to quaternary simulation.

In symbolic simulations, ternary values are usually encoded as pairs of bits. A particular symbolic simulation in a three-valued logic therefore requires computing twice as many Boolean formulas as a similar two-valued symbolic simulation. However, in practice, often either the two formulas encoding the symbolic value of a wire are closely related or else one is much simpler than the other. Because of this, three-valued symbolic simulations often require only a small overhead above similar two-valued symbolic simulations.

Offsetting this overhead, X values allow abstract values to be expressed without introducing new variables, sometimes reducing the number of Boolean variables needed to express a conjecture. That is, a simulation in which several inputs are set to X covers the same concrete Boolean cases as if each of those inputs were set to an independent Boolean variable. The trade-off is that while additional variables are expensive, simulating with X values loses precision. In hardware verification, often many inputs to a circuit are "don't-cares;" the loss of precision from setting these inputs to X is not important, and doing so prevents symbolic simulators from computing complicated formulas involving these inputs.

In our symbolic execution framework, we could take advantage of these savings by replacing each symbolic bit in our object representation with a

symbolic ternary value, encoded by a pair of Boolean functions. The evaluation of such a symbolic object would then yield a set of objects, one for each assignment of `T` or `NIL` to each X value. In general, the size of the set is exponential in the number of X present. However, there is no need to compute this set explicitly in order to use a ternary representation in a proof scheme similar to that described in Section 1.2.

The theory behind proof by symbolic execution requires modification to accomodate the fact that ternary symbolic objects evaluate to sets rather than single objects. Here we still write $\langle s \rangle(e)$ for the evaluation of a symbolic object s under environment e, with the understanding that this now represents a set of values. The coverage set of a ternary symbolic object is the union over the range of $\langle s \rangle$. A ternary symbolic execution of a function f on s then produces a symbolic object s' satisfying

$$\forall e \,.\, \langle s' \rangle(e) \supseteq \{ f(x) : x \in \langle s \rangle (e) \} \,. \tag{7.1}$$

Note that the evaluation of the result must only be a superset of the possible concrete results; in general, computation with X values may yield overapproximate results.

The definition of a symbolic counterpart for a ternary symbolic execution is the same as in Section 1.2, except that we use the definition of a ternary symbolic execution. That is, f_{sym} is a symbolic counterpart if it always yields a symbolic execution of f on its argument:

$$\forall s, e \,.\, \langle f_{sym}\,(s) \rangle(e) \supseteq \{ f(x) : x \in \langle s \rangle\,(e) \} \tag{7.2}$$

Finally, the mechanism for proof by ternary symbolic execution is a straightforward modification of Theorem 1.2.1:

Theorem 7.1.1 (Proof by ternary symbolic execution)**.** *Suppose:*

- *Some symbolic object* $\mathbf{s}$ *covers the set of inputs satisfying* HYP*:*

$$\forall x \;.\; \textsc{Hyp}(x) \Rightarrow \exists e \;.\; x \in \langle \mathbf{s} \rangle(e) \tag{7.3}$$

- $\textsc{Concl}_{sym}$ *is a symbolic counterpart for* CONCL*:*

$$\forall s, e \;.\; \langle \textsc{Concl}_{sym}\,(s) \rangle(e) \supseteq \{\textsc{Concl}(x) : x \in (\langle s \rangle(e))\} \tag{7.4}$$

- $\textsc{Concl}_{sym}(\mathbf{s})$ *yields a constant-true symbolic value:*

$$\forall e \;.\; \{\mathit{true}\} \supseteq \langle \textsc{Concl}_{sym}\,(\mathbf{s}) \rangle(e). \tag{7.5}$$

Then

$$\forall x \;.\; \textsc{Hyp}(x) \Rightarrow \textsc{Concl}(x).$$

Implementation of this modification to GL is conceptually straightforward. However, ACL2 has better support for equality-based reasoning than it does for transitive/reflexive relations such as $\subseteq$, so completing the necessary proofs may be more challenging. We expect there to also be interesting research related to improving proof automation and the user experience.

7.1.3 Symbolic model checking

Beyond increasing the scalability of our system, there is also possible future research in expanding its applications. To date we have used GL in datapath-related hardware verification problems, where the properties we sought to prove have involved finitely-constrained input spaces. However, a large class of hardware verification problems are more naturally stated as temporal logic properties of finite-state machines and proven using model checking. Our framework may aid in translating machine models and properties expressed in ACL2 into an appropriate form for model checking.

Expressing arbitrary temporal logic properties in a first-order logic such as ACL2 is not straightforward. However, safety properties (of the form **AG**P in CTL or **G**P in LTL) may be expressed in terms of paths through the state transition graph. To posit that **AG**P holds at initial state **s** is equivalent to stating that, for all finite paths through the state transition graph beginning at **s**, the final state s' in the path satisfies P(s'). Liveness properties **AF**P or **F**P may be transformed into safety properties, as demonstrated by Biere et al [8, 72].

One might wish to combine ACL2 with a model checker so that it could act as a decision procedure for certain specially formulated theorems such as the safety property above; such a model checker could either be built in the ACL2 logic or installed as a trusted external tool. Suppose the finite state machine model and the property to be checked are defined as a set of ACL2 functions:

- MSTATEP(s), a predicate recognizing the (finitely many) well-formed states of the machine
- TR(s, s'), the FSM's transition relation, recognizing pairs of states such that s may transition to s',
- P(s), a predicate recognizing states satisfying some atomic property.

A model checker must be able to determine the set of states to which a given state may transition: either a list of next-states for an explicit-state model checker, or for a symbolic model checker, a symbolic expression for the set of next-states, usually expressed as a Boolean function. A transition relation defined as an ACL2 function such as TR does not make this explicit. However, we may use GL to symbolically execute the transition relation on two independent, symbolic states, computing a Boolean expression for the transition relation. Similarly, atomic properties expressed as ACL2 functions P could be symbolically executed on a symbolic state, yielding a Boolean expression for that property. Thus an FSM model expressed at a high level in the ACL2 logic could be connected in a verifiable manner with one expressed in a Boolean language suitable for input to a symbolic model checker.

Integration of model checkers with theorem provers is not new: model checkers have been integrated into PVS [64, 70, 75] and HOL [46]; however, defining the semantics of LTL and CTL inside ACL2 is not straightforward due to its first-order logic. For example, the meaning of **AF**P(s) is "for every (infinite-length) path in the state transition graph beginning at s, there is

some i such that the ith state in the path satisfies P." This cannot be stated directly in ACL2 since paths of infinite length are not representable. A subtly different statement "there is some i such that all paths of length i beginning at s contain a state satisfying P" is equivalent in the finite-state case [8], since if there is a counterexample of length equal to the total number of states, there is a lasso-shaped (eventually periodic) counterexample path of infinite length. Ray et al defined a semantics for LTL in ACL2 [65] using this insight.

7.1.4 Generalizing the Approach

ACL2 has several features that have been very advantageous in implementing the GL framework: its logic is connected with a fast execution engine, and its terms may be represented and manipulated for metareasoning purposes within its logic. Furthermore, the hash-consing and memoization capabilities of the Hons extension yield a reasonably efficient BDD package. It is questionable whether a symbolic execution tool similar to GL could be feasibly built in another theorem proving system without first adding similar capabilities. Some other theorem provers, such as Isabelle/HOL [7] and Coq [35] have a mechanism by which the result of a ground term's execution may be inferred to be equal to the term. Reflective decision procedures have also been defined and verified in these two systems, demonstrating that their execution mechanisms can be useful for metareasoning [9,24]. Additionally, Milawa [31] is quite similar to ACL2 in its support for execution and meta-reasoning capabilities.

ACL2's logic is also dynamically typed, which appears advantageous

for the kind of reflective reasoning needed in a framework such as GL. For a statically typed logic, several difficulties would require approaches that differ from ours. For example, in our approach a symbolic object evaluator is capable of producing concrete objects of many different types, and we may bind variables to objects of many different types in an evaluation environment.

The GL framework uses a reasoning strategy based on BDDs, and we have discussed future modifications based on changing the bit representation. However, the general strategy we used in creating a verified symbolic execution engine would be equally applicable to other symbolic data representations and reasoning procedures. To create a verified symbolic execution engine, one defines (1) a symbolic object language with an evaluation semantics, (2) for each primitive function of the logic, a function that is provably a symbolic counterpart, and (3) a method of tracking path conditions and deciding control-flow tests. While alternative symbolic data formats and reasoning methods may present different challenges in verifying such a framework, our work demonstrates the feasibility of this approach.

Bibliography

[1] *IEEE Standard (1364-2005) for Verilog Hardware Description Language*. IEEE, 2005.

[2] ACL2 books repository. `http://acl2-books.googlecode.com`, 2010.

[3] Mark Aagaard, Robert Jones, and Carl-Johan Seger. Lifted-FL: a pragmatic implementation of combined model checking and theorem proving. In *Theorem Proving in Higher Order Logics*, page 840. 1999.

[4] Mark D. Aagaard, Robert B. Jones, Thomas F. Melham, John W. O'Leary, and Carl-Johan H. Seger. A methodology for large-scale hardware verification. In *Formal Methods in Computer-Aided Design*, pages 255–261. 2000.

[5] Mark D. Aagaard, Robert B. Jones, and Carl-Johan H. Seger. Formal verification using parametric representations of Boolean constraints. In *Proceedings of the 36th Design Automation Conference*, pages 402–407, 1999.

[6] Jason Baumgartner, Hari Mony, Viresh Paruthi, Robert Kanzelman, and Geert Janssen. Scalable sequential equivalence checking across arbitrary design transformations. *International Conference on Computer Design*, pages 259 –266, 2006.

[7] Stefan Berghofer and Tobias Nipkow. Executing higher order logic. In *Selected papers from the International Workshop on Types for Proofs and Programs*, pages 24–40. Springer-Verlag, 2002.

[8] Armin Biere, Cyrille Artho, and Viktor Schuppan. Liveness checking as safety checking. *Electronic Notes in Theoretical Computer Science*, 66(2):160 – 177, 2002.

[9] Samuel Boutin. Using reflection to build efficient and certified decision procedures. *Theoretical Aspects of Computer Science*, 1281:515—529, 1997.

[10] Robert S. Boyer, Bernard Elspas, and Karl N. Levitt. SELECT – a formal system for testing and debugging programs by symbolic execution. In *Proceedings of the International Conference on Reliable Software*, pages 234–245, Los Angeles, California, 1975. ACM.

[11] Robert S. Boyer, David M. Goldschlag, Matt Kaufmann, and J Strother Moore. Functional instantiation in first-order logic. In *Artificial Intelligence and Mathematical Theory of Computation: Papers in honor of John McCarthy*, pages 7–26. Academic Press Professional, Inc., 1991.

[12] Robert S. Boyer and Warren A. Hunt, Jr. Function memoization and unique object representation for ACL2 functions. In *Proceedings of the Sixth International Workshop on the ACL2 Theorem Prover and its Applications*, Seattle, Washington, 2006. ACM Digital Library.

[13] Robert S. Boyer and Warren A. Hunt, Jr. The E language. In *International Workshop on Hardware Design and Functional Languages*, 2007.

[14] Robert S. Boyer and Warren A. Hunt, Jr. Symbolic simulation in ACL2. *Proceedings of the Eighth International Workshop on the ACL2 Theorem Prover and its Applications*, 2009.

[15] Robert S. Boyer and J Strother Moore. Metafunctions: Proving them correct and using them efficiently as new proof procedures. In *The Correctness Problem in Computer Science*, International lecture series in computer science. 1979.

[16] Robert S. Boyer and J Strother Moore. Integrating decision procedures into heuristic theorem provers: A case study of linear arithmetic. *Machine Intelligence*, 11:83—124, 1985.

[17] Bishop C. Brock and Warren A. Hunt, Jr. The formalization of a simple hardware description language. In *Formal VLSI specification and synthesis: proceedings of the IFIP WG 10.2/WG 10.5 International Workshop on Applied Formal Methods for Correct VLSI Design*, page 83, 1990.

[18] Bishop C. Brock and Warren A. Hunt, Jr. The DUAL-EVAL hardware description language and its use in the formal specification and verification of the FM9001 microprocessor. *Formal Methods in System Design*, 11:71–104, 1997. 10.1023/A:1008685826293.

[19] Randal Bryant and Carl-Johan Seger. Formal verification of digital circuits using symbolic ternary system models. In *Computer-Aided Verification*, pages 33–43. 1991.

[20] Randal E. Bryant. Graph-based algorithms for Boolean function manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, August 1986.

[21] Randal E. Bryant. Symbolic simulation — techniques and applications. *Proceedings, 27th ACM/IEEE Design Automation Conference*, pages 517–521, June 1990.

[22] Randal E. Bryant, Derek L. Beatty, and Carl-Johan H. Seger. Formal hardware verification by symbolic ternary trajectory evaluation. In *Proceedings of the 28th ACM/IEEE Design Automation Conference*, pages 397–402, San Francisco, California, United States, 1991. ACM.

[23] J. A. Brzozowski and M. Yoeli. On a ternary model of gate networks. *IEEE Transactions on Computers*, 28(3):178–184, 1979.

[24] Amine Chaieb and Tobias Nipkow. Verifying and reflecting quantifier elimination for presburger arithmetic. In *Logic for Programming, Artificial Intelligence, and Reasoning*, pages 367–380. 2005.

[25] Yirng-An Chen and Randal E. Bryant. Verification of floating-point adders. *Computer Aided Verification*, 1427, 1998.

[26] Oliver Coudert and Jean Christophe Madre. A unified framework for the formal verification of sequential circuits. In *Computer-Aided Design, 1990. ICCAD-90. Digest of Technical Papers., 1990 IEEE International Conference on*, pages 126–129, 1990.

[27] Jack W. Crenshaw. Integer square roots (Programmer's toolbox). *Embedded Systems Programming*, 11:15–32, 1998.

[28] D. Cyrluk, S. Rajan, N. Shankar, and M. Srivas. Effective theorem proving for hardware verification. In *Theorem Provers in Circuit Design*, pages 203–222. 1995.

[29] John A. Darringer. The application of program verification techniques to hardware verification. In *Papers on twenty-five years of electronic design automation*, pages 373–379. ACM, 1988.

[30] Jared Davis. Defaggregate: Automated introduction of named tuples in ACL2. Unpublished.

[31] Jared Davis. *A Self-Verifying Theorem Prover*. PhD thesis, The University of Texas at Austin, 2009.

[32] Jared Davis and Sol Swords. Flag.lisp: Introduce induction schemes for mutually recursive functions. Available at `http://acl2-books.googlecode.com/svn/trunk/tools/flag.lisp`.

[33] Niklas Een. Cut sweeping. Technical report, Cadence, 2007.

[34] Niklas Een and Niklas Sörensson. An extensible SAT-solver. In *Theory and Applications of Satisfiability Testing*, pages 502–518. Springer, 2003.

[35] Benjamin Grégoire and Xavier Leroy. A compiled implementation of strong reduction. *ACM SIGPLAN Notices*, 37(9):235–246, 2002.

[36] David Greve and Matthew Wilding. Using MBE to speed a verified graph pathfinder. *Proceedings of the Fourth International Workshop on the ACL2 Theorem Prover and its Applications*, 2003.

[37] John Harrison. Metatheory and reflection in theorem proving: A survey and critique. Technical report, 1995.

[38] Scott Hazelhurst and Carl-Johan H. Seger. A simple theorem prover based on symbolic trajectory evaluation and BDDs. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 14(4):413–422, 1995.

[39] Scott Hazelhurst and Carl-Johan H. Seger. Symbolic trajectory evaluation. In *Formal Hardware Verification*, volume 1287 of *Lecture Notes in Computer Science*, pages 3–78. Springer, Berlin/Heidelberg, 1997.

[40] Warren A. Hunt, Jr., Matt Kaufmann, Robert Bellarmine Krug, J Strother Moore, and Eric Whitman Smith. Meta reasoning in ACL2. In *Theorem Proving in Higher Order Logics*, pages 163–178. 2005.

[41] Warren A. Hunt, Jr., Robert Krug, and J Strother Moore. Linear and nonlinear arithmetic in ACL2. In *Correct Hardware Design and Verification Methods*, pages 319–333. 2003.

[42] Warren A. Hunt, Jr. and Erik Reeber. Formalization of the DE2 Language. *Proceedings of the 13th Conference on Correct Hardware Design and Verification Methods (CHARME 2005)*, pages 20–34, 2005.

[43] Warren A. Hunt, Jr. and Sol Swords. Centaur technology media unit verification. In *Proceedings of the 21st International Conference on Computer Aided Verification*, 2009.

[44] Warren A. Hunt, Jr., Sol Swords, Jared Davis, and Anna Slobodova. Use of formal verification at Centaur Technology. In *Design and Verification of Microprocessor Systems for High-Assurance Applications*, pages 65–88. 2010.

[45] Christian Jacobi, Kai Weber, Viresh Paruthi, and Jason Baumgartner. Automatic formal verification of fused-multiply-add FPUs. In *Design, Automation and Test in Europe, 2005. Proceedings*, pages 1298–1303 Vol. 2, 2005.

[46] Jeffrey Joyce and Carl Seger. The HOL-Voss system: Model-checking inside a general-purpose theorem-prover. In *Higher Order Logic Theorem Proving and Its Applications*, pages 185–198. 1994.

[47] Roope Kaivola, Rajnish Ghughal, Naren Narasimhan, Amber Telfer, Jesse Whittemore, Sudhindra Pandav, Anna Slobodová, Christopher Taylor, Vladimir Frolov, Erik Reeber, and Armaghan Naik. Replacing testing with formal verification in Intel® Core™ i7 processor execution engine validation. In *Proceedings of the 21st International Conference on Computer Aided Verification*, 2009.

[48] Matt Kaufmann and J Strother Moore. An industrial strength theorem prover for a logic based on Common Lisp. *IEEE Transactions on Software Engineering*, 23(4):203–213, 1997.

[49] Matt Kaufmann, J Strother Moore, and Robert S. Boyer. ACL2 version 4.1. `http://www.cs.utexas.edu/~moore/acl2/`, 2010.

[50] Matt Kaufmann, J Strother Moore, and Panagiotis Manolios. *Computer-Aided Reasoning: An Approach*. Kluwer Academic Publishers, Norwell, MA, USA, 2000.

[51] Matt Kaufmann, J Strother Moore, Sandip Ray, and Erik Reeber. Integrating External Deduction Tools with ACL2. *Proceedings of the 6th International Workshop on Implementation of Logics*, 2006.

[52] James C. King. A new approach to program testing. In *Proceedings of the international conference on reliable software*, pages 228–233, Los Angeles, California, 1975. ACM.

[53] James C. King. Symbolic execution and program testing. *Communications of the ACM*, 19(7):385–394, 1976.

[54] Robert Krug. Correctness proof of ACL2 floating-point addition specification. Unpublished.

[55] Andreas Kuehlmann. Dynamic transition relation simplification for bounded property checking. *IEEE/ACM International Conference on Computer Aided Design*, pages 50–57, 2004.

[56] Andreas Kuehlmann, Malay K. Ganai, and Viresh Paruthi. Circuit-based boolean reasoning. In *Proceedings of the 38th Design Automation Conference*, pages 232–237, 2001.

[57] Hanbing Liu and J Strother Moore. Java program verification via a JVM deep embedding in ACL2. In *Theorem Proving in Higher Order Logics*, pages 117–125. 2004.

[58] John Matthews, J Strother Moore, Sandip Ray, and Daron Vroon. Verification condition generation via theorem proving. In *Logic for Programming, Artificial Intelligence, and Reasoning*, pages 362–376. 2006.

[59] Thomas Melham. Integrating model checking and theorem proving in a reflective functional language. *Integrated Formal Methods*, 2004.

[60] Alan Mishchenko, Satrajit Chatterjee, and Robert Brayton. DAG-aware AIG rewriting: a fresh look at combinational logic synthesis. In *Proceed-*

ings of the 43rd annual Design Automation Conference, pages 532–535, San Francisco, CA, USA, 2006. ACM.

[61] Alan Mishchenko, Satrajit Chatterjee, Roland Jiang, and Robert Brayton. FRAIGs: A unifying representation for logic synthesis and verification. *EECS Dept., UC Berkeley, Tech. Rep*, 3, 2005.

[62] S. Owre, S. Rajan, J. M. Rushby, N. Shankar, and M. Srivas. PVS: combining specification, proof checking, and model checking. In *Computer Aided Verification*, pages 411–414. 1996.

[63] Viresh Paruthi, Christian Jacobi, and Kai Weber. Efficient symbolic simulation via dynamic scheduling, dont caring, and case splitting. In *Correct Hardware Design and Verification Methods*, pages 114–128. 2005.

[64] S. Rajan, N. Shankar, and M. Srivas. An integration of model checking with automated proof checking. In *Computer Aided Verification*, pages 84–97. 1995.

[65] Sandip Ray, John Matthews, and Mark Tuttle. Certifying compositional model checking algorithms in ACL2. In *Proceedings of the Fourth International Workshop on the ACL2 Theorem Prover and its Applications*, 2003.

[66] Erik Reeber and Warren A. Hunt, Jr. A SAT-based decision procedure for the subclass of unrollable list formulas in ACL2 (SULFA). *Proceedings*

of the 3rd International Joint Conference on Computer-Aided Reasoning (IJCAR 2006), 4130:453–467, 2006.

[67] David 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.

[68] David M. Russinoff. A mechanically checked proof of correctness of the AMD K5 floating point square root microcode. *Formal Methods in System Design*, 14:75–125, 1999. 10.1023/A:1008669628911.

[69] David M. Russinoff. A case study in formal verification of Register-Transfer logic with ACL2: the floating point adder of the AMD Athlon™ processor. In *Formal Methods in Computer-Aided Design*, pages 22–55, 2000.

[70] Hassen Saïdi and Natarajan Shankar. Abstract and model check while you prove. In *Computer Aided Verification*, pages 681–682. 1999.

[71] Jun Sawada and Erik Reeber. ACL2SIX: a hint used to integrate a theorem prover and an automated verification tool. *Proceedings of the 6th International Conference on Formal Methods in Computer-Aided Design (FMCAD 2006), Springer-Verlag, San Jose, CA*, page 161168, 2006.

[72] Viktor Schuppan and Armin Biere. Efficient reduction of finite state model checking to reachability analysis. *International Journal on Software Tools for Technology Transfer (STTT)*, 5(2):185–204, March 2004.

[73] Carl-Johan H. Seger and Randal E. Bryant. Formal verification by symbolic evaluation of partially-ordered trajectories. In *Formal Methods in System Design*, pages 147–190, 1993.

[74] Carl-Johan H. Seger, Robert B. Jones, John W. O'Leary, Tom Melham, Mark D. Aagaard, Clark Barrett, and Don Syme. An industrially effective environment for formal hardware verification. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 24(9):1381, 2005.

[75] Natarajan Shankar. Using decision procedures with a higher-order logic. *14th Interational Conference on Theorem Proving in Higher Order Logics*, 2152:5—26, 2001.

[76] Sol Swords and Warren A. Hunt, Jr. A mechanically verified AIG to BDD conversion algorithm. In *Interactive Theorem Proving*, pages 435–449. 2010.

[77] Jin Yang and Carl-Johan H. Seger. Introduction to generalized symbolic trajectory evaluation. *International Conference on Computer Design*, 00:0360, 2001.