A SAT-Based Procedure for Verifying Finite State Machines in ACL2

Warren A. Hunt, Jr.
University of Texas at Austin
Department of Computer Sciences
hunt@cs.utexas.edu

Erik Reeber
University of Texas at Austin
Department of Computer Sciences
reeber@cs.utexas.edu

ABSTRACT

We describe a new procedure for verifying ACL2 properties about finite state machines (FSMs) using satisfiability (SAT) solving. We present an algorithm for converting ACL2 conjectures into conjunctive normal form (CNF), which we then output and check with an external satisfiability solver. The procedure is directly available as an ACL2 proof request. When the SAT tool is successful, a theorem is added to the ACL2 system database as a lemma for use in future proof attempts. When the SAT tool is unsuccessful, we use its output to construct a counter-example to the original ACL2 property.

Categories and Subject Descriptors

B.5.2 [**Register-Transfer-Level Implementation**]: Design Aids—*Verification*; F.4.3 [**Mathematical Logic and Formal Languages**]: Formal Languages—*Decision problems*; F.4.1 [**Mathematical Logic and Formal Languages**]: Mathematical Logic—*Mechanical theorem proving, Recursive function theory*

General Terms

verification, algorithms

Keywords

hardware verification, Satisfiability solving, theorem proving, ACL2

1. INTRODUCTION

The ACL2 theorem prover and logic have been used in many large industrial and academic hardware verification projects, such as the verification of AMD's K7 floating point unit [14] and the verification of an advanced pipelined machine [15]. Verification techniques based on satisfiability (SAT) solvers have also proven to be of use in hardware verification [17]. Techniques based on SAT solvers complement ACL2 well, since SAT solvers are used in fully automatic techniques that do not scale well to large designs, whereas the ACL2 theorem prover is not fully automatic but does scale to large designs.

We are therefore interested in discovering subclasses of decidable ACL2 properties that can be reduced to SAT. In a previous paper, we described the Subclass of Unrollable List Formulas in ACL2 (SULFA) and showed that it can be reduced to SAT [13]. SULFA is an interesting subclass because properties involving a finite number of steps of a finite state machine, modelled in ACL2, can be expressed in it. We can therefore reduce many interesting hardware properties to properties in SULFA.

We have extended the ACL2 theorem prover to include a new hint that proves or disproves SULFA properties through the use of a SAT-based decision procedure. In this paper we describe in detail the algorithm used to convert SULFA properties into the input of a SAT solver. We also present the sketch of a proof that this new hint only verifies valid properties. Furthermore, we show how to create ACL2 counter-examples for invalid SULFA properties.

We begin this paper with a simple example that illustrates how our new hint allows the verification of properties that would otherwise require manual guidance to prove. Next, in Section 2, we give background information on SAT solving and describe SULFA. In Section 3, we show how the input of SAT solvers can be embedded in the ACL2 logic. In Section 4 we describe our SAT-based procedure for verifying SULFA properties. In Section 5 we show how our algorithm performs on some examples and compare it to other techniques. In Section 6 we discuss related work in more detail, before concluding in Section 7.

Example

To show how our tool is used, we present a simple FSM that implements a ten-digit decimal counter. Each digit is represented by four Boolean values. The property we would like to prove is that if every digit of the counter is initially between zero and nine, then every digit of the counter forever remains between zero and nine.

We represent the state for this counter as an n-element list where each element is a list of four Boolean values. We then model the counter as a function `(run-counter n init clk)` that, given the initial state, returns the counter's state after *clk* clock cycles.

We define a function, `(valid-digits n x)`, to recognize whether `x` is a list of `n` elements that contains only digits between zero and nine. Our property is represented by the

ACL2 '06 Seattle, Washington USA

following ACL2 theorem.

```
(defthm valid-digit-invariant
  (implies
   (valid-digits 10 x)
   (valid-digits 10 (run-counter 10 x clk))))
```

ACL2 can prove this theorem automatically if we have already added the following theorem to ACL2's database.

```
(defthm valid-digit-step
  (implies
   (valid-digits 10 x)
   (valid-digits 10 (next-counter-state 10 x))))
```

Here `(next-counter-state n x)` returns the next state of the counter given the state from the previous cycle.

A typical way to prove this invariant is to generalize it into a theorem about an `n`-digit counter and then use induction and some lemmas to prove the generalized invariant. With our ACL2 system extension, however, we prove the theorem automatically in 0.65 seconds (proving the invariant for a hundred digit counter takes about thirteen seconds). Note that our extension does not require `x` to be restricted to a finite domain; no hypothesis is required to tell the extension that `x` is a list of 10 four bit, bit vectors.

2. BACKGROUND

We assume the reader is familiar with ACL2. For more information on ACL2, see the book by Kaufmann, Manolios, and Moore [5]. In this section we give an overview of SAT solving and define the SULFA subclass of ACL2 formulas.

2.1 Satisfiability Solving

In this paper we refer to satisfiability (SAT) solving as determining whether there exists an instantiation of variables to Booleans that satisfies a formula in Conjunctive Normal Form (CNF). The following is a simple example of a CNF formula:

$$\exists x_0, x_1, x_2 : (x_0 \vee x_1 \vee x_2) \wedge (\neg x_0 \vee \neg x_1) \wedge (x_0 \vee \neg x_2)$$

A CNF formula consists of a conjunction of clauses. A *clause* is a disjunction of literals. Each *literal* is a Boolean variable or its negation.

SAT solving is known to be NP-Complete. Nevertheless, there are tools that can solve a wide array of practical problems [10]. SAT solving is particularly well-suited to integration with other tools since SAT solvers usually conform to a standard input format.

2.2 The Subclass of Unrollable List Formulas in ACL2 (SULFA)

In our previous paper we identified a decidable subclass of ACL2 formulas, SULFA [13]. As illustrated by our previous decimal counter-example, SULFA is sufficiently expressive to describe useful FSM properties succinctly. In our previous paper, we also show that invariants used to verify components of the TRIPS processor, a prototype grid processor designed by the University of Texas and IBM [1], can be reduced to SULFA properties.

The idea behind SULFA is to include only functions that can be unrolled into expressions of `car`, `cdr`, `cons`, `consp`, and `if`. An expression involving only these primitives is known to be decidable [11]. Our hint mechanism contains an efficient recognizer of SULFA properties, which generates an error if a user attempts to apply our procedure to a property that is not in SULFA. This recognizer is defined as

```
(defun concatn (n a b)
  (if (zp n)
      b
    (cons (car a) (concatn (- n 1) (cdr a) b))))

(defun uandn (n a)
  (if (zp n)
      t
    (if (car a)
        (uandn (- n 1) (cdr a))
      nil)))

(defun bequiv (a b)
  (if a b (not b)))

(defthm SULFA-thm
 (bequiv (uandn 4 (concatn 2 a b))
         (and (uandn 2 a) (uandn 2 b))))

(defthm general-thm
 (bequiv (uandn (+ x y) (concatn x a b))
         (and (uandn x a) (uandn y b))))
```

Figure 1: In this example we present the theorem that the unary-and of the concatenation of two bit vectors is equivalent to the conjunction of the unary-and of each individual bit vector. Given vectors of known widths this theorem is in SULFA. However, the more general theorem must be verified via traditional theorem proving.

a set of mutually recursive ACL2 functions. These include functions that calculate the non-list arguments of any ACL2 function and the unbound non-list variables of an expression, as defined below:

- The *non-list arguments* of a function, intuitively, form the subset of formal arguments that must be constant in order for a function application to be unrolled into the primitives `car`, `cdr`, `cons`, `consp`, and `if`. Since no argument to a list primitive needs to be constant to unroll it into itself, its set of non-list arguments is the full set of its formal arguments. Since any other primitive cannot be converted into a list primitive, its non-list arguments is the empty set. For non-primitives, the set of non-list arguments is defined as the minimal fix point that includes both unbound variables occurring in the function's measure and unbound non-list variables occurring in the function's body.
- The *unbound non list variables* of an ACL2 expression form a subset of the unbound variables in that expression. This subset is calculated by looking at the arguments to each function application. If an argument corresponds to a non-list argument of the applied function, then all unbound variables occurring in it are unbound non-list variables.

An ACL2 formula is in SULFA if two conditions are met. First all functions in the formula must be executable and total (i.e. defined with `defun`). Second the set of unbound non-list variables in the formula must be empty.

Consider the functions and theorems shown in Fig. 1. The functions `concatn` and `uandn` each only have one non-list argument, `n`. The argument `n` is a non-list argument since it

is used in the termination measure of the function and is used as an argument to `zp`. Since `a` and `b` are only used as arguments to `car` and `cdr`, these arguments are list arguments. Therefore the theorem `SULFA-thm` is in SULFA, since all non-list arguments in it are constant. On the other hand, the theorem `general-thm`, is not in SULFA, since `(+ x y)`, `x`, and `y` are all non-constant expressions used as non-list arguments.

3. FORMALIZING CNF IN ACL2

In order to combine a SAT solver with the ACL2 logic, we first show how to represent the input of a SAT solver, an existentially quantified Boolean CNF formula, in the ACL2 logic. We represent this CNF formula by negating it and driving the negation through the existential quantification. For example the following negated CNF formula

$$\neg(\exists x_0, x_1, x_2 : (x_0 \vee x_1 \vee x_2) \wedge (\neg x_0 \vee \neg x_1) \wedge (x_0 \vee \neg x_2))$$

corresponds to the following ACL2 formula:

```
(not (and (or x0 x1 x2)
          (or (not x0) (not x1))
          (or x0 (not x2))))
```

Note that it is not necessary to constrain variables to be Boolean, since in the ACL2 logic every formula X is equivalent to $(X \neq \texttt{nil})$.

4. CONVERSION ALGORITHM

In our previous paper [13], we present a simple algorithm for converting a SULFA property into SAT and argue that this algorithm forms a valid decision procedure for SULFA properties. This simple algorithm relies on expanding function calls and replacing `(if x y z)` with `(and (or (not x) y) (or x z))`. This algorithm, however, does not achieve adequate performance. Using our implementation of the simple conversion algorithm we could not prove the associativity of an eight bit adder. In this section we present a more efficient algorithm that can verify the associativity of a 200 bit adder.

The problem with the simple algorithm is that function call expansion and `if` removal may cause unnecessary exponential increases in the size of the formula. We can alleviate this problem by creating new variables, in a similar way to that used by Tseitin [16]. In particular we create variables for function parameters and the conditions of `if` expressions. Creating new variables and properly handling the expressions containing these variables adds a considerable amount of complexity to the algorithm. In order to make the high-performance algorithm easier to understand, we divide it into five phases: initial clausification, equal clause simplification, relevant boolean discovery, boolean variable creation, and iff removal. Each phase produces an ACL2 formula that is closer to a CNF formula than its input.

In this section we describe each phase and use the `SULFA-thm` example from Fig. 1 to illustrate it. We also prove that the final phase produces a valid formula only if the original ACL2 formula is also valid. Finally we present our algorithm for generating ACL2 counter-examples from a satisfying instance to a CNF formula.

4.1 Initial Clausification

The first phase of our algorithm begins by converting the ACL2 conjecture into a negated conjunction. We refer to each member of this conjunction as a clause. The initial clausification phase is illustrated in Fig. 2. The example input is the theorem `SULFA-thm` shown in Fig. 1. This property states that the unary-and of two, one bit, bit-vectors produces the same result as the conjunction of the unary-and of each component (note that `(and x y)` is an abbreviation for `(if x y nil)`). A new variable, `x0`, is created to represent the original ACL2 property.

```
(bequiv (uandn 2 (concatn 1 a b))
        (if (uandn 1 a) (uandn 1 b) nil))

⇒

(not (not (bequiv
           (uandn 2 (concatn 1 a b))
           (if (uandn 1 a) (uandn 1 b) nil))))

⇒

(nand (equal x0
             (bequiv
              (uandn 2 (concatn 1 a b))
              (if (uandn 1 a) (uandn 1 b) nil)))
      (not x0))
```

Figure 2: A simplified version of the theorem shown in Fig. 1 is converted into a negated conjunction.

4.2 Equal Clause Simplification

The next phase, the equal clause simplification phase, removes `if`, `cons`, and user-defined functions. Equal clause simplification involves repeatedly applying the four simplification steps shown in Fig. 3. These four steps are:

1. New variables are created for non-primitive function calls that occur as arguments to other function calls and in the condition parameter of a call to `if`. For example, in part A of Fig. 3 we create variables for each parameter of `bequiv`. Eventually a variable will also be created for the condition of `(if (uandn 1 a) (uandn 1 b) nil)` (this is the variable `x4` in Fig. 4).

2. A non-primitive function call is expanded once its arguments contain no non-primitive function calls themselves, as shown in part B of Fig. 3. After a function call is expanded any obvious simplifications are performed—constant expressions are evaluated, `(if T y z)` is replaced with `y`, `(car (cons x y))` is replaced by `x`, etc.. In our example `concatn` is opened using the definition from Fig. 1 and `(zp 1)` is evaluated, leading to the removal of the outer `if` function call.

   To ensure termination of the equal clause simplification phase, we make use of the measures used to prove termination of the functions in the SULFA property. For any recursive function f, let m be the measure used in the proof of termination of f. Every formal that occurs in m is a non-list argument. Since non-list arguments are constant in SULFA, we can associate a constant ordinal with any call of f by substituting formals with arguments. From the proof of termination of the function, we conclude that this ordinal decreases on every recursive call. If an ordinal fails to decrease

A. Creating Variables

```
(equal x0
       (bequiv (uandn 2 (concatn 1 a b))
               (if (uandn 1 a) (uandn 1 b) nil)))
```

⇒

```
(and
 (equal x1 (uandn 2 (concatn 1 a b)))
 (equal x2 (if (uandn 1 a) (uandn 1 b) nil))
 (equal x0 (bequiv x1 x2)))
```

B. Opening Functions

```
(equal x3 (concatn 1 a b))
```

⇒

```
(equal x3 (cons (car a) (concatn 0 (cdr a) b)))
```

C. Breaking Up Conses

```
(equal x3 (cons (car a) (concatn 0 (cdr a) b)))
```

⇒

```
(and (consp x3)
     (equal (car x3) (car a))
     (equal (cdr x3) (concatn 0 (cdr a) b)))
```

D. Breaking Up Ifs

```
(equal x1 (if (car x3) (uandn 1 (cdr x3)) nil))
```

⇒

```
(and (or (equal x1 (uandn 1 (cdr x3)))
         (not (car x3)))
     (or (equal x1 nil) (car x3)))
```

Figure 3: An illustration of the various simplifications on equal clauses.

on a recursive call, this recursive call has already been proven to be irrelevant and can therefore be replaced with `nil` without affecting the value of the expanded body.

3. When a `cons` expression occurs on the right-hand side of an equality the clause is broken into three equivalent clauses involving `consp`, `car`, and `cdr`, as shown in part C of Fig. 3.

4. When an `if` expression with a simple condition occurs on the right-hand side of an equality, its clause is broken into two equivalent clauses, as shown in part D of Fig. 3.

The above rules are repeatedly applied until reaching a fix point. At this point, each clause consists of a disjunction of expressions containing only calls to `equal`, `car`, `cdr`, and `consp`. Fig. 4 shows the formula output by the equal clause simplification phase when run on our example.

THEOREM 1. *A formula output by the equal clause simplification phase is valid only if its input is valid.*

New Variables:

```
x0=(bequiv (uandn 2 (concatn 1 a b))
           (if (uandn 1 a) (uandn 1 b) nil))
x1=(uandn 2 (concatn 1 a b))
x2=(if (uandn 1 a) (uandn 1 b) nil)
x3=(concatn 1 a b)
x4=(uandn 1 a)
```

Resulting Negated Conjunction:

```
(nand
 (consp x3)
 (equal (car x3) (car a))
 (equal (cdr x3) b)
 (or (equal x1 t)
     (not (car x3))
     (not (cadr x3)))
 (or (equal x1 nil)
     (not (car x3))
     (cadr x3))
 (or (equal x1 nil) (car x3))
 (or (equal x4 t) (not (car a)))
 (or (equal x4 nil) (car a))
 (or (equal x2 t) (not x4) (not (car b)))
 (or (equal x2 nil)
     (not x4)
     (car b))
 (or (equal x2 nil) x4)
 (or (equal x0 x2) (not x1))
 (or (equal x0 (not x2)) x1)
 (not x0))
```

Figure 4: Our example from Fig. 2 at the end of the equal clause simplification phase.

Proof Sketch. It is sufficient to prove that if an assignment of variables to values satisfies every clause input to the equal clause simplification phase, then there exists an assignment that satisfies every clause output. We prove this by showing that for each of the four simplification steps, if an assignment σ satisfies every clause input to the step then there exists an assignment π that satisfies every clause output by the step.

1. **Creating Variables** In this step, a new variable x, not occurring in any of the input clauses, is introduced. An expression E occurring in an input clause is replaced with x and a new clause `(equal` x E`)` is added. The assignment $\pi = \sigma \cup [x \rightarrow E/\sigma]$ therefore satisfies the output clauses.
2. **Opening Functions.** By a function's definitional axiom, the assignment σ produces the same value for a function call as it does for the function's instantiated body. Therefore, $\pi = \sigma$ satisfies the output clauses.
3. **Breaking Up Conses.** The value of `(equal x (cons a b))` under σ is Boolean equivalent to `(and (equal (car x) a) (equal (cdr x) b) (consp x))` under σ. Therefore $\pi = \sigma$ satisfies the output clauses.
4. **Breaking Up Ifs.** The value of `(equal x (if a b c))` is Boolean equivalent to `(and (or (equal x b) (not a)) (or (equal x c) a))` under σ. Therefore $\pi = \sigma$ satisfies the output clauses. □

4.3 Relevant Boolean Discovery

The next step in our conversion process is finding a set of Boolean expressions γ that are relevant to the validity of the formula. In this section, we refer to the members of the disjunction of a clause as *equal elements* and *non-equal elements*, depending on whether they contain an equality. Furthermore, we use the function `bfix`, which maps ACL2 constants into the Boolean domain using the following function:

```
(defun bfix (x) (if x t nil))
```

Initially γ is the empty set. If a non-equal element is `(consp X)` or `(not (consp X))`, where X is an ACL2 expression, then `(consp X)` is added to γ. If a non-equal element is X or `(not X)` then `(bfix X)` is added to γ. Fig. 5 shows the non-equal elements found in our example from Fig. 4, and the relevant Boolean expressions added to γ due to these elements.

The equal elements of clauses are traversed in a specific order, from the last clause to the first (this ensures that the variable on the left hand side of an equality is traversed after traversing every equality where it occurs on the right hand side). Equal elements with a constant right-hand side are ignored. For every other equal element we propagate relevant Boolean expressions involving the left-hand side of the equality into corresponding expressions involving the right-hand side. For example, the first equal element in Fig. 5 is `(equal x0 (not x2))`. Since the Boolean expression `(bfix x0)` is relevant, `(bfix x2)` is added to γ. If the right hand-side had been `(not (consp x2))` then we would have added `(consp x2)` to γ instead. When the right-hand side has no negation or call to `consp`, then many Boolean expressions can be propagated. For example, if `(consp (cadr x3))` and `(bfix (cdr x3))` were relevant, then `(consp (car b))` and `(bfix b)` would be added to γ after traversing the third equality in Fig. 5.

Non-Equal Elements:

`(consp x3)`	`(not (car x3))`
`(not (cadr x3))`	`(not (car x3))`
`(cadr x3)`	`car x3)`
`(not (car a))`	`(car a)`
`(not x4)`	`(not (car b))`
`(not x4)`	`(car b)`
`x4`	`(not x1)`
`x1`	`(not x0)`

Relevant Boolean Expressions Added To γ:

`(consp x3)`	`(bfix (car x3))`
`(bfix(cadr x3))`	`(bfix x4)`
`(bfix (car a))`	`(bfix (car b))`
`(bfix x0)`	`(bfix x1)`

Equal Elements:

1: `(equal x0 (not x2))`
2: `(equal x0 x2)`
3: `(equal (cdr x3) b)`
4: `(equal (car x3) (car a))`

Relevant Boolean Expressions Added To γ:
`(bfix x2)`

Figure 5: The non-equal and equal elements from our example and the relevant Boolean expressions after considering these elements

4.4 Boolean Variable Creation

The next phase of our algorithm creates a variable for each Boolean expression found to be relevant. First, new variables are chosen for each Boolean expression, as illustrated in part A of Fig. 6. Next these variables are substituted for the non-equal elements, as illustrated in parts B and C. Since a relevant Boolean expression exists for each non-equal element, every non-equal element is replaced.

The last of the non-Boolean variables are removed by replacing each clause with an equal element with clauses involving `iff` as shown in part D. One `iff` expression is created for each relevant Boolean expression involving the left hand side of the equality. For example we replace a clause involving `(equal x1 t)` with one involving `(iff (bfix x1) t)` since the only relevant expression involving `x1` is `(bfix x1)`. Then the Boolean relevant expression is replaced with its new variable—e.g. `(iff (bfix x1) t)` becomes `(iff y3 t)`.

If the right hand side of the equality is a variable, then a Boolean expression for the right hand side corresponding to the left hand side's relevant Boolean expression is guaranteed to be relevant, by virtue of the way relevant Boolean expressions are propagated from left to right through equalities. For example, if a clause existed containing `(equal x3 xN)` for some variable `xN`, then this clause would become three clauses containing `(equal (consp x3) (consp xN))`, `(equal (bfix (car x3)) (bfix (car xN)))`, and `(equal (bfix (cadr x3))`
`(bfix (cadr xN)))`. All of these Boolean expressions would then be replaced by Boolean variables.

Clauses that can be deduced from the following theorems are also added.

A. Choose New Variables

```
y0 := (bfix (car a))
y1 := (bfix (car b))
y2 := (bfix x0)
y3 := (bfix x1)
y4 := (bfix x2)
y5 := (consp x3)
y6 := (bfix (car x3))
y7 := (bfix (cadr x3))
y8 := (bfix x4)
```

B. Substitute consp non-equal elements

```
(or (consp x3) ...)
⇒
(or y5 ...)
```

C. Substitute bfix non-equal elements

```
(or  (equal x1 t)
     (not (car x3))
     (not (car (cdr x3)))
⇒
(or  (equal x1 t)
     (not y6)
     (not y7))
```

D. Break Equal Clauses into Components

```
(or (equal x1 t) (not y6) (not y7))
⇒
(or (iff y3 t) (not y6) (not y7))
```

E. Add List Axioms

```
(implies (bfix (car x3)) (consp x3))
⇒
(or (not y6) y5)
```

Figure 6: Examples illustrating the substitution of relevant Boolean variables for ACL2 variables.

```
(implies (bfix (car x)) (consp x))
(implies (bfix (cdr x)) (consp x))
(implies (consp x) (bfix x))
```

For each relevant Boolean expression we follow the chain of reasoning implied by the above theorems until we reach another relevant Boolean expression, at which point we add the implication as a new clause. For example the chain of reasoning

```
(and
 (implies (bfix (cadr x3)) (consp (cdr x3)))
 (implies (consp (cdr x3)) (bfix (cdr x3)))
 (implies (bfix (cdr x3)) (consp x3)))
```

leads to the addition of a clause for `(implies (bfix (cadr x3)) (consp x3))` in part E of Fig. 6. These implications are intended to ensure that a valid list structure can be constructed from any assignment of Boolean expressions to Boolean values that satisfies every clause.

The result of the Boolean creation phase on our example is shown in Fig. 7. The two clauses in the top row resulted from list axioms; each of the rest of the clauses corresponds to one of the clauses in Fig. 4.

THEOREM 2. *A formula output by the boolean variable creation phase is valid only if the formula input to it is valid.*

```
(nand
 (implies (not y6) y5)
 (implies (not y7) y5)
 y5
 (iff y6 y0)
 (iff y7 y1)
 (or (iff y3 t) (not y6)  (not y7))
 (or (iff y3 nil) (not y6) y7)
 (or (iff y3 nil) y6)
 (or (iff y8 t) (not y0))
 (or (iff y8 nil) y0)
 (or (iff y4 t) (not y8) (not y1))
 (or (iff y4 nil) (not y8) y1)
 (or (iff y4 nil) y8)
 (or (iff y2 y4) (not y3))
 (or (iff y2 (not y4)) y3)
 (not y2))
```

Figure 7: Our example from Fig. 4 after creating new variables for each Boolean component shown in Fig. 5.

Proof Sketch. It is sufficient to prove that if an assignment σ satisfies every clause input to the Boolean variable creation phase, then there exists an assignment π that satisfies every clause output. Such a π is the assignment that maps each variable y_i in the output of the phase with E_i/σ, where E_i represents the relevant Boolean expression corresponding to y_i. Clearly `(implies x (bfix x))`, `(implies (not x) (not (bfix x)))`, and for any function f, `(implies (equal x y) (iff (`f `x) (`f `y)))`. Therefore any clause in the output that is derived from a clause in the input is satisfied (clause generated by rules B, C, and D of Fig. 6). The remaining clauses in the output are merely instantiations of the valid theorems `(implies (bfix (car x)) (consp x))`, `(implies (bfix (cdr x)) (consp x))`, and `(implies (consp x) (bfix x))`. □

4.5 Iff Removal

The final phase in the conversion process removes each clause with an `iff` expression and replaces it with two clauses using the equivalence of `(or (iff x y) z)` and `(and (or (not x) y z) (or x (not y) z))`. We also remove clauses containing a literal and its negation, remove redundant literals, and remove constants. Fig. 8 shows the final formula output by our conversion algorithm on our example. At this point the expression is in negated CNF form and an external SAT solver can be used to prove it.

THEOREM 3. *A formula F_{out} output by the conversion algorithm is valid only if the original ACL2 property P is valid.*

Proof Sketch. If P is invalid, then the formula input to the equal clause simplification phase is invalid since the initial clausification phase produces a formula trivially equivalent to P. Therefore, by Lemma 1 and Lemma 2, the formula output by the Boolean variable creation phase is also invalid. The iff removal phase produces a formula trivially equivalent to its input; therefore, the formula output by the conversion algorithm is invalid. □

Theorem 3 ensures that it is safe to add SULFA properties resulting in unsatisfiable CNF formulas into ACL2's theorem database. We believe the converse of Theorem 3 also

Final Negated Conjunction:

```
(nand
 (or y6 y5)
 (or y7 y5)
 y5
 (or y6 (not y0))
 (or (not y6) y0)
 (or y7 (not y1))
 (or (not y7) y1)
 (or y3 (not y6)  (not y7))
 (or (not y3) (not y6) y7)
 (or (not y3) y6)
 (or y8 (not y0))
 (or (not y8) y0)
 (or y4 (not y8) (not y1))
 (or (not y4) (not y8) y1)
 (or (not y4) y8)
 (or y2 (not y4) (not y3))
 (or (not y2) y4 (not y3))
 (or y2 y4 y3)
 (or (not y2) (not y4) y3)
 (not y2))
```

Final Variables:

```
y0 := (bfix (car a))
y1 := (bfix (car b))
y2 := (bfix x0)
y3 := (bfix x1)
y4 := (bfix x2)
y5 := (consp x3)
y6 := (bfix (car x3))
y7 := (bfix (cadr x3))
y8 := (bfix x4)
```

Figure 8: Our example from Fig. 7 with all the iffs removed. At this point we have reached our target negated CNF form. We also show our final variables and how they relate to the original inputs.

holds; SULFA properties resulting in satisfiable CNF formulas are invalid. There are no severe consequences, however, if the converse does not hold; the counter-example generation algorithm, presented in the following section, will merely report that the SAT-based procedure has failed to prove or disprove the property.

4.6 Counter-Example Generation

If the SAT solver returns a satisfying instance for the final formula, a counter-example for the original ACL2 expression is generated, which maps free variables into constants under which the original expression evaluates to `nil`. To generate this counter-example we find a value for each free variable i in the original ACL2 expression using the following algorithm:

1. If `(bfix` i`)` is a relevant Boolean expression (an expression identified in the relevant Boolean discovery phase of the conversion algorithm) and its corresponding variable is $false$ then i gets `nil`.

2. Else if `(consp` i`)` is a relevant Boolean expression and its corresponding variable is $false$ then i gets `t`.

Table 1: Performance of the ACL2 Simplifier, BDD System, and SAT System

N	Example	ACL2	BDD	SAT
1	4 Adder-A	248s	0.0s	0.0s
2	32 Adder-A	****	0.3s	1.1s
3	200 Adder-A	****	48.6s	33.2s
4	32x6 Shift-0	54.8s	5.1s	8.6s
5	64x7 Shift-0	1710s	181s	55.6s
6	32x4 Add-Shift	2540s	2.7s	12.7s
7	64x6 Add-Shift	****	205s	230s
8	100 Digit-Inv	****	1.7s	3.5s

3. Else if there are no relevant Boolean expressions involving `(car` i`)` or `(cdr` i`)`, then i gets `t`.

4. Else return to step 1 with `(car` i`)` in place of i and recursively find a value `x` for `(car` i`)`. Repeat this process to recursively find a value `y` for `(cdr i)`. The value of i is then `(cons x y)`.

4.7 Optimization

There is one optimization we use that we have not yet discussed. This optimization is implemented during relevant Boolean expression discovery and Boolean variable creation phases and takes advantage of the fact that a singleton equal clause can be treated as a rewrite rule. If a clause `(equal` X Y`)` is encountered, where Y is an expression that involves neither negation nor `consp`, then `(equal` X Y`)` is removed. For later purposes X and Y are treated as indistinguishable. This optimization is implemented by sharing the data structures that store relevant expressions for X and Y.

This optimization removes the two singleton equal clauses from our example in Fig. 4. Furthermore, in part A of Fig. 6 the variables `y0` and `y1` are used to represent `(bfix (car x3))` and `(bfix (cadr x3))` rather than `y6` and `y7`.

5. RESULTS

In this section we compare the performance of our SAT system to two other approaches available for proving hardware theorems with the ACL2 theorem prover. One of these uses ACL2's built-in BDD system, and the other uses the ACL2 simplifier (without generalizing the problems and invoking induction). For this analysis we used zChaff version 2003.11.04 on a 3.0GHz Intel(R) $XEON^{TM}$ processor with 2 GB of RAM running ACL2 v3.0 on Allegro Common Lisp 7.0.

On eight examples, Table 5 compares the performance three approaches available in the ACL2 theorem prover, simplifying by brute-force, appealing to the BDD-based system, and using our SAT-based system (**** denotes results which require more than an hour or more memory than was available). In general, our SAT-based system performed better than the ACL2 simplifier and on par with the BDD system. It should be noted that the ACL2 simplifier is not designed to be used in a brute-force manner. Nevertheless, we wanted to show that ACL2 is capable of verifying SULFA properties automatically, albeit slowly. Furthermore, the BDD system is different from the new SAT system in many ways other than merely using BDDs. The BDD system is internal to ACL2, is not completely automatic, and is applicable to a larger set of ACL2 problems.

Table 2: Code Size Needed for the ACL2 Simplifier, BDD System, and SAT System

N	Example	Functions ACL2	BDD	SAT	
1	4 Adder-A	4	17	25	21
2	32 Adder-A	4	17	42	21
3	200 Adder-A	4	17	202	21
4	32x6 Shift-0	6	53	60	34
5	64x7 Shift-0	6	53	65	34
6	32x4 Add-Shift	4	58	71	44
7	64x6 Add-Shift	4	58	77	44
8	100 Digit-Inv	4	44	280	36

For each example, Table 5 gives the number of lines of code required to write the functions needed, followed by the number of lines required to specify and verify the property using the ACL2 simplifier, the BDD-based system, and the new SAT-based system. The SAT-based system requires fewer lines since properties can be expressed compactly, with no type hypotheses and since the proof is fully automatic. The ACL2 simplifier also can express the properties compactly, but requires a few more lines since the definitions must be rewritten as rewrite rules. The BDD system requires a substantial amount of code, because of the need for definitions to be rewritten, variable orderings to be specified, extra hypotheses stating that free variables are Boolean lists, and rewrite rules that push `if` expressions to the leaves of list structures.

5.1 Description of Examples

The first problem we consider is proving the associativity of a ripple carry adder. We give results for 4 bit, 32 bit, and 200 bit versions of the adder. On this example, the SAT system performs much better than the ACL2 simplifier and on par with the BDD system. The BDD system requires a user-specified variable ordering to achieve these results (the default ordering was poor).

The BDD system requires a substantial number of lines of code to prove the associativity of our adders. Most of this code is simple and has the potential to be automated, merely consisting of rewrite rules that replace function calls by their bodies. The final theorem though is also verbose since we must give a variable order for 200 variables and each variable is Boolean. The SAT system, by contrast, requires only a small statement of the theorem followed by a line instructing the theorem prover to use SAT.

We next implement a $N_{reg} \times N_s$ shifter which left shifts a N_{reg} bit register by the value of a N_s bit input. We first prove that 32x6 and 64x7 shifters output all zeros when given a large enough shift value. For a 32 by 6 bit shifter the BDD system is comparable to the SAT system. For any larger problems, however, the BDD system performs poorly, because there is no good variable ordering for this problem.

We also combine our shifter and ripple-carry adder to prove that shifting a value by x and then by y is equivalent to shifting by $x+y$ (with an extra bit of shift to handle carry).

In our final example, we prove the invariant mentioned in our introduction needed to show that the 100 digit version of the decimal counter described in Section 1 stays in its valid range. On this problem, the BDD system performs better than the SAT system. However, the BDD system requires more guidance on this problem than on any other problem due to the unique structure of the state of our 100 digit counter.

5.2 Performance Analysis

The table shown in Table 5.2 shows the minimum number of Boolean variables that would be required to represent the input SULFA property, the number of Boolean variables actually present in the final CNF formula, the number of clauses in the final CNF formula, the time spent during conversion, the time spent SAT solving, and the total time to solution.

Note there is also a trade-off between the conversion time and the SAT solving time—the more effort spent producing an easy problem for the SAT solver, the more time spent during conversion. Our tendancy has been to reduce the conversion time as much as possible, at the expense of the time spent SAT solving.

There is also a trade-off between the number of variables and the number of clauses in the final CNF formula. For example, one could minimize the number of variables by not creating variables during conversion. Such a methodology, however, would lead to an exponential number of clauses, which would slow down both the conversion time and the SAT solving time. Our rule of thumb has been to minimize the number of variables while keeping the number of clauses at most quadratic.

6. RELATED WORK

The complementary techniques of theorem proving and model checking have been combined previously on many occasions. Issabelle, for example, incorporates a number of decision procedures inspired by model checking, including a SAT-based decision procedure for propositional logic [18]. The general-purpose theorem prover PVS was designed with the combination of model-checking and theorem proving in mind [12]. Intel's FORTE system uses a HOL-based theorem prover built on top of an efficient procedure for symbolic trajectory evaluation [4]. The most general combination of theorem proving and model-checking of which we know is implemented by the SyMP model prover [2].

Fewer attempts, however, have been made to integrate model checking with ACL2. A BDD-based engine has been built into the theorem prover for some time [9], but uses a significantly different approach. The BDD-based engine is not fully automatic over any clearly defined subclass of ACL2 formulas, but instead, with some human guidance, attempts to operate on a wider set of ACL2 formulas.

Other work incorporating fully automatic tools with ACL2 includes the definition in ACL2 of a simple model checker for the Mu-Calculus [7] and the integration of ACL2 with UCLID [8]. The main difference between our work and this work is the relative simplicity of the logic used by SAT solvers, which allows us to create a more natural embedding into the ACL2 logic. We also have used UCLID [6] to verify some of our examples from Section 5, but found that it was not designed with low-level hardware models in mind. When used with the zChaff SAT solver, UCLID requires 6.33s to verify the invariant of the ten digit decimal counter and after ten minutes was still unable to verify the invariant of the hundred digit decimal counter. It may be possible to improve on these results, however, with the help of an experienced UCLID user.

Table 3: Performance Data

N	Example	Input Vars	Final Vars	Clauses	Conversion Time	Solving Time	Total Time
1	4 Adder-A	12	61	202	0.0s	0.0s	0.0s
2	32 Adder-A	96	509	1910	0.1s	1.0s	1.1s
3	200 Adder-A	600	3197	12158	0.4s	32.8s	33.2s
4	32x6 Shift-0	38	104	2677	8.6s	0.0s	8.6s
5	64x7 Shift-0	71	201	10469	55.6s	0.0s	55.6s
6	32x4 Add-Shift	40	181	3573	12.0s	0.8s	12.7s
7	64x6 Add-Shift	76	351	23025	137s	92.5s	230s
8	100 Digit-Inv	400	2705	39209	2.0s	1.6s	3.5s

7. CONCLUSION

We have developed an algorithm for converting a subset of ACL2 conjectures automatically into CNF so their validity may be determined by a SAT solver. This subset of ACL2 conjectures is rich enough to include properties about a finite number of cycles of an FSM. Both the finite state machine models and the properties about them are written in a compact, human-readable format, which is critical for integrating our technique with interactive theorem proving.

We have already begun to apply the SAT-based procedure as part of a larger hardware verification system described at CHARME 2005 [3]. In this verification system one uses ACL2 theorem prover to decompose a hardware property into multiple FSM properties, each of which involve only a finite number of steps of an FSM. As shown in our paper on SULFA, we have used this technique to verify a number of properties of the TRIPS processor [13]. The TRIPS processor is a prototype grid processor being designed at the University of Texas and IBM [1]. The SAT-based procedure automatically verifies ACL2 properties that would otherwise require human guidance to prove and it eases debugging of invalid conjectures by generating counter-examples for them in the ACL2 environment.

We believe that the general-purpose nature of the ACL2 language is key for specifying the kind of complex properties that should be checked in modern hardware and software systems. By integrating finite decision procedures with ACL2 as external tools, we can make much of the proof of these properties fully automatic.

8. REFERENCES

[1] Tera-op Reliable Intelligently adaptive Processing System, www.cs.utexas.edu/users/cart/trips.

[2] S. Berezin. *Model Checking and Theorem Proving: A Unified Framework*. PhD thesis, Carnegie Mellon University, 2002.

[3] W. A. Hunt, Jr. and E. Reeber. Formalization of the DE2 Language. In *Proceedings of the 13th Conference on Correct Hardware Design and Verification Methods (CARME 2005)*, pages 20–34. Springer, 2005.

[4] R. Jones, J. O'Leary, C.-J. Seger, M. Aagaard, and T. Melham. Practical Formal Verification in Microprocessor Design. *IEEE Design and Test of Computers*, 18:16–25, 2001.

[5] M. Kaufmann, P. Manolios, and J. S. Moore. *Computer Aided Reasoning: An Approach*. Kluwer Academic, 2000.

[6] S. K. Lahiri and R. E. Bryant. Deductive Verification of Advanced Out-of-Order Microprocessors. In *Proc. 15th Int. Conf. Computer Aided Verification (CAV'03)*, pages 341–354. Springer, 2003.

[7] P. Manolios. *Mechanical Verification of Reactive Systems*. PhD thesis, The University of Texas at Austin, 2001.

[8] P. Manolios and S. K. Srinivasan. Automatic Verification of Safety and Liveness for XScale-Like Processor Models Using WEB Refinements. In *DATE*, pages 168–175, 2004.

[9] J. Moore. Introduction to the OBDD Algorithm for the ATP Community. *Journal of Automated Reasoning*, 12(1):33–45, 1994.

[10] M. Moskewicz, C. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an Efficient SAT Solver. In *39th Design Automation Conference*, 2001.

[11] G. Nelson and D. C. Oppen. Simplification by Cooperating Decision Procedures. *ACM Transactions on Programming Languages and Systems*, 1(2):245–257, 1979.

[12] S. Rajan, N. Shankar, and M. K. Srivas. An Integration of Model Checking with Automated Proof Checking. In P. Wolper, editor, *Proc. 7th International Conference on Computer Aided Verification (CAV)*, volume 939, pages 84–97. Springer Verlag, 1995.

[13] E. Reeber and Warren A. Hunt, Jr. A SAT-Based Decision Procedure for the Subclass of Unrollable List Functions in ACL2 (SULFA). In *Proceedings of the Third International Joint Conference on Automated Reasoning (IJCAR)*. LNCS, 2006.

[14] D. Russinoff. A Mechanically Checked Proof of IEEE Compliance of a Register-Transfer-Level Specification of the AMD K7 Floating Point Multiplication, Division and Square Root Instructions. *LMS Journal of Computation and Mathematics*, 1:148–200, 1998.

[15] J. Sawada. *Formal Verification of an Advanced Pipelined Machine*. PhD thesis, University of Texas at Austin, 1999.

[16] G. Tseitin. On the complexity of derivation in propositional calculus. *Seminars in Mathematics*, 8, 1968.

[17] M. Velev and R. E. Bryant. Effective Use of Boolean Satisfiability Procedures in the Formal Verification of Superscalar and VLIW Microprocessors. *Journal of Symbolic Computation*, 35(2):73–106, 2003.

[18] T. Weber. Using a SAT solver as a fast decision procedure for propositional logic in an (lcf)-style theorem prover. In J. Hurd, E. Smith, and A. Darbari, editors, *Proc. TPHOLS*, pages 180–189, 2005.