Certifying Compositional Model Checking Algorithms in ACL2*

Sandip Ray† John Matthews‡ Mark Tuttle§

July 7, 2003

Abstract

We prove the soundness of a compositional model checking algorithm using ACL2. The algorithm uses conjunctive and cone of influence reductions to reduce a large model checking problem into a collection of smaller problems, and we prove the soundness of the composition of these reductions. The algorithm checks properties specified in Linear Temporal Logic (LTL), but the ACL2 logic does not allow us to express either the classical semantics of LTL or the classical soundness proofs for these reductions. We discuss an approach to circumvent such restrictions in ACL2. We also propose enhancements to ACL2 that would make similar proof attempts easier in future. Finally, we argue in favor of providing better integration of ACL2 with external model checkers and other analysis tools.

1 Introduction

Theorem proving and model checking are powerful techniques for verifying system designs. Modern theorem provers such as ACL2 [17, 16], HOL [8], and PVS [26], and model checkers such as Cadence SMV [23], have produced amazing results. Model checking, in particular, is widely used in the industry today. It has the advantage of being completely automated, but it suffers from the state explosion problem, limiting its application to designs of relatively small size. There are a number of approaches to dealing with this state explosion problem, but their implementations are nontrivial and are themselves prone to error. In this paper, we use the ACL2 theorem prover to prove the correctness of a particularly promising approach called compositional model checking, and our experience motivates enhancing ACL2 in several ways.

1.1 Compositional model checking

The essence of model checking is easy to understand [4]. First we model the system as a state machine, then we express the property to be checked in a suitable logic, and then we verify that the property is true in every reachable state of the system. To be more precise, we model the system as a Kripke structure, which is a state machine together with a set of primitive propositions, where each state maps each primitive proposition to true or false.

We use Linear Temporal Logic (LTL) to state the property being verified [4]. LTL consists of the usual boolean operators and temporal path operators such as G and F, sometimes read "always" and "eventually." A formula is true or false of a path through a Kripke structure, and a formula is true of a Kripke structure if and only if it is true of every path through the Kripke structure. The formula $G\varphi$ is read "always φ" and is true of a path if φ is true at every state in the path. The formula $F\varphi$ is read "eventually φ" and is true of a path if φ is true at some state in the path. These temporal operators combine to form more interesting formulas such as $GF\varphi$, read "always eventually φ," which is true of a path if at every state there

*This work was performed at HP Labs Cambridge.
†Department of Computer Sciences, The University of Texas at Austin, Austin, TX 78712; sandip@cs.utexas.edu.
‡Department of Computer Science and Engineering, OGI School of Science and Engineering, Oregon Health and Science University, Beaverton, OR 97006; johnm@cse.ogi.edu.
§HP Labs Cambridge, One Cambridge Center, Cambridge, MA 02142; mark.tuttle@hp.com.

is a later state satisfying φ. Given this interpretation, the formula $GF\varphi$ can only be true of infinite paths, and, in fact, all of the paths we consider are infinite. More and more, LTL is being chosen over other logics to specify industrial systems, and we note that the highly-regarded model checker Cadence SMV [23] has recently switched to LTL.

Compositional model checking is a promising approach to improving the performance of model checkers. The idea is to take a formula φ and a Kripke structure M, to generate a collection of simpler formulas $\varphi_1, \ldots, \varphi_\ell$ and Kripke structures $M_1, \ldots, M_\ell$, and to reduce the problem of checking that φ is true of M to the simpler problems of checking that φ_i is true of M_i for each i. Two examples of such reductions are conjunctive and cone of influence reduction. *Conjunctive reduction* reduces the problem of checking that $\varphi_1 \& \varphi_2$ is true of M to the problems of checking that φ_1 is true of M and that φ_2 is true of M. *Cone of influence reduction* is based on the idea that if the property φ refers only to variables in V, then we can reduce the problem of checking that φ is true of M to the problem of checking that φ is true of a possibly much simpler N whose states give values only for a small subset of variables containing V. Composing these two reductions can have a dramatic effect. If φ_1 and φ_2 refer to disjoint and small sets of variables, then the reduced Kripke structures N_1 and N_2 can be dramatically smaller than M, and checking that φ_1 and φ_2 are true of N_1 and N_2 can be much faster than checking that φ is true of M.

The classical soundness proof [4] for the cone of influence reduction is based on the notions of *bisimulation* and *corresponding paths*. The proof proceeds by showing that given two bisimilar Kripke structures M_1 and M_2 and a formula φ, for every path σ_1 through M_1 there exists a corresponding path σ_2 through M_2 such that φ is true of σ_1 iff φ is true of σ_2, and vice versa. It follows that φ is true of M_1 iff φ is true of M_2, and we can check that φ is true of M_1 and M_2 by doing model checking on either Kripke structure. The proof then concludes by showing that the original Kripke structure M and the simpler Kripke structure N returned by the cone of influence reduction are bisimilar, thus justifying that cone of influence is a sound reduction.

1.2 Soundness proofs in ACL2

We chose to implement and verify compositional reduction algorithms using the ACL2 theorem-prover. The choice of ACL2 for this verification stems from our desire to reason about executable tools rather than mathematical models. Among the state-of-the-art theorem-provers, ACL2 provides the best support for executability. The logic of ACL2 is executable; in fact the executable portion of ACL2 is just a subset of Common Lisp. Hence in ACL2, verification of algorithms is synonymous with verification of actual implementation code.

There are, however, inherent difficulties in doing this work in ACL2. The basic problem stems from the weakness of the ACL2 logic. In particular, the ACL2 logic permits little support for modeling or reasoning about infinite sequences. For example, if a sequence is modeled by a list, as is customary in ACL2, then it is easy to prove that the list is finite. In fact, it is easy to prove in ACL2, that for every list l there exists a list m of length strictly greater than l. However, as we pointed out, the traditional definition of semantics of LTL is in terms of infinite paths, which are sequences of states. Since there is no straightforward way of mapping infinite sequences into ACL2 objects, we cannot model the traditional semantics of LTL in ACL2.

The standard ACL2 workaround for dealing with infinite objects is to use encapsulated functions. In our example, we can think of *path* as an encapsulated function that takes a natural number i and returns the i-th state in the sequence. Sumners [30] used this approach to model bisimulation arguments in ACL2. However, we could not use of encapsulated functions in our work to model the semantics of LTL, since it does not let us reason about corresponding paths of bisimilar Kripke structures M_1 and M_2. To see why, consider defining the function *ltl* defining the truth of an LTL formula using an encapsulated function to represent the path. We need to prove that for every path through M_1 there is a corresponding path through M_2 such that *ltl* returns the same result for both paths. Using an encapsulated function to model the path in the definition of *ltl*, however, and instantiating this encapsulated function to these two paths, results in two separate instances of the function *ltl*. In contrast, we need to prove that the same function *ltl* returns the same result when applied to the two paths, so we need to be able to define *ltl* with an explicit path argument. We note that Manolios, Namjoshi, and Summers have used bisimulation arguments to verify pipelined machines, distributed protocols, and concurrent programs in ACL2 [20, 22, 31], but they argue this equivalence of bisimilar models outside of ACL2, and our proofs require having this equivalence installed as a theorem in ACL2 rather than using them in meta-theory.

However, note that even a simple axiomatization of infinite paths in ACL2 does not allow us to model the traditional semantics. To see this, assume for the moment that we have extended ACL2 with axioms modeling infinite paths. Consider the problem of defining the semantics of LTL in such an extended ACL2. The most natural approach is to define an auxiliary function *ltl* that maps a formula and a path to true or false. We can then define what it means for an LTL formula to be true of a given Kripke structure by universally quantifying over all of the structure's initial paths, and checking that *ltl* is true for each one. The natural definition of *ltl* is a collection of recursive definitions for formulas such as $F\varphi$ saying

$$\mathit{ltl}(F\varphi, \mathit{path}) = \exists i . \mathit{ltl}(\varphi, \mathit{suffix}(i, \mathit{path}))$$

where $\mathit{suffix}(i, \mathit{path})$ is the i^{th} suffix of *path* obtained by deleting the first $i-1$ states from *path*.

Unfortunately, ACL2 does not permit recursive function definitions with quantifiers in the body. The traditional ACL2 work-around to this limitation is to define a function that explicitly computes the i required above. This function would have to iterate down the path to find the appropriate i^{th} suffix, but we cannot guarantee that this function would terminate since the path is infinite, and hence this function would not be admissible in the logic of ACL2. We conclude that the semantics of LTL can be defined in ACL2, but not the standard definition, and not a definition that allows us to reuse the standard soundess proofs.

Manolios [20, 21] faced a similar problem with the semantics of μ-calculus. He solved this problem by defining a model checker for μ-calculus in ACL2 and declaring the semantics of μ-calculus to be what the model checker returns. This is not a viable solution for LTL, however, since the complex tableau construction used in LTL model checkers is so hard to implement. A model checker for μ-calculus is sufficiently simple to make it reasonable to define its semantics with a model checker, but this is not true for LTL.

Our solution to the problem of modeling the semantics of LTL is to realize that there is an infinite path through a Kripke structure satisfying φ if and only if there is an eventually periodic path satisfying φ. An *eventually periodic path* is one that can be decomposed into a finite path followed by a finite cycle (that is repeated forever). This finite representation of an infinite path allows us to define the semantics of LTL in a relatively natural way, but it dramatically complicates our soundness proofs.

1.3 Lessons learned

We have successfully proven the soundness of compositional model checking for LTL formulas with ACL2. Based on our experience, however, we draw two conclusions.

We consider it important to reconsider ACL2's restrictions on the use of recursion and quantification, and to make it easier to specify properties of infinite sequences in ACL2. Our use of eventually periodic paths to define the semantics of LTL has two serious drawbacks. First, it is not the standard definition, which reduces confidence in its validity. Second, it complicates the soundness proofs, forcing us to take the simple and elegant classical proofs of soundness and encrypt them with difficult and annoying arguments. The ACL2 community uses the concept of *conservativity* [18] to justify the restrictions on recursion and quantification, but we will argue in Section 6 that it might be worth giving up on conservativity when the analysis of difficult, practical problems demands an extension with less stringent restrictions on recursion and quantification.

We also consider it to be extremely important to consider extending ACL2 to integrate external decision procedures like model checkers with ACL2. Our function defining the semantics of LTL cannot be evaluated efficiently and run on real systems, and implementing an LTL model checker in ACL2 is a bad idea for several reasons. First, it would be a lot of work to implement a model checker in ACL2 that is competitive with the model checkers currently on the market such as SMV [23]. Second, we contend that the core model checking routines found in state-of-the-art implementations are often simple enough or have been used enough to be trusted, just as we trust a theorem-proving system like ACL2. Errors in model checkers are more likely to appear in reductions than in the core routines, and we should be more worried about verifying the correctness of the reductions using a tool like ACL2 than verifying the correctness of the core model checking implementations. We discuss our solution to this problem in Section 5, with some recommendations for constructing a less monolithic ACL2.

1.4 Related Work

The idea of using a theorem prover to mechanically verify a meta-theory of model checking has been used successfully earlier in order to decompose a large model checking problem into smaller, more managable pieces. For example, Chou and Peled [3] use the HOL theorem prover [8] to formally verify a partial order reduction algorithm. Further, HOL has been successfully used in mechanical specification and verification of Temporal Logic of Actions [32]. HOL and PVS [26] provide facilities for integration with model checking procedure [29, 28]. Verification of temporal properties of finite state systems using a combination of theorem proving (using HOL) and model checking (using SPIN [13]) has been reported in a number of papers [2]. In the ACL2 world, Manolios [21, 20] report on the implementation of a μ-calculus model checker in ACL2. He verifies that the model checker returns a fixpoint.

The crucial difference between our work and the related research involving mechanical verification of model checking and decomposition of finite state machines arises from our motivation to verify actual implementation code, which can then be used to analyze concrete descriptions of state machines. Whereas the previous work was mostly involved in verification of mathematical models, without concern for executability, we intend to apply the compositional algorithm on real-world model checking problems. This motivation has led us to choose a logic like ACL2 which supports efficient executability. We note that the previous model checking work in ACL2, namely [21, 20], do involve implementing model checkers in ACL2. As we pointed out, while μ-calculus model checker allows for a simpler specification, allowing Manolios [21] to declare the model checker to also serve as the specification for μ-calculus model checking, we cannot apply a similar technique for LTL. In particular, an implementation of an executable model checker for LTL is complicated enough, requiring us to distinguish between the implementation and a specification of LTL.

The remainder of the paper is organized as follows. In Section 2, we provide an overview of LTL. In Section 3, we give the reduction algorithms, and the high-level arguments for their soundness. In Section 4, we show how we model and verify these reductions in ACL2, the problems imposed by the restrictions of ACL2, and how we circumvent these problems. In Sections 5 and 6, we discuss our proposed extensions to ACL2. Finally, in Section 7, we provide some concluding remarks.

2 Linear Temporal Logic

We begin with a review of Linear Temporal Logic (LTL), and we refer the reader to the book by Clarke, Grumberg, and Peled [4] for a nice presentation of the rich body of literature on LTL model checking.

2.1 Syntax

An LTL formula is constructed from the boolean constants true and false; a set AP of atomic propositions; the boolean connectives `~`, `+`, and `&`;[1] and temporal operators `G`, `F`, `X`, `U`, and `W`. The ACL2 predicate **`ltl-formulap`** recognizes LTL formulas. It will soon be convenient to equate an atomic proposition with a boolean variable giving the truth of the proposition in a given state, so our ACL2 functions refer to propositions as variables.

```
(defun ltl-constantp (f)
  (or (equal f 'true)
      (equal f 'false)))
```

[1] Throughout this paper, we use the symbol `~` for the boolean `NOT` operator, `+` for boolean `OR` operator and `&` for boolean `AND` operator.

```
(defun ltl-variablep (f)
  (and (symbolp f)
       (not (memberp f '(+ & U W X F G)))))

(defun ltl-formulap (f)
  (cond ((atom f) (or (ltl-constantp f)
                      (ltl-variablep f)))
        ((equal (len f) 3)
         (and (memberp (second f) '(+ & U W))
              (ltl-formulap (first f))
              (ltl-formulap (third f))))
        ((equal (len f) 2)
         (and (memberp (first f) '(~ X F G))
              (ltl-formulap (second f))))
        (t nil)))
```

2.2 Semantics

The truth of an LTL formula is defined using Kripke structures. Given a set AP of atomic propositions, a *Kripke structure* is a tuple $\langle S, R, L, S_0 \rangle$ where

1. S is a set of *states*,
2. $S_0 \subseteq S$ is the set of *initial states*,
3. $R \subseteq S \times S$ is the *transition relation*, which must be total, meaning that for every $s \in S$ there is an $s' \in S$ such that $R(s, s')$, and
4. $L : S \to 2^{AP}$ is a labeling function that maps each state to the set of atomic propositions that are true in that state.

A *path* in a Kripke structure is an infinite sequence $\pi = s_0 s_1 s_2 ...$ such that $s_i \in S$ and $R(s_i, s_{i+1})$ both hold for every $i \geq 0$. Given such a path π, we define π^i to be the suffix of π starting at s_i. If π is a path, then π^i is a path for every $i \geq 0$, and $\pi^0 = \pi$. We define π to be an *initial path* if π is a path and s_0 is an initial state.

We define what it means for a path $\pi = s_0 s_1 ...$ of a Kripke structure M to satisfy a formula f by induction:

1. If f is `true` then π satisfies f.
2. If f is `false`, then π does not satisfy f.
3. If f is an atomic proposition, then π satisfies f if and only if $f \in L(s_0)$.
4. If f is of the form ~φ, then π satisfies f if and only if π does not satisfy φ.
5. If f is of the form (φ_1 `&` φ_2), then π satisfies f if and only if π satisfies φ_1 and π satisfies φ_2.
6. If f is of the form (φ_1 `+` φ_2), then π satisfies f if and only if π satisfies φ_1 or π satisfies φ_2.
7. If f is of the form (`F` φ), then π satisfies f if and only if π^i satisfies φ for some $i \geq 0$.
8. If f is of the form (`G` φ), then π satisfies f if and only if π^i satisfies φ for every $i \geq 0$.
9. If f is of the form (`X` φ), then π satisfies f if and only if π^1 satisfies φ.

10. If f is of the form $(\varphi \texttt{ U } \psi)$, then π satisfies f if and only if π^i satisfies ψ for some $i \geq 0$ and π^j satisfies φ for every $0 \leq j < i$.

11. If f is of the form $(\varphi \texttt{ W } \psi)$ then π satisfies f if and only if π^i satisfies ψ for some $i \geq 0$ and π^j satisfies φ for every $0 \leq j < i$, or if π^j satisfies φ for every $j \geq i$.

The Kripke structure M satisfies f if and only if π satisfies f for every initial path π of M.

We model a finite Kripke structure in ACL2 as an object with components named variables (the atomic propositions), states, initial-states, and transition. A state maps each variable to true or false (so the labeling function in the formal definition of a Kripke structure just maps a state to the variables that are true in that state).[2] A transition maps each state to the list of possible next states. The following predicate `circuit-modelp` recognizes Kripke structures. The predicate refers to other predicates whose definitions are given in the supporting material.[3]

```
(defun circuit-modelp (m)
  (and (only-evaluations-p (states m) (variables m))
       (all-evaluations-p (states m) (variables m))
       (strict-evaluation-list-p (variables m) (states m))
       (only-all-truths-p (states m) m (variables m))
       (only-truth-p (states m) m)
       (label-subset-vars (states m) m (variables m))
       (transition-subset-p (states m) (states m) (transition m))
       (subset (initial-states m) (states m))
       (consp (states m))
       (next-states-in-states m (states m))))
```

We also model what it means for a Kripke structure to satisfy a formula as an ACL2 predicate, but modeling the semantics of LTL in ACL2 is a nontrivial contribution of this paper, so let us leave this definition until Section 4.

2.3 Finite State Systems

The mathematical model of a Kripke structure given above is a bit cumbersome, and we need a more compact representation upon which our reduction algorithms can operate. We define a finite state system to be a set of boolean *variables*, one set of *equations* for each variable, and a set of *initial states*. Each initial state defines one of the ways values can be assigned to variables at reset. Each equation associated with a variable v is a formula that gives one possible value for v in the next state in terms of the values of variables in the current state. We note that there are many equivalent ways to represent a finite state system in ACL2. We could associate a single formula with v instead of a set of formulas by taking their disjunction, and Hunt [14] has formulated even better representations. We want to focus on proofs in this work, and not models, and the primary advantage of our representation is that it is very easy to generate a mathematical Kripke structure from a finite state system.

The ACL2 predicate `circuitp` recognizes a finite state system.[4] This predicate depends on another predicate `consistent-equation-record-p` that recognizes syntactically correct equations. We assume that

[2] In this work, we freely use the book `records.lisp` provided with the ACL2 distribution. The operator `<-` and `->` are macros expanding into `g` and `s` functions respectively, and provide a convenient way of retrieving and updating records. The symbol `>_` is a macro for updating the empty record.

[3] We note that the predicate does not consider a Kripke structure with no states to be a valid Kripke structure. It is possible to extend the theorems so that this restriction is removed. However, such fine-tuning of the recognizing predicates is not a part of this work.

[4] The name `circuitp` comes from the fact that we were initially targeting this work to model hardware circuits. Indeed, the restriction of the range of variables to booleans was derived initially from the authors' desire to model hardware. However, there is no need for this work and terminology to be restricted to hardware designs, since any system with variables ranging in

the formulas appearing in these equations are expressed in terms of the variables in a set V, the constants `T` and `NIL`, and the operators `&`, `+`, and `~`, although there is nothing in our work that depends on using such a simple set of operators.

```
(defun find-variables (equation)
  (cond ((and (atom equation) (not (booleanp equation)))
         (list equation))
        ((and (equal (len equation) 3) (memberp (second equation) '(& +)))
         (set-union (find-variables (first equation))
                    (find-variables (third equation))))
        ((and (equal (len equation) 2) (equal (first equation) '~))
         (find-variables (second equation)))
        (t nil)))

(defun-sk consistent-equation-record-p (vars equations)
  (forall (v equation)
          (implies (and (uniquep vars)
                        (memberp v vars)
                        (memberp equation (<- equations v)))
                   (subset (find-variables equation) vars))))


(defun circuitp (C)
  (and (only-evaluations-p (initial-states C) (variables C))
       (strict-evaluation-list-p (variables C) (initial-states C))
       (uniquep (variables C))
       (cons-list-p (variables C) (equations C))
       (consistent-equation-record-p (variables C) (equations C))))
```

This predicate `circuitp` insists, for example, that initial states assign only `T` or `NIL` to a variable, and assign a value to each variable; and that the equations assigning values to variables are syntactially correct.

We can now define a function `create-kripke` that takes a finite state system S and computes the corresponding Kripke structure M as follows:

1. The variables of M are the variables of S.
2. The states of M are all possible assignments of `T` and `NIL` to the variables of S.[5]
3. The initial states of M are the initial states of S.[6]
4. The transitions of M are computed from the equations of S as follows. An *equation record* is a collection of equations, one for each variable. Two states s and s' are related by an equation record if, for each

a specific finite domain can be modeled by a system in which all variables are only booleans. For the purpose of this paper, we use the terms *circuit* and *finite state system* interchangeably.

[5]Technically, the number of states we can attain by simply enforcing this condition is infinite, since every assignment that assigns a boolean to the variables in V, and any ACL2 object to any variable not in V, qualifies as a different "state" according to this definition. To alleviate this problem, we define an "equivalence relation" `evaluation-eq` with respect to assignments. The equivalence relation does not distinguish between two assignments that produce the same value for every variable in V. We can then say that a given set of assignments contains all the assignments to the variables in V, if given an arbitrary assignment D to the variables in V, there exists an assignment in the set that is `evaluation-eq` to V.

[6]Technically, a subset function guaranteeing to test that the initial states are a subset of the set of states should check membership using the relation `evaluation-eq` instead of equality. On the other hand, notice that we have never enforced that the set of states needs to be unique. Hence, to simplify matters, we simply consider the union of the set of all evaluations to variables with the set of initial states as our set of states.

variable v, the value of v in s' is the value of the equation for v evaluated in s. An equation record is *consistent* if, for each variable v, the equation for v is one of the equations for v in S. The transitions of M map each state s to the set of states s' such that s and s' are related by a consistent equation record.

The following theorem states that `create-kripke` will produce a well-formed Kripke structure from a well-formed finite state system.[7]

```
(DEFTHM create-kripke-produces-circuit-model
  (implies (circuitp C)
           (circuit-modelp (create-kripke C))))
```

3 Reductions

In this section, we give the model checking reduction that we want to prove is sound. This reduction is the composition of two simple reductions called *conjunctive reduction* and *cone of influence reduction.* We state the complete reduction in Section 3.3, but in this paper we focus on proving the soundness of the cone of influence reduction. We define the cone of influence reduction in Sections 3.1 and sketch the classical soundness proof for this reduction in Section 3.2.

3.1 Cone of Influence Reduction

Informally, cone of influence reduction is simply elimination of "dead code" or redundant state variables. Let V be the set of variables in the system, and recall that the system is defined by associating with each variable a set of equations. Each equation is a boolean function of the variables in V. Suppose we are now given a set of variables $V' \subseteq V$ that are of interest with respect to the required specification. In our parlance, that implies that the only variables ever accessed by the LTL formula we are checking are variables in V'. We would like to simplify the description of the system by referring only to the variables in V'. However, the values of the variables in V' might depend on values of variables not in V'. The cone of influence for V' is the set of variables whose values could affect the value of a variable in V'. We want to compute this cone of influence, and use it to reduce the size of the system description.

Given a finite state system, the cone of influence of V' is the minimal set C of variables such that:

- $V' \subseteq C$.
- If $v_i \in C$ and there is an equation corresponding to v_i that depends on v_j, then $v_j \in C$.

Cone of influence reduction is based on the idea that if an LTL formula φ refers only to the variables in V' of a finite state system F, then we can reduce the problem of checking the formula φ on the Kripke structure M corresponding to finite state system F to the problem of checking the formula φ on the Kripke structure corresponding to a much simpler finite state system F', constructed as follows:

- The set of variables in F' is the cone of influence C.
- The set of equations corresponding to each variable v in F' is the set of equations corresponding to v in F.

[7] We note that `create-kripke-produces-circuit-model` is not a trivial theorem to prove. The reason is the limited support that ACL2 provides for quantifiers. Functions like `all-evaluations-p` used by the predicate `circuit-modelp` use quantifiers mainly because quantifiers make the specifications clearer. On the other hand, proofs turn out to be harder since for every existential quantifier, we need to virtually construct an instance satisfying the quantified property.

- The set of initial states of F' is formed from the initial states of F by projecting out only those assignments to variables in C.

Notice that if C has a small cardinality relative to the number of variables in F, then the Kripke structure corresponding to F' will have many fewer states than the Kripke structure corresponding to F. Hence model checking can be much more efficient on the reduced finite state system. In practice, cone of influence reduction is routinely applied before model checking finite state systems.

To implement cone of influence reduction in ACL2, we define functions to augment the set V' to the set C. The approach is to explore the equations corresponding to the variables in V' to obtain more variables. We augment V' with these additional variables, and we repeat until we converge. The function `find-all-variables` achieves that goal. In this function `variable` and `vars` correspond to V and V'.

```
(defun find-variables* (equation-list)
  (if (endp equation-list) nil
    (set-union (find-variables (first equation-list))
                      (find-variables* (rest equation-list)))))

(defun find-all-variables-1-pass (vars equations)
  (if (endp vars) nil
    (set-union (find-variables* (<- equations (first vars)))
                      (find-all-variables-1-pass (rest vars) equations))))

(defun find-all-variables (vars variables equations)
  (declare (xargs :measure (nfix (- (len variables) (len vars)))))
  (if (or (not (uniquep variables))
          (not (uniquep vars))
          (not (subset vars variables)))
      vars
    (let ((new-vars (set-union (find-all-variables-1-pass vars equations)
                               vars)))
      (if (not (subset new-vars variables)) nil
        (if (set-equal vars new-vars) vars
          (find-all-variables new-vars variables equations))))))
```

Finally, the function `cone-of-influence-reduction` creates the reduced finite state system F'.

```
(defun cone-of-influence-reduction (C vars)
  (let ((variables (cone-variables vars C)))
   (>_ :variables variables
       :initial-states (corresponding-states (initial-states C) variables)
       :equations (find-all-equations variables (equations C) ()))))
```

3.2 Soundness of Cone of Influence Reduction

The standard soundness proof for the cone of influence reduction involves arguments about *bisimulation*. Given two Kripke structures $M = \langle S, R, S_0, L\rangle$ and $M' = \langle S', R', S'_0, L'\rangle$ with respect to the same set of abstract propositions AP, a relation $B \subseteq S \times S'$ is called a *bisimulation relation* if and only if for any $s \in S$ and $s' \in S'$ such that $B(s, s')$ the following conditions hold:

1. $L(s) = L'(s')$.

2. For every state $s_1 \in S$ such that $R(s, s_1)$ there exists $s'_1 \in S'$ such that $R'(s', s'_1)$ and $B(s_1, s'_1)$.

3. For every state $s'_1 \in S'$ such that $R'(s', s'_1)$, there exists $s_1 \in S$ such that $R(s, s_1)$ and $B(s_1, s'_1)$.

The two Kripke structures M and M' are said to be *bisimulation equivalent* (denoted $M \equiv M'$) if there exists a bisimulation relation B such that for every initial state $s_0 \in S_0$ there is an initial state $s'_0 \in S'_0$ satisfying $B(s_0, s'_0)$, and for every initial state $s'_0 \in S'_0$ there is an initial state $s_0 \in S_0$ satisfying $B(s_0, s'_0)$. Given a bisimulation relation B between two Kripke structures M and M', we say that two paths $\pi = s_0 s_1 s_2 ...$ in M and $\pi' = s'_0 s'_1 s'_2 ...$ in M' *correspond* if and only if for every $i \geq 0$, $B(s_i, s'_i)$.

We now discuss two lemmas that enable us to use bisimilarity arguments to prove soundness of reductions for LTL model checking.

Lemma 1 *Let s and s' be two states such that $B(s, s')$. Then for every path starting from s there is a corresponding path starting from s' and for every path starting from s' there is a corresponding path starting from s.*

Lemma 2 *Let f be an LTL formula and let π and π' be corresponding paths. Then π satisfies f if and only if π' satisfies f.*

From Lemmas 1 and 2, it is easy to see that the following theorem holds.

Theorem 1 *If $M \equiv M'$, then for every LTL formula f, M satisfies f if and only if M' satisfies f.*

Both Lemmas 1 and 2 are well-known results, and can be found in text-books on model-checking [4].[8] The soundness of cone of influence reduction now follows by showing a bisimulation relation between the Kripke structure for the original system and the Kripke structure for the reduced system.[9] We can exhibit such a bisimulation relation as follows. Assume $v_1, ...v_n$ are variables in V and, without loss of generality, $v_1, ..., v_k$ are variables in C, with $k \leq n$. Recall also that a state for a finite state system is obtained by a boolean assignment to each of the variables in the system. Given a state $s = \langle d_1, ..., d_n \rangle$ assigning booleans to the variables in V and a state $s' = \langle \hat{d_1}, ...\hat{d_k} \rangle$ assigning booleans to the variables in C, we say that $B(s, s')$ holds if $d_i = \hat{d_i}$ for all $i \leq k$. It is easy to prove the claim that B is a bisimulation relation between the two models for labeling restricted to C, and the Kripke structures produced from the original system and its cone of influence are equivalent with respect to this bisimulation relation. (For details of the proof see [4].) Hence any formula is true for the structure created from the reduced system if and only if it is true of the original system.

3.3 Compositional Reduction

We now present a simple compositional reduction algorithm based on conjunctive and cone of influence reductions. We note here, that this algorithm is for the purpose of demonstration alone. It is possible to implement more sophisticated compositional reduction strategies. However, in this paper, our focus is on techniques to verify reductions in ACL2, rather than implement sophisticated reductions. We, therefore, use

[8] [4] presents a more general version of Lemma 2. The original theorem is stated in terms of formulas written in CTL*, a logic that is more expressive than LTL. However, the version we provided in this paper follows immediately from the theorem in [4], and we decided not to confuse the reader with CTL* for this work.

[9] We should note that the two structures cannot be bisimilar according to the original definition of bisimulation we presented here, since they differ in the set of abstract propositions (AP). Recall that the abstract propositions for Kripke structures representing finite state systems has been described as the set of variables in the system. However, we can look at the set of variables in the cone of influence and consider that set to be the set of abstract propositions. To define bisimulation based on such a notion, we need that two bisimilar states agree in their labels restricted to the variables in the cone of influence. In order to be able to reason more generally with such a notion, we need to define a notion of bisimulation with respect to a collection of variables `vars`. Under such a notion of bisimilarity, the theorem above transforms to a statement that if an LTL formula contains only the variables in `vars` and two Kripke structures are equivalent with respect to `vars` then the formula is true of one structure if and only if it is true of another. The proof of such a statement is exactly in the line of Theorem 1.

this algorithm as a simple but illustrative example of the roadblocks one would face in verifying compositional reductions.

Our compositional algorithm first decomposes the LTL property using conjunctive reduction. This produces a collection of verification problems using the "simpler" formulas. This is shown in the following ACL2 function.

```
(defun reduce-problem-conjunction (f C)
  (if (and (equal (len f) 3)
           (equal (second f) '&))
      (append (reduce-problem-conjunction (first f) C)
              (reduce-problem-conjunction (third f) C))
    (list (list f C))))
```

We then apply cone of influence reduction on each of the simpler sub-problems. The reduction is shown in the following function `reduce-problem-cone*`.

```
(defun reduce-problem-cone (f C)
  (let ((vars (create-restricted-var-set f)))
    (cone-of-influence-reduction C vars)))

(defun reduce-problem-cone* (list)
  (if (endp list) nil
    (cons (list (first (first list))
                (reduce-problem-cone (first (first list)) (second (first list))))
          (reduce-problem-cone* (rest list)))))
```

Finally, our compositional algorithm is shown by the following function.

```
(defun compositional-reduction (C f)
  (let ((list (reduce-problem-conjunction f C)))
    (reduce-problem-cone* list)))
```

4 Verification

Can our compositional reduction algorithm be verified to be sound in ACL2? A compositional reduction procedure takes an LTL formula φ and a finite state system F, and produces a list of pairs. Each pair in the list is of the form $\langle \varphi_i, F_i \rangle$, where φ_i is an LTL formula and F_i is a finite state system. The soundness theorem for a reduction is expressed in the form of the following statement: *If every φ_i satisfies the corresponding F_i, then φ satisfies F.* To express such a statement in ACL2, therefore, we need to model the semantics of LTL inside ACL2.

As we noted in Section 1.2, it is not possible to model in ACL2 the natural semantics of LTL presented in Section 2 with respect to infinite paths in the Kripke structure. Modeling such a specification requires both infinite path objects and the use of recursion with quantification, neither of which is supported by ACL2. This imposes a road-block on our desire to use the standard specification of LTL. Further, as we mentioned, we could not use previous ACL2 work on model checking to solve the problem. In Section 4.1, we present our approach to circumvent the restrictions, and discuss some implications of our approach in our proof. In Section 4.2, we provide a brief overview of the proof of soundness of the compositional reduction algorithm using our approach.

4.1 ACL2 Specification of LTL

Instead of using paths to define the semantics of LTL, we use *eventually periodic paths*. Given a Kripke structure M, an *eventually periodic path* is an infinite path in the Kripke structure with the characteristic that the path is composed of a finite (possibly empty) "prefix" followed by the infinite repetition of a finite (non-empty) "cycle." Given the semantics of LTL as presented in Section 2, it is a theorem that if there is an path π in M that does not satisfy an LTL formula f, then there is an eventually periodic path σ that does not satisfy the same formula. Since eventually periodic paths are compositions of finite prefix and finite cycles, they can be represented as finite ACL2 objects. We therefore define the function `ltl-ppath-semantics` that specifies the semantics of an LTL formula over a path, assuming that the path is eventually periodic.

The function `ltl-ppath-semantics` is not trivial, but it is not hard to see that it encodes the semantics of LTL given in Section 2. We omit the definition of `ltl-ppath-semantics` in this paper in order to focus on the soundness proof.[10] Given the function `ltl-ppath-semantics`, we can now define the function `ltl-semantics` by quantifying over all eventually periodic paths in the Kripke structure as follows.

```
(defun-sk ltl-semantics (f m)
  (forall ppath
          (implies (compatible-ppath-p ppath m)
                   (ltl-ppath-semantics f ppath m))))
```

Based on this definition of the semantics of LTL in ACL2, we can now prove the soundness of our compositional reduction. We will summarize the key lemmas and the main soundness theorem in Section 4.2. However, we note that our unconventional definition of the semantics of LTL complicates our proof in unexpected ways. To demonstrate the complication, we now highlight a fragment of our proof, that deals with bisimulation. Recall that bisimulation is a key ingredient in the verification of cone of influence reduction. Specifically, we consider the proof of the following key lemma in ACL2.

- If two Kripke structures are bisimilar, then for every eventually periodic path in one there is a corresponding eventually periodic path in the other.

Notice that this statement is a restatement of Lemma 1 for eventually periodic paths. We, therefore, discuss the classical proof of Lemma 1 and then consider the complications arising because of the restatement in terms of eventually periodic paths.

Proof of Lemma 1: Let $B(s, s')$ and let $\pi = s_0 s_1 ...$ be a path starting from $s = s_0$. We construct a corresponding path $\pi' = s'_0 s'_1 ...$ from $s' = s'_0$ by induction. It is clear that $B(s_0, s'_0)$. Assume $B(s_i, s'_i)$ for some i. We will show how to choose s'_{i+1}. Since $B(s_i, s'_i)$ and $R(s_i, s_{i+1})$, there must be a successor t' of s'_i such that $B(s_{i+1}, t')$. We choose s'_{i+1} to be t'.

Given a path π' from s', the construction of a path π is similar. ■

Consider attempting to mimic this proof for the restatement of the lemma in terms of eventually periodic paths. Then for every eventually periodic path in a Kripke structure M, we need to produce a corresponding eventually periodic path in the second Kripke structure M', under conditions of bisimilarity. Assume that

[10] The definition is provided in the supporting materials for demonstration purposes. The function has been renamed to `concrete-ltl-semantics` in the file `concrete-ltl.lisp`. However, we note that for the purpose of this work, we needed only a number of simple properties of this function. For example, the only non-trivial property we had to prove was to demonstrate that if two eventually periodic paths have the same labels in corresponding states, then the function returns the same value for every LTL formula. This statement is similar to the Lemma 2 in Section 3.2. Our proofs were developed by first defining an encapsulated function `ltl-ppath-semantics` that is assumed to have these properties. Finally, these properties were proved for the concrete function. In the supporting materials, we simply describe the encapsulated properties, and show how the proofs of reductions can be derived from those properties. We will have more to say about this encapsulated function in Section 4.2.

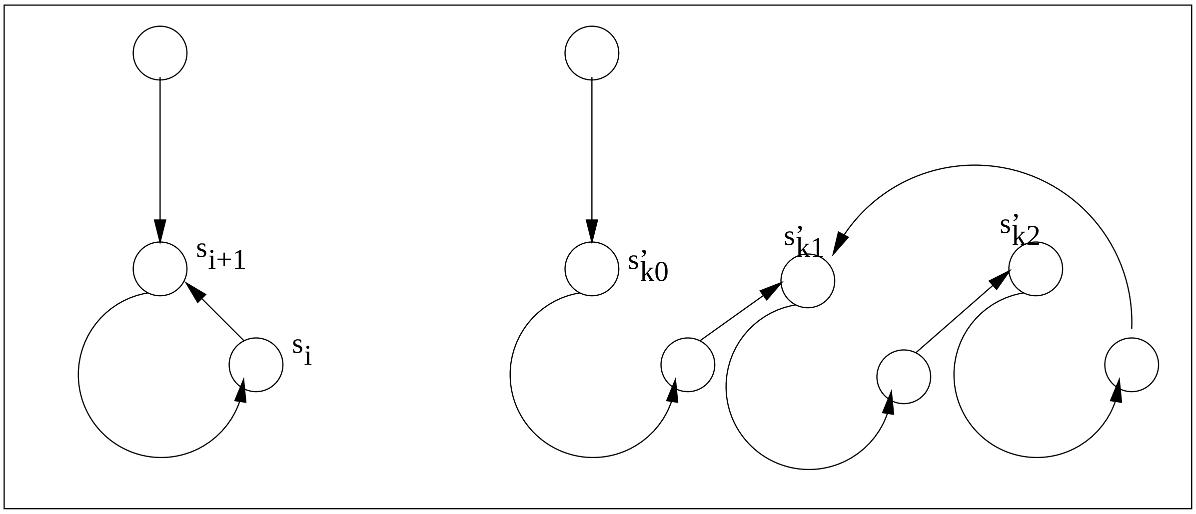

Figure 1: Example showing a periodic path and its match. Note that we need to use the property of bisimilarity to "unwind" the cycle a number of times in the original path in order to get the match. We are guaranteed a match from Pigeon hole arguments.

we have done this for states up to s_i, and that the corresponding state for s_j is s'_j for every $j \leq i$. Now, we need to invoke the condition of bisimilarity to determine the corresponding state for state s_{i+1}. Since we know $R(s_i, s_{i+1})$, we therefore know from the condition of bisimilarity that there exists a state s'_{k1} such that s_{i+1} is bisimilar to s'_{k1} and $R(s'_i, s'_{k1})$. We will be therefore tempted to "match" s'_{k1} to s_{i+1}. However, as we illustrate in Figure 1, we face a complication since the edge from s_i to s_{i+1} might be an edge back to the beginning of a cycle containing s_i. In that case, s_{i+1} might have already been matched to some state s'_{k0}, and $s'_{k0} \neq s'_{k1}$. We, of course, know that both s'_{k0} and s'_{k1} are bisimilar to s_{i+1}. However, to construct an eventually periodic path, we are required to produce a prefix and a (non-empty) cycle. We continue matching corresponding states until we reach a state s'_{k2} in M' that once again matches the state s_{i+1} at the start of the cycle in M. We continue matching states until we reach some state s'_{kl} such that there exists a state s'_{km}, with $m < l$ and $s'_{km} = s'_{kl}$. We are guaranteed that such a situation occurs because the number of states is finite in M'. The proof therefore, corresponds to a pigeon-hole argument.

The pigeon-hole argument is non-trivial but possible to formalize in ACL2. However, we point out that the argument is not inherent in the notions of bisimulation and corresponding paths. It is simply an outcome of our formalization of the semantics of LTL in terms of eventually periodic paths, which forces us to restate the standard facts about the executions of Kripke structures in terms of eventually periodic paths. While the corresponding facts about infinite paths are simple to prove, our restatement often makes the proofs complicated in unexpected ways. In Section 6, we therefore, argue for an extension of ACL2 which will allow us to state the semantics of LTL in terms of infinite paths, and hence simplify verification of such reductions and model checking algorithms considerably in future.

4.2 Verification of Compositional Reduction

In this section, we briefly summarize the key theorems stating the soundness of our compositional reduction algorithm in terms of `ltl-semantics` defined in Section 4.1. The main result we prove is the theorem `compositional-reduction-is-sound` below. The hypothesis `(subset (create-restricted-var-set f) (variables C))` is required since the predicate `ltl-formulap` does not check if the variables in the formula are also variables in the system.

```
(defun ltl-semantics-for-circuit (C f)
  (ltl-semantics f (create-kripke C)))

(defun ltl-semantics-for-circuits* (list)
  (if (endp list) T
    (and (ltl-semantics-for-circuit (second (first list))
                                    (first (first list)))
         (ltl-semantics-for-circuits* (rest list)))))

(DEFTHM compositional-reduction-is-sound
  (implies (and (circuitp C)
                (ltl-formulap f)
                (subset (create-restricted-var-set f) (variables C)))
           (equal (ltl-semantics-for-circuits* (compositional-reduction C f))
                  (ltl-semantics-for-circuit C f))))
```

The theorem goes through virtually automatically in ACL2 once we establish that each of the individual reductions, namely conjunctive and cone of influence reductions are sound.

The soundness of conjunctive reduction is given by the following theorem.

```
(DEFTHM ltl-semantics-is-decomposed-over-conjunction
  (implies (and (ltl-formulap f)
                (equal (len f) 3)
                (equal (second f) '&))
           (equal (ltl-semantics f m)
                  (and (ltl-semantics (first f) m)
                       (ltl-semantics (third f) m)))))
```

The proof of this theorem is a straightforward ACL2 exercise. ACL2 does not prove the theorem automatically, mainly because the definition of `ltl-semantics` uses quantification. However, it is almost trivial to lead ACL2 (with hints) to the proof.

The soundness of cone of influence reduction is given by the following theorem.

```
(DEFTHM cone-of-influence-reduction-is-sound-generalized
  (implies (and (subset interesting-vars (cone-variables vars C))
                (circuitp C)
                (restricted-formulap f interesting-vars))
           (equal (ltl-semantics f (create-kripke
                                    (cone-of-influence-reduction C vars)))
                  (ltl-semantics f (create-kripke C)))))
```

The proof of this theorem roughly follows the lines of the traditional proof. The proof follows by showing the key properties of bisimulation,[11] and then exhibiting a bisimulation relation `circuit-bisim` and the

[11] The key lemmas regarding bisimulation correspond to Lemmas 1 and 2, stated in terms of eventually periodic paths. We discussed the proof of Lemma 1 in Section 4.1. We have proved a restatement of Lemma 2 for the function `concrete-ltl-semantics`, defined in the supporting material in file `ltl.lisp`. The proof is conceptually simple, and at a high level, no different from the classical proof of Lemma 2. In particular, the proof requires induction using the same set of measures that justify the admission of `concrete-ltl-semantics`. However, the ACL2 proofs appear complicated chiefly because of ACL2's lack of support for mutually recursive functions. Since it is our endeavor to discuss proofs of reductions, without distracting the scripts by

corresponding equivalence relation `c-bisim-equiv` on Kripke structures created out of finite state system descriptions.

5 Integration with external Oracles

With compositional reductions verified using ACL2, we can use the reduction algorithms to reduce the original verification problem into one or more "simpler" problems. These individual problems now need to be solved by a model checking procedure. However, note that our function `ltl-semantics` is defined using encapsulation,[12] and hence such a function cannot be evaluated efficiently and run on real systems. To alleviate such problems, we can possibly implement an efficient executable model checking routine for LTL inside ACL2. Manolios [21] did a similar work on designing an executable model checker for μ-calculus formulas. In the current work, we refrained from implementing a model checker inside ACL2 for two reasons. First, it is non-trivial to implement an efficient model checker in ACL2 that is competitive with the model checkers such as SMV [23] that are already on the market. Note that the reduction algorithms are applied on design descriptions sparingly to reduce the problem into simpler problems. On the other hand, model checking routines are required to be invoked on each of the problems, and hence, potentially, the model checkers are required to be extremely efficient in order to enable us to obtain performance benefits using compositional reductions. Second, it is our contention that the basic model checking routines from industrial packages are often simple enough or have been used enough to be trusted, with a similar confidence as the trust bestowed on a theorem-proving system like ACL2. Hence the high confidence attained from verification using ACL2 is necessary more for the reductions than implementations of model checking. For the current work, we decided to find ways of integrating existing state of the art model checkers in ACL2 instead of implementing such functions inside the language of ACL2.

From ACL2's perspective, it is important to have the capability of being able to use external model checking tools for efficient verification of computing systems. For example, Moore [24] provides a "grand challenge" in formal methods, inviting researchers in formal methods to verify a complete stack from transistors to high-level program semantics. Presumably, such a verification effort would need collaboration of different tools built to provide support to the theorem-proving engine. A practical approach to verifying a stack will probably allow for automated analysis tools like model checkers to certify parts of the stack, wherever applicable, and theorem-provers for proving higher-level theorems which are beyond the reach of such automated tools. However, if ACL2 is now used to embark on the challenge, then ACL2 has no way of using currently available automated analysis tools to aid the verification work, at least inside the logic. ACL2 is monolithic in the sense that every tool used in conjunction with the theorem-prover is required to be implemented in the logic of the theorem-prover. This appears to the authors to be in some cases tantamount to "reinventing the wheel," and possibly with loss of efficiency to unacceptable levels. Hence we believe it is important for the ACL2 community to work on integrating current state of the art decision procedures with ACL2.

Short of a certified way to integrate external model checkers with ACL2, we still tried to find appropriate ways of calling an external model checker (Cadence SMV) for model checking the reduced systems produced from our reduction algorithms. Our approach is to define a function `ltl-semantics-hack` with a guard of `T`. This function is specified axiomatically to be the same as the function `ltl-semantics`. Since the guard of the function is `T`, the evaluation of the function proceeds according to the underlying Lisp implementation of the function.

```
(defaxiom ltl-semantics-hack-revealed
  (equal (ltl-semantics-for-circuits C f)
```

complicated lemmas on mutually recursive code, we decided to present the supporting materials using the encapsulated function `ltl-ppath-semantics` that is constrained to satisfy Lemma 2. As we pointed out, we still provide the definition of the function `concrete-ltl-semantics` for demonstration purposes.

[12]The `defun-sk` construct used in the definition of `ltl-semantics` is actually a macro which expands into an encapsulate.

```
        (ltl-semantics-hack C f)))
```

We then proceed to define a function `ltl-semantics-hack*` to correspond to `ltl-semantics-for-circuits*` in ACl2.

```
(defun ltl-semantics-hack* (list)
  (if (endp list) T
    (and (ltl-semantics-hack (second (first list))
                             (first (first list)))
         (ltl-semantics-hack* (rest list)))))

(defthm ltl-semantics-hack*-revealed
  (equal (ltl-semantics-for-circuits* list)
         (ltl-semantics-hack* list))
  :hints (("Goal"
           :induct (ltl-semantics-for-circuits* list))))
```

We then prove a rewrite rule that expands calls to `ltl-semantics-hack*` into a collection of calls to `ltl-semantics-hack` when the input to the function is a quoted constant.

```
(DEFTHM ltl-semantics-hack-revealed-for-rewriting
  (implies (syntaxp (quotep list))
           (equal (ltl-semantics-hack* list)
                  (if (endp list) T
                    (and (ltl-semantics-hack (second (first list))
                                             (first (first list)))
                         (ltl-semantics-hack* (rest list)))))))
```

In the underlying Lisp, we then define a function `ltl-semantics-hack` to override the definition in the ACL2 logic. The idea is to use the ACL2 `sys-call` construct to invoke calls to the underlying operating system. The `sys-call` calls the underlying script `smv-script`, which takes the current representation of the state machine, translates it in the form that is understood by the SMV model checker, and calls the SMV model checker on the representation. The call returns with an answer according as the model checker returns.

This script, which has been implemented in Perl, along with the rewrite rules above, forces ACL2 to rewrite calls to `ltl-semantics-for-circuits*` to successive calls to `ltl-semantics-hack` which are then evaluated using the underlying operating system (assumed to be Unix) via the SMV model checker. In this way, we benefit from the efficiency of a state-of-the-art model checker.

Our initial results indicate that the performance of the compositional model checking using this technique is nowhere near the performance results obtained by the use of reductions built-in with the SMV model checker. On the other hand, the performance is still significantly better than applying SMV's model checking algorithm alone on the original system. However, we ran our current tests only on small designs where performance is of the order of seconds or minutes. We plan to run tests on larger examples in order to come up with reliable comparative figures. For that purpose, we need to design translators that can transform benchmarks systems (written in VHDL or Verilog) to the form understood by our home-grown ACL2 predicate `circuitp`. We do not anticipate this to be a difficult implementation task, since we have already implemented translators from ACL2 descriptions to forms understood by SMV. However, implementation of such a translator has not been done yet.

The use of Cadence SMV in conjunction with ACL2 in this work might give the impression that we have shown a way of integrating external tools with the ACL2 theorem-prover. However, we emphasize that our "integration" is an intermediate workaround, and is inadequate in many respects. In particular, since `ltl-semantics-hack` is a function in ACL2, the user can easily prove `NIL` in ACL2 by opening up the body of the function, for example using a `:use` hint to the theorem-prover. Hence, our current approach clearly makes ACL2 unsound. Further, our approach replacing the definition of the function by getting into the underlying Lisp is repugnant at best. On the other hand, we believe it would be much better if ACL2 itself allows some way of hooking in an external oracle to verify theorems of certain forms. The Isabelle theorem prover provides the concept of external checkers and allows the possibility of attaching external oracles to verify certain theorems [27, § 6]. For example, Isaballe adds a tag to every proposition certified by an external oracle stating the name of the oracle, and the proposition it certified. Any subsequent theorem that uses "external" proposition in its proof inherits the tags of the proposition. This process is transitive, so that a user can inspect a top-level theorem to see exactly what oracles were used in its proof, and what axioms they certified. This allows the user to determine what level of trust to place in the top-level theorems, based on their trust of the oracles and of Isabelle. Isabelle's external oracle mechanism has been used to integrate

- an efficient μ-calculus model checker, as part of a logic for I/O-Automata [25].
- the Stanford Validity Checker (SVC), as an arithmetic decision procedure for the real-time interval logic Duration Calculus (Isabelle/DC) [12].
- the MONA model checker, as an oracle for deciding formulas expressed in the weak second-order monadic logic of one successor (WS1S) [1].

External oracles are also used in the HOL family of higher order logic theorem provers. Gunter [10] first published the concept of "tagging" a theorem with the external oracles called during its proof, as an extension to the HOL90 theorem prover. The PROSPER project [5] carried the idea of external oracles further, using the HOL98 theorem prover as a uniform and logically-based coordination mechanism between external verification tools. The most recent incarnation of this family of theorem provers, HOL 4, continues to use an external oracle interface to decide large boolean formulas through connections to state-of-the-art BDD and SAT-solving libraries [9, 15].

We believe that adding support for external oracles is possible in ACL2 and will facilitate the verification of large-scale systems. Admittedly, as an anonymous referee pointed out, there is a possibility of the integrated system being unsound if the user makes a mistake in this attachment. However, checking of propositions by external oracles can be simply viewed as a process of adding some axioms in ACL2 using a `defaxiom` event, with the user having greater trust at the axioms added by checking them with some external tools. It is only the integrated environment, consisting of ACL2 and the external oracles along with the translation programs between them that guarantees soundness of a theorem proved by a combination of ACL2 with external oracles. If any of the processes is unsound then the composite system is unsound and hence ACL2 does not take any responsibility of theorems proved by the integrated system. Another way of interpreting such a theorem is as an implication. If every proposition certified by the external oracles in a theorem is true, then the theorem is true in the ACL2 logic.

6 Augmenting the ACL2 logic

The proof we described in Section 4.2 provides assurance that our compositional reductions are sound. We anticipate verifying more sophisticated decomposition algorithms in ACL2 in the future. In order to facilitate this effort, this section provides a commentary on some of the problems we faced and suggestions for alleviating them by strengthening the logic.

The primary challenge lies in ACL2's inability to specify natural properties of infinite sequences. As we mentioned in Section 4, resolving this challenge requires some way of axiomatizing infinite path objects, as well as the ability to define recursive functions containing occurrences of quantifiers. ACL2 rules this out by only allowing already-defined predicates to be quantified over. Similarly ACL2 does not allow mutual recursion where one of the functions in the clique is defined using quantification.

We have with difficulty circumvented this limitation using eventually periodic paths. However, even this avenue will likely be closed to us once we try to incorporate known algorithms for model checking parameterized infinite state systems.

6.1 Conservativity

The chief argument to maintaining the separation between well-founded recursion and quantification is that combining them would violate the *conservativity* of the ACL2 logic. A definition is not conservative if its associated axiom(s) can be used to create a previously unprovable theorem that does not mention the new definition's symbol. ACL2's encapsulation mechanisms depend on conservativity, and a proof that the ACL2 logic is conservative is sketched out in [18].

To see why combining recursion with quantification violates the conservativity of ACL2, consider the following definition of `true-formula`, a truth-predicate for Peano arithmetic.[13]

```
(mutual-recursion
(defun true-formula (formula assignment D)
  (declare (xargs :guard (and (quantified-formula-p formula)
                              (symbol-alistp assignment)
                              (defun-listp D))))
  (cond
   ((exists-p formula)
    (true-exists-formula (qvar formula) (qbody formula) assignment D))
   ((forall-p formula)
    (true-forall-formula (qvar formula) (qbody formula) assignment D))
   ((and-p formula)
    (and (true-formula (conjunct-1 formula) assignment D)
         (true-formula (conjunct-2 formula) assignment D)))
   ((or-p formula)
    (or (true-formula (disjunct-1 formula) assignment D)
        (true-formula (disjunct-2 formula) assignment D)))
   ((not-p formula)
    (not (true-formula (not-body formula) assignment D)))
   (t
    (term-value formula assignment D))))

(defun-sk true-exists-formula (var formula assignment D)
  (exists var-value
          (true-formula formula
                        (cons (cons var var-value) assignment)
                        D)))

(defun-sk true-forall-formula (var formula assignment D)
  (forall var-value
          (true-formula formula
                        (cons (cons var var-value) assignment)
                        D))))
```

[13]This example was provided to us by Matt Kaufmann.

However, it is well-known that Peano arithmetic augmented with such a truth-predicate allows us to prove the consistency of Peano arithmetic itself, by proving by induction over formulas that `true-formula` holds of everything that is provable. By Gödel's Incompleteness Theorem, any augmentation of a theory of arithmetic that can prove the consistency of the original theory must not be conservative. This non-conservativity carries over to the logic of ACL2, which is essentially a formalization of first order logic with arithmetic.

6.2 Consistency of an extended ACL2

Hence, we believe that augmenting the logic with the ability to apply recursion and quantification simultaneously will produce a strictly stronger (and therefore non-conservative) extension of ACL2. However, we think such an extension is still consistent with respect to several known logics, for example ZFC [6, 19, 11], and higher order logic.

Our informal consistency argument for the case of ZFC is as follows. We presume that there is a family of models M for ACL2 in ZFC, such that for every model $m \in M$, the class of ACL2 objects is an element k in the Von Neumann cumulative hierarchy, and the class of ACL2 functions is contained in the set $k \to k$. In that case, every axiom and inference rule of ACL2 could be proved as a theorem of model m in ZFC. However, we can directly model well-founded mutually recursive nests containing `defun-sk` in m using the axiom of choice. Since quantification is defined in ACL2 directly in terms of `defun-sk`, the consistency of such an augmented ACL2 will follow from the consistency of ZFC.

We note here that a logic that formalizes both naturals and reals is non-conservative in the above sense. In order to see this, notice that we can construct a model of Peano Arithmetic in such a logic, where the natural numbers and arithmetic operators of Peano Arithmetic are encoded as natural numbers and arithmetic operators in the logic, and formulas of Peano Arithmetic are encoded using real numbers. In order to apply such encoding, we need to construct a mapping `toReal` from $P = \texttt{Nat} \to \texttt{Bool}$ to the range $R_1 = [-1, 1]$ of reals, where, for every natural number i and predicate p in P, $p(i)$ is mapped to $2^{-(i+1)}$. That implies that the binary decimal expansion of r in R_1 is the truth table for p.[14] Using Godel encodings, we can construct injective mappings from predicates over tuples of numbers to R_1 as well. Then we can model the semantic meaning of a Peano arithmetic formula as the pair:

$$'((v_1 ... v_n)\ .\ (\texttt{ToReal}\ (\texttt{Godel-encoding}\ (\texttt{predicate-of-formula}\ p))))$$

The first component of the pair gives the free variables in p and the second component encodes the truth value of p for each (encoded) assignment to the free variables. It should be also possible to define each Peano Arithmetic formula operator, including quantification, as a function in such a logic, that respects the above mapping. Hence, we can define a truth predicate for Peano Arithmetic in the logic, similar to `true-formula` above.

We provide the above arguments to show that natural extensions of ACL2 to real numbers might lead to similar violations of conservativity, as mentioned in [18]. ACL2(R) [7] is an extension of ACL2 that supports real numbers. We have been told that ACL2(R) is still conservative with respect to ACL2 by imposing restrictions on the operations that can be performed on reals. In particular, while real numbers can be stored as infinite precision numbers, it is not possible to define an ACL2(R) function that recurs down this infinite precision.

In any event, it is our position that loss of conservativity with regard to the base logic of ACL2 is justifiable when analysis of practical problems demand such extension, and when the consistency of the stronger logic is still assured.

[14] A complication in this scheme arises since some pairs of binary decimal expansions map to the same real number. Every such pair is either of the form $(0.x_1...x_n1000..., 0.x_1...x_n0111...)$ or is the pair $(1.000..., 0.111...)$. To disambiguate these pairs, the mapping `ToReal` inverts the sign of the final real number when constructing a decimal expansion that matches the right hand side of such a pair.

7 Conclusions and future directions

We have presented an approach toward formalizing abstract semantics of LTL inside ACL2, with a view to proving that compositional reductions are sound. We note that rigorous proofs of correctness of our reductions are already present in the literature on model checking [4]. However, as we pointed out in Section 1, our approach is to verify implementations rather than the abstract algorithms. Admittedly, it is possible to implement the reductions in a more efficient manner in ACL2. However, our experience indicates that once the key insights are obtained by verifying that an inefficient but simple implementation has the requisite property, it is relatively simple to verify that an efficient implementation always returns the same result as the inefficient one.

We plan to verify more powerful reduction techniques in ACL2, and hence, compositional algorithms that can reduce the problem more efficiently. For such algorithms to be effectively used in practical problems, then we need ACL2 to be less monolithic and allow for easy and seamless integration with external trusted decision procedures like model checkers. We plan to work on integrating external decision procedures with ACL2. As we mentioned in Section 5, our current approach is not fully satisfactory and an intermediate hack at best. Further, we believe that for large-scale verification of non-trivial decision procedures it is important to be able to state the specification of such procedures in the most natural way, so that we can approach the verification using classical techniques in the literature. We note that the correctness of most of the decision procedures in hardware verification [4] are based on the notion of infinite sequences and recursion. Hence we believe that if ACL2 is to be used in conjunction with decision procedures, as would almost certainly be required in a large verification project like [24], it is important to have an extension to the logic so that we can define functions reflecting such specifications with relative ease. We contend that for the sake of simplicity of specifications, it is important to consider such an extension as long as it is consistent, even at the cost of conservativity. On the other hand, we are also considering possible extensions of ACL2(R) instead of ACL2, in implementing such enhancements. ACL2(R) might be a better tool for such enhancements since we believe that it might be possible to provide quantification and recursion in ACL2(R) by implementing a recursive interpreter function similar to [18], without unreasonably extending the logic of theorem-prover.

Acknowledgments

We thank Ernie Cohen for suggesting the idea of using eventually periodic paths to represent the semantics of LTL. We are also thankful to our colleagues in the Scalable Computing Group at HP Labs Cambridge, and the ACL2 group at the University of Texas at Austin, for numerous comments and suggestions. We specifically thank Matt Kaufmann, Rob Sumners, Robert Krug, and Michael Bogolomny at UT, who helped us with ACL2 proofs and implementation issues on more than one occasion during this endeavor. In particular, the ideas in Section 5 are direct contributions of Robert Krug. We also thank J Moore for reading and providing useful comments on a preliminary draft of this paper, and the anonymous referees for providing several constructive suggestions.

References

[1] D. Basin and S. Friedrich. Combining WS1S and HOL. In Dov M. Gabbay and Maarten de Rijke, editors, *Frontiers of Combining Systems 2*, pages 39–56. Research Studies Press/Wiley, Baldock, Herts, UK, February 2000.

[2] K. Bhargavan, C. A. Gunter, and D. Obradovic. Routing Information Protocol in HOL/SPIN. In *Proceedings of the 13th International Conference on Theorem Proving in Higher Order Logics (TPHOLs '00)*, volume 1869 of *Lecture Notes in Computer Science*, pages 53–72. Springer-Verlag, 2000.

[3] Ching-Tsun Chou and D. Peled. Formal Verification of a Partial-Order Reduction Technique for Model Checking. *Journal of Automated Reasoning*, 23:265–298, 1999.

[4] E. M. Clarke, O. Grumberg, and D. A. Peled. *Model-Checking.* The MIT Press, Cambridge, MA, January 2000.

[5] L. A. Dennis, G. Collins, M. Norrish, R. Boulton, K. Slind, G. Robinson, M. Gordon, and T. Melham. The PROSPER toolkit. In S. Graf and M. Schwartbach, editors, *Tools and Algorithms for Constructing Systems, TACAS, Berlin, Germany*, number 1785 in Lecture Notes in Computer Science, pages 78–92. Springer-Verlag, 2000.

[6] K. Devlin. *The Joy of Sets: Fundamentals of Contemporary Set Theory.* Springer-Verlag, 2nd edition, 1992.

[7] R. Gamboa. *Mechanically Verifying Real-valued Algorithms in ACL2.* PhD thesis, University of Texas at Austin, 1999.

[8] M. J. C. Gordon and T. F. Melham, editors. *Introduction to HOL: A Theorem-Proving Environment for Higher-Order Logic.* Cambridge University Press, 1993.

[9] Michael J. C. Gordon. Programming combinations of deduction and BDD-based symbolic calculation. *LMS Journal of Computation and Mathematics*, 5:56–76, 2002.

[10] E. L. Gunter. Adding external decision procedures to HOL90 securely. *Lecture Notes in Computer Science*, 1479:143–152, 1998.

[11] P. R. Halmos. *Naive Set Theory.* Van Nostrand, 1960.

[12] S. T. Heilmann. *Proof Support for Duration Calculus.* PhD thesis, Technical University of Denmark, Dept. of Information Technology, 1999.

[13] G. J. Holzmann. The Model Checker Spin. *IEEE Trans. on Software Engineering*, 23(5):279–295, May 1997.

[14] W. A. Hunt (Jr). The De Language. In P. Manlolios, M. Kaufmann, and J Moore, editors, *Computer-Aided Reasoning: ACL2 Case Studies*, pages 119–131. Kluwer Academic Publishers, June 2000.

[15] HOL 4, Kananaskis 1 release. http://hol.sf.net/.

[16] M. Kaufmann, P. Manolios, and J Moore, editors. *Computer-Aided Reasoning: ACL2 Case Studies.* Kluwer Academic Publishers, June 2000.

[17] M. Kaufmann, P. Manolios, and J Moore. *Computer-Aided Reasoning: An Approach.* Kluwer Academic Publishers, June 2000.

[18] M. Kaufmann and J Moore. Structured Theory Development for a Mechanized Logic. *J. of Automated Reasoning*, 26(2):161–203, 2001.

[19] K. Kunen. *Set Theory — An Introduction to Independence Proofs*, volume 102 of *Studies in Logic and Foundations of Mathematics.* North-Holland, Amsterdam, 1980.

[20] P. Manolios. *Mechanical Verification of Reactive Systems.* PhD thesis, University of Texas at Austin, 2000.

[21] P. Manolios. Mu-Calculus Model Checking in ACL2. In P. Manlolios, M. Kaufmann, and J Moore, editors, *Computer-Aided Reasoning: ACL2 Case Studies*, pages 73–88. Kluwer Academic Publishers, June 2000.

[22] P. Manolios, K. Namjoshi, and R. Sumners. Linking Model-checking and Theorem-proving with Well-founded Bisimulations. In N. Halbwacha and D. Peled, editors, *Computer-Aided Verification (CAV)*, volume 1633 of *LNCS*, pages 369–379, 1999.

[23] K. McMillan. *Symbolic Model Checking.* Kluwer Academic Publishers, 1993.

[24] J Moore. A Grand Challenge Proposal for Formal Methods: A Verified Stack. Presented at *10th Anniversary Colloquium of the UN University International Institute for Software Technology: Formal Methods at the Crossroads*, March 2002.

[25] O. Müller and T. Nipkow. Combining model checking and deduction of I/O-automata. In E. Brinksma, editor, *Proceedings of the First International Workshop on Tools and Algorithms for the Construction and Analysis of Systems*, Aarhus, Denmark, May 1995. Springer-Verlag LNCS 1019.

[26] S. Owre, J. M. Rushby, and N. Shankar. PVS: A Prototype Verification System. In D. Kapoor, editor, *11th International Conference on Automated Deduction (CADE)*, volume 607 of *Lecture Notes in Artificial Intelligence*, pages 748–752. Springer-Verlag, June 1992.

[27] L. Paulson. *The Isabelle Reference Manual.* http://www.cl.cam.ac.uk/Research/HVG/Isabelle/dist/Isabelle2003/doc/ref.pdf.

[28] H. Saiedi and N. Shankar. Abstract and Model Check while You Prove. *Lecture Notes in Computer Science*, 1633:443–454, 1999.

[29] K. Schneider and D. W. Hoffmann. A HOL Conversion for Translating Linear Time Temporal Logic to omega-Automata. In *Theorem Proving in Higher Order Logics*, pages 255–272, 1999.

[30] R. Sumners. Bisimulation in ACL2. Private Communication.

[31] R. Sumners. An Incremental Stuttering Refinement Proof of a Concurrent Program in ACL2. In *Second International Workshop on ACL2 Theorem Prover and Its Applications*, Austin, TX, October 2000.

[32] J. von Wright. Mechanizing the Temporal Logic of Actions in HOL. In Myla Archer, Jennifer J. Joyce, Karl N. Levitt, and Phillip J. Windley, editors, *Proceedigns of the International Workshop on the HOL Theorem Proving System and its Applications*, pages 155–161, Los Alamitos, CA, USA, August 1992. IEEE Computer Society Press.