Using Theorem Proving with Algorithmic Techniques for Large-Scale System Verification*

Sandip Ray
Department of Computer Sciences
The University of Texas at Austin
sandip@cs.utexas.edu
http://www.cs.utexas.edu/users/sandip

Abstract

We propose to write a thesis on using theorem proving with algorithmic techniques for verification of large scale computer systems. Large-scale computer systems tend to have a non-terminating computation, and reasoning about such systems involves exhibiting some temporal property of the system. For large system models, automatic verification of non-trivial temporal properties is often infeasible, because of the *state explosion* problem. Theorem proving has a better potential to scale up to large designs, but it often requires considerable user assistance to guide the proof. We consider an intermediate approach, using theorem proving to verify correspondence between executions of a concrete system design and its abstraction, and then using automated analysis to verify the temporal properties of abstractions. We explore the logical issues and problems involved in the use of automated analysis on abstractions verified by a theorem prover. We propose to implement tools and techniques that provide a reasonable amount of automation in the verification process. Further, we plan to demonstrate our techniques on verification of multiprocessor system models, and specifically focus on synchronization protocols, cache coherence, and pipelined architecture of realistic systems. We use ACL2 (a programming language, logic, and theorem prover) to illustrate our techniques.

1 Introduction

Design of large-scale multiprocessor systems is an error-prone exercise. Multiprocessor systems normally involve a number of processes communicating and synchronizing, often using complicated protocols. Such systems typically have non-terminating or infinite computations, and reasoning about them involves exhibiting some non-trivial temporal property of the system. In this context, it is often difficult for a human designer to keep track of or reason about the various possible states a system can reach in the course of such complex interactions. Thus an "innocent" optimization in design made with an inaccurate mental picture of the behavior of the system might easily introduce a bug which can be hard to detect or diagnose. A viable approach to guarantee sufficient assurance in correct executions of large-scale multiprocessor systems is the use of formal verification. Formal verification involves modeling the executions of the systems under interest in some formal reasoning framework and using a trusted computer program to prove the desired properties of the model. Thus, if a verification effort is successful, it provides a mathematical guarantee for the correctness of the system under inspection, up to the accuracy of the model and the soundness of the computer program employed in the reasoning process.

Research in formal verification can be broadly divided into two categories, namely deductive reasoning or theorem proving, and algorithmic verification techniques like model checking [14, 11], symbolic simulation [25], and symbolic trajectory evaluation [10]. The deductive approach is to model the executions of

*This is a dissertation proposal for the Department of Computer Sciences at The University of Texas at Austin. Support for this work has been provided in part by the SRC under contracts 99-TJ-685 and 02-TJ-1032.

the system as a formal theory in some general-purpose, expressive, but typically undecidable logic, and then prove the desired properties of the system as theorems for the formal theory. Modern theorem provers like ACL2 [27, 28], HOL [19], and PVS [45], provide a certain amount of automated support in the proof generation process. On the other hand, the algorithmic approach is to restrict the system model and the correctness properties to those expressible in a decidable logic, usually some form of propositional temporal logic over finite automata. Such logics are often sufficient to describe properties of interest of commercial systems. In this case, verification can be completely automated, at least in principle. The algorithms involve either explicit or symbolic traversal of all possible states the system can reach during any execution. If a property violation is found, then a counterexample trace can be returned.

We propose to explore a combination of the deductive and algorithmic approaches for verification of large-scale industrial system models. Our principal focus is on verifying multiprocessor systems implementing synchronization protocols, cache coherence, and pipelined architectures. For reasoning about practical systems of such complexity, both algorithmic and deductive reasoning frameworks are inadequate as is. Algorithmic verification procedures cannot be applied directly to such systems because of the *state explosion* problem. Practical multiprocessor systems have too many states for even a symbolic state exploration of the complete system model to be feasible as is. Hence, in practice, concrete system models are "simplified" or *abstracted* to a system model with fewer states. However, such abstractions are often manual or adhoc, leading to abstract models very different from the original models of interest. Thus no formal guarantee can be provided in general regarding the original system model based on automatic verification of the abstract system. Theorem proving, on the other hand, is relatively insensitive the state explosion problem, and hence, has more potential to scale up to large system models. However, since the logic used in theorem proving is normally undecidable, successful use of theorem proving for large-scale system verification often requires considerable involvement of a human user to guide the theorem prover through the non-trivial reasoning steps. Such a user needs to be experienced both with the formal logic and the reasoning engine of the theorem prover, and the details of the system model being verified. Further, the structure of the proof itself is highly dependent on the underlying design. Seemingly minor design changes often require a great deal of proof reworking. Given the rate of growth of modern industrial systems, maintenance of proofs in the face of design evolutions is problematic at best.

Our reasoning framework exploits the generality and robustness of theorem proving with the automation provided by the algorithmic verification procedures. Roughly, the idea is to use theorem proving to verify some correlation between the executions of the concrete system and those of an abstract model. If the abstract model is simple enough, then the correctness of the concrete system then follows from simple inspection of the abstract model. In many cases, the abstract model can be expressed in a decidable logic with a relatively small number of states, and automatic analysis can be effectively performed on the model. In that situation, the corresponding properties of the concrete system then follows from the theorems verifying the correlation between the two systems. Hence, our methodology provides a nice combination of deductive and automatic verification techniques in verifying real-world system models. On the other hand, to be effective and practical, the methodology must additionally satisfy the following two requirements.

1. Verification of correspondence between the concrete and abstract system models using theorem proving should be achieved with a reasonable amount of automation.

2. The integration of automatic procedures with the theorem proving environment should be both sound and efficient.

We address requirements 1 and 2 in the remaining sections. In Section 2, we discuss a specific methodology for showing correspondence between two models specified at different level of abstractions. Our methodology is based on "well-founded refinements" introduced by Sumners [51], and is a consequence of work by Namjoshi [44], and Manolios, Namjoshi, and Sumners [37] in the context of "well-founded bisimulations". We argue that our framework has a number of methodological advantages in the context of proofs about multiprocessor system models. To address requirement 1, we then carefully analyse typical proofs of correspondence based on this notion of correlation, in order to identify the manual portions of typical multiprocessor proofs. We conclude that most of the human effort goes in the definition and construction of *invariants*. In Section 3, we then build tools and techniques that assist the human user in such construction. Our approach uses an efficient term-rewriting strategy to construct a *finite model*, which can be used by automatic algorithms to check purported invariants.

We address requirement 2 in Sections 4, 5, 6, and 7. Roughly, the problem of soundness arises from the possible incompatibility of the logics implemented by the decision procedures with the logic of the theorem prover. Since the theorem prover and the automatic procedures implement different logics, care needs to be taken to use the decision procedures on abstractions verified by the theorem prover and justify soundness of the composite verification. We examine the issue in detail in Section 4, and discuss our solution to the issue. Our solution is to define the decision procedures as formal theories *inside* the theorem prover, and use the theorem prover to prove lemmas that characterize the actions of the procedures. In Section 5 and 6, we demonstrate the viability of this approach by verifying two specific decision procedures, namely a compositional model checker and a procedure for Generalized Symbolic Trajectory Evaluation. In Section 7, we discuss the efficiency issues with this integration and propose approaches to allow for integration of efficient, trusted decision procedures available in the market with our framework, in order to obtain the high assurance obtained from verification by theorem proving and the efficiency of high-quality industrial tools.

Our techniques require that we work with a theorem prover in which we can efficiently model large industrial systems and automatic decision procedures, and prove properties about such artifacts with relative ease. We use ACL2 [27, 28] for that purpose. ACL2 is an industrial-strength version of the Boyer-Moore theorem prover (Nqthm) [7]. ACL2 has been successfully used to verify industrial systems [30, 9]. One advantage of ACL2 for our work is that the semantics of ACL2 is like a programming language, namely Applicative Common Lisp. Hence it is relatively simple to code up system models and decision procedures in ACL2. Further, the logic of ACL2 is executable, and in fact, the executable portion of ACL2 is simply a subset of Common Lisp. Hence verification of decision procedures in ACL2 is equivalent to verifying executable code. However, a problem with the use of ACL2 comes from the fact that the logic is essentially first-order. Hence, a specification or proof technique requiring higher-order theory is unavailable to us in ACL2. This often leads to some of the characterization lemmas of decision procedures being awkward, and some of the proofs more complicated. However, notice that characterization lemmas of decision procedures are required to be proved once and for all, and hence, executability and simplicity of logic are sufficient to justify complication of such "once-off" proofs.

We have tested the viability of our methodology in the context of ACL2 theorem proving. In particular, we have modeled multiprocessor systems in ACL2, and used our techniques to verify properties of such models. In Sections 2 and 3 we discuss simplified models of synchronization protocols and multiprocessor memory systems that we have verified in ACL2 using our correspondence framework. We are currently modeling more realistic system models in ACL2, and reasoning about them using our correspondence framework. We have also used the decision procedures, in particular the compositional model checker described in Section 5 to verify properties of simple abstract systems. Our initial experiments indicate that our approach is effective in practice. However, we should note that the models we have analyzed so far are still very simplistic compared to commercial designs. We plan to work on realistic and practical systems that reflect the complexity of commercial designs. We plan to demonstrate the efficacy of our framework by verifying modules of detailed realistic systems using our composite framework.

We remark that a lot of recent researches has focused on the combination of theorem proving with algorithmic techniques for practical verification of large-scale multiprocessor system models. Related approaches include, among others, the STeP system [6], and the Intel verification system Forte [1]. Both STeP and Forte are special purpose theorem provers for reasoning about system models, and integrate a number of algorithmic techniques like STE and invariant proving strategies in their inference rules. Algorithmic techniques have also been successfully used with general purpose theorem provers like HOL and PVS for verification of large-scale hardware. The standard approach is to apply automatic algorithms to verify properties of the models, modulo certain proof obligations. The proof obligations are then dispatched by a theorem prover. Examples of successful verification using this approach for pipelined processor models can be found in [23, 24].

The chief difference between our work and related research is in our emphasis on requirement 2 in the context of combination of deductive and algorithmic strategies. In a special purpose decision procedure like STeP or Forte, the algorithmic procedures are themselves used as inference rules in the logic. For example, STeP provides an inference rule `G-INV` which is a "general invariance rule" applicable to system models, and Forte has GSTE as an inference rule. As a result, the logics of the reasoning systems are more complicated. In contrast, ACL2 is a very general-purpose theorem prover, whose logic is essentially a first order logic of recursive functions [31]. In fact, theorems about pure mathematics are often proved with ACL2. The simplicity of the logic of ACL2 allows for a clearer interpretation of the theorems proved and models verified with the theorem prover. Further, as we discuss in Section 4, there is possibility of inconsistency between the deductive and algorithmic reasoning techniques even if each is consistent on its own. Hence, if a theorem prover is used to dispatch proof obligations generated by algorithmic procedures, one must argue that the combination of the two logics is sound. This aspect of the combination is often argued informally, or not argued at all, in related approaches to combination of deductive and algorithmic procedures. On the other hand, in our approach, we can provide a characterization of the algorithmic procedures by defining the procedures inside the logic of the theorem prover and verifying characterization theorems for the procedures. Once characterization theorems are proved, we can use freely use them to verify properties of system models, with the knowledge that the decision procedures are consistent with our logic. In case of external tools, we propose to use "defining axioms", as we discuss in Section 7. The axioms provide a summary of the assumptions that we need to make to apply algorithmic procedures. At any rate, our framework makes explicit the assumptions required in using a algorithmic decision procedures with the theorem prover. This clarity, of course, comes at the cost of complexity in specifying and verifying decision procedures in the logic of the theorem prover. As we show in Sections 5 and 6, this can be tedious and complicated, but we demonstrate that verification of decision procedures can be achieved effectively within the ACL2 system.

In Sections 2, 3, 4, 5, 6, and 7, we provide describe our completed work. We have omitted some detail in this paper, including proofs of some of the theorems. The systems and decision procedures we describe have been modeled using ACL2, and their verification carried out with the ACL2 theorem prover. The relevant ACL2 scripts for all the completed work are available upon request from the author. In each section, we summarize related research wherever appropriate. We should note that our summary is by no means complete and more elaborate discussions of related work, and comparisons with our approach will appear in the thesis. In Section 8, we describe what remains to be done. In Section 9, we summarize our work and provide some concluding remarks.

2 Well-founded Refinements with Fairness Constraints

In this section, we discuss the framework we use to verify correlations between executions of a concrete design and those of its abstraction. Our notion of correspondence is based on what we call "fair well-founded refinements". The idea of refinements follows the presentation of Sumners [51] and is a direct consequence of the work by Namjoshi [44], and Manolios, Namjoshi, and Sumners [37] in the context of *well-founded equivalence bisimulations*. In Section 2.1, we discuss our approach to modeling a computer systems as a formal theory in the ACL2 logic. In Section 2.2, we review the notion of stuttering refinements in the context of verification of such models. In Section 2.3, we discuss an approach to integrate fairness constraints in the well-founded refinement framework and discuss the methodological advantages of such integration. Section 2.4 uses our methodology for a case study, verifying an abstract multiprocessor system model implementing a mutual exclusion protocol.

2.1 System Models

In this section, we focus our attention on modeling computer systems. The specific modeling framework we describe in this section is not new or novel, and it has been used in the verification community, in particular ACL2, for a while. Nevertheless, we describe the framework here for the purpose of clarity, and to make the paper self-contained.

A system model is defined by a (possibly infinite) set S of *states*, a (possibly infinite) set I of *inputs*, a subset $init \subseteq S$, called the set of *initial states*, and a function $\texttt{next} : S \times I \rightarrow S$, called the *next state function*. A state is defined by an assignment of values to each system variable. We assume that the collection of variables in the system is finite. Note, however, that we make no restriction on the number of values any variable can take, resulting in a possibly infinite set of states. Finally, an *execution* of the system is defined to be a sequence of states, $s_0 s_1 ....$, such that each $s_i \in S$, and for any pair of consecutive states (s_i, s_{i+1}), there exists an input $i \in I$ such that $s_{i+1} = \texttt{next}(s, i)$. If the sequence is finite, then it is termed a *finite execution*; otherwise it is termed an *infinite execution*.

In ACL2, a system model is defined as a formal theory by a unary predicate `init?` that recognizes the set of initial states, a binary predicate `legal` that declares a specific input to be applicable at a particular state, and an implementation of the function `next`. To avoid deadlocks, we will assume that for every state s, there exists some input i such that $\texttt{legal}(s, i)$. When we talk about a specific system model, we will refer to a specific triple of ACL2 functions $\langle \texttt{init?}, \texttt{legal}, \texttt{next} \rangle$. In Section 2.2, we will show that reasoning about system models often involves another component, called the *labeling function*. When a labeling function is added to the system model, we will refer to it as an *augmented system model* or simply, a *model*. We will also often loosely describe the application of the function `next` on a particular state of a system model and a particular input to be a *step* of the model.

The approach of modeling a system using a (total) next state function has been used successfully within the ACL2 community. For example, [40] describes a (simplified) model of JVM in ACL2 using a similar methodology. We note that in Section 5, we will discuss a different approach to specifying system models, using *Kripke Structures*. The principal difference between a Kripke Structure representation and the model described here is that in the former, the next state is specified by a *relation* instead of a function. In particular, Kripke Structures provide the concept of a *transition relation* $R \subseteq S \times S$, and two states are related by this relation if it is possible to go from the first state to the second in one step. Kripke Structures have been used in the automatic verification community as the framework to specify and study system models. In particular, reasoning about Kripke Structures often obviates the need for formalizing the notion of inputs. However, for specifying large systems, it is often better to model the systems using the next state as a function rather than a relation for two reasons. First, our representation is closer to the implementations of real industrial systems. In particular, real systems do have the concepts of inputs, and formalizing system models with a next state function rather than a relation allows easier validation that our formal theory reliably reflects the actual system it models. Secondly, our approach allows efficient simulation of the system models. Recall that the logic of ACL2 is executable, and hence system models can be simulated on real values. Often such simulation can detect errors either in the actual system or on the modeling process. From a logical perspective, we note that our representation and the representation with a transition relation are semantically equivalent in the sense that one can be transformed into the other, as in fact we discuss in one direction in Section 5.1.

2.2 Well-founded Refinements

In this section, we briefly discuss the theory of well-founded refinements. Specifically, we will consider two models $\texttt{impl} = \langle \texttt{init}_I, \texttt{legal}_I, \texttt{next}_I \rangle$, and $\texttt{spec} = \langle \texttt{init}_S, \texttt{legal}_S, \texttt{next}_S \rangle$, and consider a reasoning framework that relates the executions of `impl` to the executions of `spec`. In this section, we describe the system model `spec` as the "high level system", and the model `impl` as a lower level system. Technical details of the correspondence notion described here, including a case study verifying a system model using the notion of refinements, can be found in the work of Sumners [51], and this section merely serves as a review of existing work. In our review, we focus on the informal ideas behind the concept, and state the underlying theorems. A reader interested in the technical details of this framework is urged to consult [51] or [36] for a thorough technical treatment.

Well-founded refinement allows us to correlate the executions of two systems at different levels of abstraction. To understand what we mean by such correlation, consider a model `spec` that executes only the simple assignment statement: $\texttt{i} \leftarrow \texttt{i + 1}$. Here `i` is assumed to be a positive integer. According to naive high level language semantics, the execution of this statement increments the value of the (memory) location indexed by `i`. Yet, a model `impl` executing a program formed by compiling the statement in a standard machine architecture will normally execute a sequence of instructions, possibly involving operations such as loading the corresponding memory location to a register, followed by an ALU operation and finally a storage of the resulting value. In such modeling, it is even possible that the result stored in the location pointed to by `i` after executing the corresponding instructions might not increment `i` in all cases. For example, consider the situation in which the value in the memory location indexed by `i` is the largest value that can fit in the corresponding location. Different architectures treat this situation differently; by far the most common situation is for the next value to be negative.

The observation is trivial, but shows that high-level language semantics are often simplifications of what actually happens in an actual execution of a system. The high-level semantics may be reconciled with the executions of the underlying system by asserting that such anomalous behavior never occurs in any "real execution" of the underlying system. This statement is often formalized by the notion of an *inductive invariant*.[1] Specifically, an *inductive invariant* is often defined as a unary predicate `inv` of the states of `impl` with the following properties:

1. Every initial state of `impl` satisfies `inv`.

2. If a state s satisfies `inv`, then for all inputs i, the state $\texttt{next}_I(s, i)$ also satisfies `inv`.[2]

The two statements guarantee that any state encountered in an execution of `impl` starting from an initial state satisfies `inv`, and can be proved by a simple induction on the length of the execution. Now, consider for example, a predicate `inv` that formally states a property of the form: *The value stored in the memory location pointed to by variable i is a positive integer, and does not exceed the largest value that can be stored in the location.* If indeed such a statement can be proved as an invariant as defined by (1) and (2), then we can legitimately claim that the anomaly we discuss will never occur in any execution of `impl` that we encounter. In this manner, the predicate `inv` can be thought of as a characterization of the "real executions" of the system.

Another trivial observation regarding reasoning about correlations between two different systems is that the two systems are often described at different levels of detail. In the simple example of the assignment statement above, notice that reasoning about `spec` involves reasoning about execution of *one* statement in a program containing only *one* variable. However, our discussion should make clear that an execution of the `impl` can possibly involve several steps and many registers and memory locations. The updates of any register (or memory location) other than the location `i` in `impl` are uninteresting for the purpose of relating the two systems. Let us call the memory location indexed by `i` to be the *observable* of `impl` at any state; correspondingly, call the value of variable `i` to be the *observable* of `spec` at any state. Hence, a statement correlating the behaviors of two systems should satisfy the following requirements:

1. The correspondence notion must relate the observables of the two systems at corresponding states.

2. A `step` of `spec` corresponds to several `steps` of the `impl`.

To address the first requirement, the approach is to augment the system model by defining functions L_S and L_I for the two systems `spec` and `impl` respectively. Such functions are called *state labeling functions*, and map every state of the corresponding system to the observables in the state. For the present, we do not formally define the notion of observables. An *augmented system model* is a pair $\langle M, L\rangle$, where M is a system model and L is a state labeling function. Our notion of correspondence crucially depends on labeling functions. Whenever we discuss correspondence between models, we will have in mind two augmented models $\langle \texttt{impl}, L_I\rangle$ and $\langle \texttt{spec}, L_S\rangle$. However, when no confusion arises, we will be tempted to simply refer to the augmented system models as `impl` and `spec` respectively, and refer to them as simply *models*. In Section 5.1, we discuss the notion of state labeling functions in the setting of Kripke Structures built out

[1] For the purpose of this paper, whenever we talk about an"invariant", we mean an "inductive invariant".

[2] We note that technically, $\texttt{inv}(\texttt{next}(s, i))$ is required only under the additional assumption of $\texttt{legal}(s, i)$. However, in practice, we have not found the restriction necessary.

of system models. In terms of Kripke Structures, the observables are often defined as a collection of *atomic propositions* and the state labeling function maps a state to the set of atomic propositions that are `true` at that state. In the current context, such observables can be treated as values of a subset of state variables which are "interesting" to the high level specification. It should be clear from the context that in the trivial program of our example, the labeling function maps every state to the value of the variable `i` at the state in the `spec`, and the value stored at the memory location corresponding to `i` in `impl`.

The states of the two models, `spec` and `impl` are related by *representative functions*. The representative function is simply a unary function `rep` that maps a state of `impl` to a state of `spec`. The notion of correspondence between some state s of `impl` and the state $\texttt{rep}(s)$ is then justified by showing that the observables of the two systems are same.

- For every state s such that $\texttt{inv}(s)$ holds, $L_I(s) = L_S(\texttt{rep}(s))$.

We remark that the notion of labeling functions in a correspondence proof is crucial in the following sense. Consider two (trivial) labeling functions L_I and L_S that map every state to `true`. The above notion of correspondence is then trivially satisfied for any two system models `impl` and `spec` augmented by such a labeling function. Hence the notion of correspondence we describe below really shows correspondence between two augmented system models. It remains for the user to check if the labeling functions do preserve the set of interesting variables on both systems.

The question of correlation of executions is addressed by showing correspondence between "next states" of corresponding states. Consider a state s of `impl`, and the corresponding state $\texttt{rep}(s)$ of `spec`. Consider the state $\texttt{next}_I(s,i)$ obtained by "stepping" `impl`. Ideally, we would like this state to correspond to a possible next state of `spec`. That is, we would want the following property:

- $\texttt{rep}(\texttt{next}_I(s,i)) = \texttt{next}_S(\texttt{rep}(s), j)$, for some `legal` input j.

However, since according to requirement (2) we wish to correspond several steps of `impl` to a step of `spec`, the framework needs to allow for a possibility of *finite stuttering*. The idea is to show that if for a state s of `impl`, the state $\texttt{next}(s,i)$ maps either to $\texttt{rep}(s)$ or to $\texttt{next}_S(\texttt{rep}(s), j)$. Hence, for every step of `impl`, `spec` can be assumed to either take a "matching step", or to "stutter" at the same state. To ensure that the stuttering is finite, we further need to verify that `spec` eventually takes a matching step. To specify such a requirement in ACL2, it is often convenient to exhibit a mapping m from the states of `impl` to a well-founded set W under some binary relation $<$, and proving that $m(\texttt{next}_I(s,i)) < m(s)$, every time $\texttt{rep}(\texttt{next}_I(s,i))$ is not equal to $\texttt{next}_S(\texttt{rep}(s), j)$ for some input j [2]. A convenient example of a well-founded set in ACL2 is the set of *ordinals* [4] up to ϵ_0, with $<$ taken to be the $<_{\epsilon_0}$ operator on such a set. In fact, the set of ordinals also plays a pivotal role in verifying termination of functions defined as formal theories in the ACL2 logic [27].

In order to simplify reasoning about the stuttering requirements, one typically defines a predicate `commit` that maps the "current state" and "current input" of `impl` to a boolean. Informally, the predicate `commit` is used to decide whether `spec` should take a matching step for the current `impl` step, or it should stutter.

Formally, for two models `impl` and `spec` under labeling functions L_I and L_S respectively, we say that `impl` is a *well-founded refinement* of `spec`, denoted `impl` $>>$ `spec`, if there exist functions `rep`, `commit`, m and `inv` with the following properties:

1. $\texttt{inv}(s)$ holds for every initial state s of `impl`.
2. For any state s such that $\texttt{inv}(s)$ and any input i, $\texttt{inv}(\texttt{next}_I(s,i))$ must hold.
3. For any state s such that $\texttt{inv}(s)$, $L_S(\texttt{rep}(s)) = L_I(s)$.
4. $m(s)$ must be an ordinal for every s.
5. For any state s and input i such that $\texttt{inv}(s)$, $\texttt{legal}_I(s,i)$, and $\texttt{commit}(s,i)$, $\texttt{rep}(\texttt{next}_I(s,i)) = \texttt{next}_S(\texttt{rep}(s), j)$ for some input j such that $\texttt{legal}_S(\texttt{rep}(s), j)$.

6. For any state s and input i such that $\texttt{inv}(s)$, $\texttt{legal}_I(s,i)$, and $\neg\texttt{commit}(s,i)$, $\texttt{rep}(\texttt{next}_I(s,i)) = \texttt{rep}(s)$.

7. For any state s and input i such that $\texttt{inv}(s)$, $\texttt{legal}_I(s,i)$, and $\neg\texttt{commit}(s,i)$, $m(\texttt{next}_I(s,i)) <_{\epsilon_0} m(s)$.

We notice that the notion of well-founded refinements is compositional, in the sense that if $A >> B$ and $B >> C$ for models A, B, and C, then $A >> C$. Further, as we will note in Section 5.1, the system models can be reduced to Kripke Structures under an appropriate notion of labeling functions. In that context it can be shown that if a model `impl` is a well-founded refinement of `spec`, then any $(\text{ACTL}^* \setminus X)$ property satisfied by `spec` is also satisfied by `impl`. In certain cases, it is possible to show that the two systems are equivalent with respect to any $(\text{CTL}^* \setminus X)$ property.[3] We do not discuss the theory here, but it follows directly from the results in [44] and [37]. For now, we note that the crucial advantage of the framework follows from the observation that all the requirements in the characterization of well-founded refinement involve reasoning about no more than single steps of at most two models. This approach, therefore, has methodological advantages, obviating the need for induction over the executions of the model for any of the proof obligations. In practice, once the relevant functions have been defined, it is sometimes possible to prove some of the theorems by simple computation of the next state function on concrete values [42].

2.3 Fair Environments

In this section, we discuss how fairness can be imposed as a constraint in correspondence proofs between two systems using well-founded refinements. We will illustrate the use of fairness constraints in this context in example proofs in Section 2.4. The idea of integrating fairness constraints in well-founded refinement framework is due to Sumners, and has recently been considerably extended and expanded to include stronger notions of fairness [53].

The notion of *fair executions* plays an interesting part in proofs of correspondence between models. To understand this, consider again two system models, `impl` and `spec`. As we mentioned before, the model `impl` is often taken as an "implementation" of some system, and `spec` is often chosen as the "specification". In this context, showing correspondence is equivalent to showing that the executions of the implementation satisfies its specification. Once such correspondence is verified, it is up to the user of `impl` to check if the specification satisfies his/her requirement. As a result, it is imperative that `spec` be an extremely simple model whose "correctness" can be checked by inspection. In this context, therefore, consider analyzing a specific model `impl` such that only the "well-formed" executions of `impl` satisfy a certain property. To analyze such a system, we have two choices in coming up with the model `spec` for the purpose of showing correspondence. One option is to come up with some model which also has the requisite property, namely, that only *some* executions of `spec` satisfy the desired property. The other option is to come up with a `spec` such that *all* the executions of `spec` do satisfy the property of interest, and characterize the correspondence proof by showing that well-formed executions of `impl` correspond to executions of `spec`.

In practice, the balance often tilts in favor of the second approach. The chief reason is that the `spec` program we can come up with in the second approach is normally simpler than that in the first. Consider, for example, a mutual exclusion algorithm in a distributed multiprocessor system model. Assume that certain executions of the system, for example those that satisfy some well-defined real-time requirements, actually satisfy the mutual exclusion, and the rest do not. To analyze such an algorithm it is normally more convenient to have a `spec` program that clearly implements mutual exclusion, and characterize the executions of `impl` that do satisfy the real-time constraint.

In models of multiprocessors, a specific notion of "well-formed"-ness that often comes into play is *fairness* [17, 39]. Fairness constraints are particularly important in justifying progress properties of the underlying system. We will see the use of fairness requirements in verifying progress properties of synchronization protocols in Section 2.4, but fairness is important in other contexts. For example, consider a processor

[3] We will not use the ACTL* or CTL* logic here, except for a passing references. However, we will note at this point that thet are expressive temporal logics that often often used to specify properties of finite state system models. In particular, ACTL* is a subset of CTL*, and CTL* is strictly more expressive than the LTL logic that we will consider in Section 5.

executing instructions in pipelined stages. If some pipelined stages can be stalled indefinitely, then fairness constraints are often required to justify that the pipeline stages eventually make progress.

To describe an approach towards simulation of fairness constraints in the context of well-founded refinements, we will consider a specific model $M = \langle \texttt{init}, \texttt{legal}, \texttt{next} \rangle$, and infinite executions of M. Recall that an execution of M is a sequence of states such that for any two consecutive states s_i and s_{i+1} in the sequence, there exists an input j such that $\texttt{legal}(s, i)$, and $\texttt{next}(s_i, j) = s_{i+1}$. Our notion of fairness can be roughly stated as follows: If s is a state that occurs infinitely often in some infinite execution π of M, and if there exists an input i, such that $\texttt{legal}(s, i)$ and $\texttt{next}(s, i) = t$, then π will be called *fair* if and only if t immediately follows s infinitely often in π. We note that our notion of fairness corresponds to the notion of *weak fairness* in [39], reformulated in terms of models.[4]

We now discuss how the notion of fair executions can be incorporated in the framework of well-founded refinements. To understand the notion, we treat M as a "closed system", by augmenting every state of M with an extra component `env`, which provides a legal input for s. More specifically, we assume the existence of a function $\texttt{env}(s)$, called the *environment* of s, such that for all s, $\texttt{legal}(s, \texttt{env}(s))$. The function `next` will, then be assumed to use $\texttt{env}(s)$ as the input in state s. Finally, we assume that there is a unary function `upd` such that $\texttt{env}(\texttt{next}(s, \texttt{env}(s))) = \texttt{upd}(\texttt{env}(s))$. We discuss the properties of the function `upd` now.

To understand the nature of `upd` for fairness guarantees, we examine the definition of fairness more closely. Consider a particular state s in a fair execution π, and assume that there exists an input i such that $\texttt{legal}(s, i)$, and $t = \texttt{next}(s, i)$, but $\texttt{env}(s) \neq i$. Using the definition of fairness, we note that to achieve fairness, `env` must produce i after a finite number of steps. Specifically, there exists a number $n(s)$, such that $\texttt{upd}^{n(s)}(\texttt{env}(s)) = i$. However, there is a convenient approach to verifying this requirement in ACL2. In particular, this is guaranteed if the properties 1 and 2 are satisfied:

1. There exists a mapping f from the range of `env` to some well-founded set W under relation $<$.
2. For any state s and input i such that $\texttt{legal}(s, i)$ holds, if $\texttt{env}(s) \neq i$, then $f(\texttt{upd}(\texttt{env}(s))) < f(\texttt{env}(s))$.

From properties of well-founded sets, as we described in Section 2.2, it follows that if we can define function `upd` with the above properties, then an execution using the output of `env` as corresponding inputs will be fair.

To be able to define the function `upd`, we will first restrict the range of `legal` to be countable. In particular, then, the range of `env` is countable. The existence of function `upd` then follows from the following lemmas.

Lemma 1 *There exists a bijection from any countable set to the set* $\mathbb{N}$ *of natural numbers.*

Lemma 2 *The set* $\mathbb{N}$ *of natural numbers is well-founded under relation the ordinary* $<$ *relation.*

Lemma 3 *There exists a sequence* σ *of natural numbers, such that for every* $n \in \mathbb{N}$, n *appears infinitely often in* σ.

Lemmas 1 and 2 are standard theorems in number theory. Lemma 3 can be proved by exhibiting an actual sequence σ.

Notice again, that the fairness constraints have been stated in terms of constraints involving no more than one step of M. In using the fairness constraints in actual proofs of well-founded refinements in ACL2, the fairness constraints are specified as constrained functions `env`, `upd` and `f`, posited to have the properties i and 2. Exhibiting soundness of such constraints involves exhibiting "witness functions" satisfying the constraints, based on proofs of Lemmas 1, 2, and 3.

[4] [39] discusses two other notions of fairness, namely *minimal progress* and *strong fairness*. It is clear to see that in our reformulation in terms of models, every infinite execution of M satisfies minimal progress. Further, Sumners [53] also discusses how to formulate strong fairness in the context of stuttering refinement proofs in ACL2. We do not discuss the notion here, since for the multiprocessor proofs we look at, the notion of weak fairness is normally sufficient, as will be illustrated in Section 2.4.

```
1:  choosing[i] ← true
2:  pos[i] ← 1 + max(pos[1], ...  pos[n])
3:  choosing[i] ← false
4:  for j = 1 to n do
5:    if choosing[j] ≠ false then
6:      go to 5
7:    end if
8:    if (pos[j] ≠ 0) and ((pos[j],j)<(pos[i],i)) then
9:      go to 8
10:   end if
11: end for
12: ⟨critical section⟩
13: pos[i] ← 0
14: ⟨non-critical section⟩
15: go to 1
```

Figure 1: The Bakery algorithm for processor i. For pairs (a, b) and (c, d), the $<$ relation is defined as follows: $(a, b) < (c, d)$ iff either $a < c$, or $a = c$ and $b < d$. Note: The line numbers do not correspond to the program counters but only to the different conceptual steps of the algorithm.

2.4 A Case Study: The Bakery Algorithm

We now sketch the use of fair well-founded refinements to verify multiprocessor systems. We note first, that our exposition of the example in this paper is by no means complete, and is only for the purpose of demonstrating the basic process. More detailed report of this and other proofs will appear in the thesis. Our example is a shared memory system, with processes synchronizing by implementing a simplified version of the *Bakery Algorithm* [33]. Well-founded refinements have been used, additionally, (by us and others) to verify several other concurrent algorithms, concurrent protocols (for example the mutual exclusion algorithm by Fisher, an implementation of the Alternating Bit Protocol [37] and a concurrent deque implementation in [51]), and pipelined processors (for example in [36]).[5]

The Bakery algorithm is a concurrent protocol often implemented in multiprocessor systems to achieve synchronization between processes. The goal of synchronization algorithms is to achieve "mutual exclusion" and "progress". Mutual exclusion guarantees that two processors cannot enter the *critical section* of the program at any time. Progress ensures that every contending process will eventually enter the critical section.

To describe the Bakery algorithm, we follow the original presentation from [33]. The protocol for processor i, $i = 1..n$, is shown in Figure 1. The algorithm is based on one commonly used in the bakeries, in which a customer receives a number upon entering the store, and the holder of the lowest number is the next one served. Consider the system of n processors with indices $1..n$. We assume two common arrays `choosing[1..n]`, and `pos[i..n]` with `choosing[i]` and `pos[i]` being in the memory of processor i. In particular, `choosing[i]` and `pos[i]` can be written only by processor i, but can be read by other processors.

The Bakery Algorithm is subtle in analysis and hence, a suitable candidate to verify using the well-founded refinement framework. To reason about the algorithm, we model the code in Figure 1 fairly accurately but realistically, as an ACL2 function, that creates the model `impl` for our implementation.[6] The next state function for the model non-deterministically picks a process to execute its next instruction, corresponding to the code in Figure 1.[7] The initial states correspond to the states in which `choosing[i]` is `false` and

[5] The notion of correspondence used in the verification of the Alternating Bit Protocol, and pipelined processors [36] is slightly different from well-founded refinements in that they use well-founded *bisimulation* (WEB) techniques. However, the central idea is similar. In fact, research on well-founded refinement for verifying multiprocessor systems was initiated to apply the notion of WEBs on the refinement framework using system models instead of Kripke Structures.

[6] Our implementation is not exact. In particular, we replace the asynchronous computation of max by a specific global variable `max` that keeps track of the current maximum value in the system, and is atomically incremented by every process executing line 2. The atomic update is implemented using "compare and swap" operations available in many architectures.

[7] The instructions do not correspond exactly to the code in the sense that they have been implemented by lower level

pos[i] is 0 for every i. For simplicity, we allow the legal inputs to be any natural number, and the current input determines the index of the process scheduled to take the next step. The labeling function keeps track of which line of code each process has executed. In particular, we divide the code into different sections and associate the following status to a process. A process executing lines 3 through 11 is assigned the status wait; a process executing line 12 is assigned the status go; otherwise a process assigned the status idle. The labeling function maps every process to its status.

Our specification spec is a simple model that captures the essence of the algorithm. In particular, the specification can take the following steps:

1. an idle process takes a step by being added to a queue and changing its status to wait. The position in queue for a particular process i corresponds to the number of processes j having $(\texttt{pos[j]}, j) < (\texttt{pos[i]}, i)$.

2. A process with status wait can make progress if it is only at the head of the queue, in which case it enters the critical section by updating its status to go; otherwise it does not change its status.

3. A process in the critical section is removed from the queue and has status idle.

To facilitate the proof of correspondence between the implementation and the system, we create an *intermediate model* impl+ by augmenting the implementation model with a queue which is updated according to 2, and 3, upon completion of lines 1, 11, and 12, respectively. Notice that queue is only updated and never accessed, and therefore, only provides auxiliary information for reasoning about the system with no effect on the program behavior. The queue in the intermediate model, thereby, acts as a *history variable* [34]. Since such history variables do not affect program behavior, the verification impl $>>$ impl+ is relatively simple to demonstrate.[8] For the remainder of this presentation, we will therefore focus on the proof of impl+ $>>$ spec.[9]

We note that the implementation follows the specification above only for fair executions. In particular, consider a process i attempting to execute line 5 of the code in Figure 1, and checking choosing[j]. Assume that process j has just executed line 1, setting choosing[j] to be true, but is never scheduled. Then process i will forever remain in the loop. Thus even when process i is at the head of the queue, it cannot make progress as required by the specification.

We now discuss the basics of the proof. For practical proofs with stuttering refinement, it is convenient to define a unary predicate suff and replace the proofs of steps 5 through 7 in Section 2.2 by the following four steps.

1. For all states s, $\texttt{inv}(s)$ implies $\texttt{suff}(s)$.

2. For any state s and input i such that $\texttt{suff}(s)$, $\texttt{legal}_I(s,i)$, and $\texttt{commit}(s,i)$, $\texttt{rep}(\texttt{next}_I(s,i)) = \texttt{next}_S(\texttt{rep}(s), j)$ for some input j such that $\texttt{legal}_S(\texttt{rep}(s), j)$.

3. For any state s and input i such that $\texttt{suff}(s)$, $\texttt{legal}_I(s,i)$, and $\neg\texttt{commit}(s,i)$, $\texttt{rep}(\texttt{next}_I(s,i)) = \texttt{rep}(s)$.

4. For any state s and input i such that $\texttt{suff}(s)$, $\texttt{legal}_I(s,i)$, and $\neg\texttt{commit}(s,i)$, $m(\texttt{next}_I(s,i)) <_{\epsilon_0} m(s)$.

primitives. For this section, though, we will pretend as if the instructions executed correspond exactly to the corresponding lines of code.

[8] We can actually demonstrate impl $\leftrightarrow$ impl+, where $A \leftrightarrow B$ is a shorthand for "$A >> B$ and $B >> A$".

[9] In our verification effort, we actually create an even simpler specification spec+ which simply non-deterministically chooses a process and allows it to go to critical section. When the process finishes, another process is allowed to enter. This specification is created to reflect the essence of the mutual exclusion algorithm. However, the proof of spec $>>$ spec+ is also trivial based on the description of spec. Our observation that it is relatively simple to demonstrate the correlation between a model and an augmented model with auxiliary variables, supports similar comments by Sawada [50] in the context of multiprocessor proofs. In particular, the use of intermediate model impl+ is a direct consequence of Sawada's work.

It should be clear that proofs of steps 5 through 7 in Section 2.2 follow from the steps above. However, this approach decouples the proofs relating the two models `impl` and `spec` from the construction of the predicate `inv` purported to be an invariant for model `impl`. In particular, we will call a proof of these steps along with steps 3 and 4 in Section 2.2 a proof of well-founded refinement "up to sufficient condition". In particular, these steps relate the steps of two different systems, and can be achieved without constructing the invariant.[10]

To prove stuttering refinement up to sufficient condition in the context of the verification `impl+` $>>$ `spec`, we define the predicates `commit` and `suff` as follows. We note that the specification makes a step corresponding to the code if the scheduled process completes either line `8` or `12` or `13`. Hence, $\texttt{commit}(s, i)$ is `true` at a state if and only if process i is about to complete one of these operations. Otherwise the predicate is `false`. The predicate `suff` simply characterizes how the processes behave with respect to the queue corresponding to `pos`. In particular we associate the corresponding assertions for the different lines of code for process i, $i = 1 \ldots n$.

1. If process i is currently executing line `1` it is not in `queue`.
2. If process i is currently executing line `8` and j is n and $(\texttt{pos[j]}, j) < (\texttt{pos[i]}, i)$, then the process is at the head of the queue.
3. If the process is currently executing line `12`, then the process is at the head of the queue.

The proof up to sufficient condition for `impl+` $>>$ `spec` now follows from the definition of the next state functions of the two programs. We note here, that the proof of requirement 4 is non-trivial. In particular, it is non-trivial to come up with a definition of m satisfying the condition. However, reflection indicates that a process advances in the `queue` since in fair executions, the process at the head of the queue is eventually selected to take a step.

Notice that, even with the complications in defining m, the process of verification up to sufficient condition is fairly straightforward. In particular, coming up with the condition `suff` merely entails relating the corresponding steps between `impl+` and `spec`, and the predicate simply enforces the conditions on which `spec` is required to execute, on `impl+`. This process, in particular, has yet required no insight on how the algorithm works. However, coming up with the predicate `inv` is more complicated. The complication is due to the fact that `inv` is required to hold for every step of `impl+`, and has to be constructed as a "global predicate" `true` for all processes, not simply on the steps that match `spec`, nor on the specific process that takes the step. It is difficult to reconstruct the reasoning about every observation that goes in the construction of an invariant, but let us look at a specific example. Consider the property that will be required to be true in order to guarantee item 3 of the `suff` predicate for a process. We note that item 3 has indicated that the pair $(\texttt{pos[i]}, i)$ is least for process i than any other process. A slight reflection might suggest that since process i checks each other process for this quantity before advancing to the head of the queue, a possibility is that the queue is ordered according to this quantity. The reflection and the corresponding intuition has value, but requires some work to establish as invariant. In particular, notice that a process enters into the queue at line `3`. Since the process is at the tail of the queue on entry, we then need to prove that the process has the largest value of $(\texttt{pos[i]}, i)$ among every process in the queue. This might seem "obvious" since the process just computed the max among the `pos` values of every processes, and hence, in particular, every process in the queue. Yet, demonstrating this adds the proof obligation that no process executing lines `1`, or `2` can possibly be in the queue.

The informal description of the reasoning process above should demonstrate that construction of invariants provides the crux of the proof of stuttering refinement. For the simplified example we considered, it is relatively simple to come up with the appropriate conjunction of properties that can be demonstrated as invariants. However, even for this example, coming up with the appropriate predicate constituted bulk of the proof process. Further, the invariants change drastically if the model is slightly refined. For example,

[10] We should note that the step 3 for Section 2.2 does specify the invariant. However, as noted by Sumners [51], the hypothesis $\texttt{inv}(s)$ is hardly ever required for that proof.

consider the slightly more advanced model in which the computation of the max is achieved by reading `pos` of every process sequentially.[11] Such a system does not, in fact, guarantee that the value of `pos[i]` is the maximum among the processes currently in the queue. Yet, it is possible to come up with a set of invariants, possibly weaker, and demonstrate the well-founded refinement above. Experiments (by us and others) with different systems indicate that the invariants become progressively complicated as the implementation system is refined and elaborated.

3 Invariant Proving

As we discussed in Section 2.4, a large portion of the manual effort involved in a well-founded refinement proof goes into the definition and checking of invariants. In this section, therefore, we discuss procedures that assist the user in this process. We note that while we focus in this presentation on invariant checking as a component of a well-founded refinement proof, invariant checking is important and interesting in its own right. An interesting class of properties, called *safety properties*, can be roughly described as an assertion of the form: *The system never does anything bad.* A typical proof of such a statement often follows by showing: (1) the initial states are not "bad", and (2) if s is not a "bad state", then for every i, $\texttt{next}(s, i)$ is not a bad state either. In other words, the assertion "not bad" is thereby proved to be an invariant on any execution of the system starting from an initial state.

In hardware and systems verification, safety properties play a pivotal role. For example, important specification properties for multiprocessor systems, like mutual exclusion, cache coherence, and so on, are essentially safety properties, and checking such properties for real systems is tantamount to verifying some invariant holds for the system. We saw an example of the use of invariants in the verification of mutual exclusion properties for the Bakery implementation in Section 2.4. We should note that our proof verified both safety and progress properties. But as we illustrated, even in such a comprehensive verification, invariant checking often provides the maximum challenge to the success of the proof effort.

In this section, we therefore, explore ways of checking invariants. In Section 3.1, we provide a rigorous framework. In particular, our framework allows us to strengthen a "sufficient condition" `suff` into an invariant `inv` as we described in Section 2.4. In Section 3.2, we discuss some aspects of our tool and ongoing invariant checking using the procedure. In Section 3.3, we summarize our approach and discuss related research.

3.1 Invariant Proving Framework

In this section, we provide a framework for reasoning about invariants. For the purpose of invariant proving, we will assume we have been provided with a fixed model $M = \langle \texttt{init?}, \texttt{legal}, \texttt{next} \rangle$. Given a model M, and a unary predicate `good`, our object is to determine a unary predicate `inv` with the following properties:

- For all s, $\texttt{init}(s)$ implies $\texttt{inv}(s)$.
- For all s, $\texttt{inv}(s)$ implies $\texttt{good}(s)$.
- For all s and i such that $\texttt{inv}(s)$, $\texttt{inv}(\texttt{next}(s, i))$.

For this section, we will make the simplifying assumption that the model M has exactly one initial state, and the specification of predicate `init?` is provided by the description of the initial state, which we call `*init*`. We note that it is easy to relate the notion of invariants in the current framework with the notion of well-founded refinements that we described in Section 2. In particular, note that the predicate `suff` we discussed in Section 2.4 directly corresponds to the predicate `good` in the general framework. Recall that we argued in Section 2.4 that a proof showing stuttering refinement up to sufficient condition is relatively simple, and the manual effort is manifested in the "strengthening" of the predicate `suff` into `inv` so that

[11] In fact, it is for such systems that the lexicographic pair with the process index comes into play. For the simple system we discussed above, the second parameter used is irrelevant, and was left simply for demonstration.

the predicate `inv` is an invariant. In that context, our work in this section can be viewed as designing a procedure to assist the user in the strengthening work.

The idea behind such an invariant strengthening procedure is to use term rewriting to construct a finite model corresponding to the given system, which can then be analyzed by automatic analysis. In particular, we will create an abstract system M' such that the states of M' are simply lists of booleans. To elaborate on this, we provide some notational conventions. We assume a fixed collection `pred` of predicates, which include unary or binary functions, and assume that $\texttt{good} \in \texttt{pred}$. For the procedure for verifying and strengthening invariants, we will find it convenient to talk about *operators* on functions. In the sequel, we will call the operator $\circ$ the *function composition operator*, and treat $f \circ g$ as the composition of functions f and g. For example, $(\texttt{good} \circ \texttt{next})(s, i)$ is simply $\texttt{good}(\texttt{next}(s, i))$. For any two functions f and g, we will write $f \doteq g$, to mean f and g are identically equal. In particular, if f and g are unary functions, then $f \doteq g$ means that for all s, $f(s) = g(s)$. We will say C is an *n-ary boolean composition* if C is a predicate of n arguments such that $C(x_1, ..., x_n)$ is simply a boolean combination of its arguments. Further, for a given boolean composition C, we will present its parameters within angular brackets $\langle\rangle$. In particular, if C is a binary predicate, the expression $C\langle f, g\rangle$ is used to mean $((C \circ f) \circ g)$.

Our invariant proving procedure uses a *basic simplification process*. To understand the process, we first attempt to prove that `good` is an invariant. The goal of the simplification process is to determine a set of predicates P_0, P_1, P_2, ..., P_n in `pred`, and corresponding boolean compositions C_0, C_1,...C_n, such that the following requirements 1 and 2 are satisfied.

1. $\texttt{good} \doteq P_0$.
2. For every P_i, $i = i, ..., n$, $(P_i \circ \texttt{next}) \doteq C_i\langle P_0, P_1, ..., P_n\rangle$.

We can then define an *abstract model* M' as follows. We will associate with each predicate P_i a variable v_i, for $i = 0...n$. The states are simply the all possible boolean assignments of these variables. The next state function `Abs` updates variable v_i to be $C_i(v_0, v_1, ..., v_n)$. The initial state init_A is the boolean assignment of variables such that v_i is assigned to $P_i(\texttt{*init*})$.

The following lemma justifies that the checking of invariant can be transferred to the abstract model.

Lemma 4 *The predicate* `good` *is an invariant in M if and only if v_0 is* `true` *for all states in M' reachable from $init_A$.*

However, notice that the model M' has a finite number of states. Hence, checking if v_0 remains `true` can be solved with an automatic analysis using model checking techniques [14]. We will not discuss that procedure here, but we comment that such invariant checking procedures are available with many commercial model checking tools, for example SMV [38], and VIS [54]. In particular, we have implemented a simple invariant checker in ACL2 that explicitly checks properties of the finite models we generate. Since our concern is on generation of effective and small finite models, our checker is relatively inefficient, but still sufficient for our purpose. We do not anticipate to frequently produce finite models having a large number of reachable states. However, for such occasions, we have also implemented a tool integrating our procedure with VIS [54] to enable effective checking of invariants for large models.[12]

Now we consider the more general scenario, in which we are searching for an invariant predicate `inv` which implies `good`. We modify the requirements of the *simplification process* to consider this more general situation. In particular, we now require the simplification process to generate, in addition to predicates P_i and C_i, for $i = 0 \ldots n$, another collection of predicates $Q_{i,j}, j = 1 \ldots n_i$ in `pred`, such that the following requirements are satisfied.

1. $\texttt{good} \doteq P_0$.
2. For every P_i, $i = 0, ..., n$, $(\wedge_{j=i}^{n_i} Q_{i,j}) \Rightarrow (P_i \circ \texttt{next}) \doteq C_i\langle P_0, P_1, ..., P_n\rangle$.

[12] This tool is still under construction and has not been extensively tested yet.

We will then attempt to prove that the predicate $(\wedge_{i=1}^{n_i}(\wedge_{j=i}^{n_i} Q_{i,j}) \wedge \texttt{good}$ can be strengthened to be an invariant. This is justified by the following lemma.

Lemma 5 *The predicate $f \wedge g$ is an invariant for model M if the following condition holds.*

1. *$f(\texttt{*init*})$ and $g(\texttt{*init*})$ hold.*
2. *$f(s) \wedge g(s) \Rightarrow f(\texttt{next}(s,i))$.*
3. *$f(s) \wedge g(s) \Rightarrow g(\texttt{next}(s,i))$.*

Lemma 5 provides the following recursive approach to generating and proving invariants. The approach is to apply the simplification process to create the finite model M', thereby incurring the determining predicates $(\wedge_{j=i}^{n_i} Q_{i,j})$. Then we can apply the finite model M' to verify `good`, and finally apply the process recursively for each of the predicates $Q_{i,j}$. The recursive process will probably accrue more predicates. Assuming that the process terminates, Lemmas 4 and 5 guarantee that the conjunction of all the predicates we accrue in the process is an invariant.

3.2 An Invariant Prover

We have discussed the basic procedure that can refine and verify invariants. We have implemented a tool in the ACL2 theorem prover, based on the above approach. The approach has been implemented in the context of ACL2 theorem proving, where the simplification process we have discussed can be achieved using the simplification engine of the theorem prover. We assume that the user can provide the set of predicates `pred` for helping the process of simplification. As is customary in ACL2, the simplification uses rewrite rules proposed by the user and verified by the theorem prover. The efficacy of the tool depends on the presence of effective rewrite rules to guide the simplification process.

Our tool has been optimized with several heuristics to guide the process of generation of the abstract model. Space does not permit discussion of the heuristics, some of which involve technical details of the internals of the ACL2 theorem prover. We will provide a description of our strategies in the thesis. The strategies are implemented to make the tool efficient without compromising the soundness of the basic procedure described above.

We are currently applying the procedure to generate and prove invariants on a multiprocessor memory system. The system contains n processors, with one level of cache at each processor, pipelined instruction execution, and delayed writes to the memory. The model has been implemented in ACL2 and can be obtained upon request from the author. Our initial experiments on applying the tool on our model suggest that the checking of invariants can be done efficiently even on complex system models if effective rewrite rules are present to guide the simplification process.

We remark on the importance of rewrite rules in this context. Loosely speaking, rewrite rules are theorems the user provides the tool in order to help in simplifying terms built out of the composition of predicates in `pred` and operators building the next state function `next`. Such rules are verified by the ACL2 theorem prover to guarantee soundness of their applications. In order to effectively use the tool, the user needs to have an understanding of the terms the simplification process is likely to encounter in the process of generating finite models. The simplifier provides feedback to the user showing terms that failed to simplify to requisite forms. However, on viewing such a failed simplification, the user needs to reflect on the failure to devise the appropriate rules to allow the simplification.

Our discussion seems to indicate that the tool puts the onus on the user to verify effective invariants. In practice, however, that is not the case. In particular, rewrite rules have been traditionally used in ACL2 and other theorem provers for effective theorem proving [29]. The advantage of this approach is that the libraries are normally reusable, and hence the same collection of rules can be used to check invariants of related system models. For example, in our example multiprocessor system, we modeled the entire system using operations involving "sets" and "records" [32], implemented in ACL2 and available with the distribution. An implementation of a data structure in ACL2 normally involves a collection of theorems specifying how terms involving such structures can be simplified. Since our tool uses the simplification engine of ACL2, such theorems can be effectively used by the tool in generating abstract models. As an aside, we anticipate that wider use of our tool will provide effective feedback on the efficacy of the rules currently available for data structures implemented in ACL2, and hence facilitate building of better libraries.

A particularly important feature of our tool is that the complexity of the simplification process is independent of the number of states in the actual model M. Indeed, the tool is applicable even if the number of states in M is infinite. The complexity is rather dependent on the size of the intermediate terms generated in the simplification process. Use of effective rewrite rules can control the growth of large terms, and hence enable us to generate effective models.

3.3 Summary and Related work

We have discussed an approach to implement a simple rewriting strategy for specifying and strengthening invariants in ACL2. We note that the importance of discovery and strengthening of invariants has been widely recognized in the formal verification community. For example, the STeP system [6] provides techniques for strengthening and discovery of invariants. Invariant strengthening is known to be one of the major bottlenecks in large-scale system verification [51, 50]. Several approaches have been used to strengthen invariants, based on model checking and counterexamples [12, 13]. The STeP theorem prover [6] provides explicit strategies for invariant discovery and strengthening. Further, PVS [45] uses a tactic called f *predicate abstraction* [21] that uses the theorem prover to discover invariants on system models.

The chief difference between our strategy and the strategies in STeP arise from our use of a general purpose logic. In ACL2, the system models are specified using an operational model as described in Section 2.1, and the invariants are simply ACL2 predicates applicable to any objects specified by ACL2. This disallows the possibility of having special inference rules like `G-INV` that is used in STeP to discover and strengthen invariants. Our inference rules in ACL2 take the form of standard rewrite rules, and the theorem proving engine is used to apply those rewrite rules on system models to produce and strengthen invariants. Further, our technique is more general than both the STeP and PVS strategies in that we use user-specified simplification rules in our invariant discovery. Such rules, as we discussed in Section 3.2, provide the domain-specific user insight and controls the theorem prover in generation of effective abstract models.

We are currently experimenting with verifying non-trivial invariants for system models using the invariant prover in the context of well-founded refinements. Our current results seem to indicate that the approach is useful. We are planning to verify large RTL modules using the invariant prover and analyze its efficacy in practice.

4 Automatic Verification Procedures

In this section, we turn our attention to automatic decision procedures. We discuss the logical problems associated with integrating decision procedures in a theorem prover. We consider the issues with ACL2 as our example.

Notice that the semantics of ACL2 is like a programming language, namely Applicative Common Lisp. Hence it is possible to efficiently code up decision procedures like model checking in ACL2. The theorem prover, on being given a system model and a property to be proved for the model, will then be able to invoke the procedure to verify the property. Such tactics have often been effective in the theorem proving community. For example, ACL2 has successfully used built-in decision procedures for handling linear inequalities [8], and binary decision diagrams (BDDs) for Boolean equivalence checking [52]. A problem, however, arises from the possible incompatibility of the logic implemented by the decision procedure with the logic of the theorem prover. For example, model checking procedures are used to verify "temporal properties" of finite state systems. In order to use such a procedure in conjunction with a theorem prover, it is necessary to provide the semantics of temporal properties of finite state systems inside the theorem prover.

Let us examine the incompatibility question by an example. Assume that we have a model of a computer system in ACL2 and we have coded up a CTL* [14] model checker to check the properties of the model.

A possible property that can be verified by the model-checker is of the form: *There exists an execution in which eventually variable v becomes* `true`. According to the semantics of CTL*, this property implies that there exists an infinite "execution" of the system, in which the system reaches a state in which the variable v is assigned to `true`. However, if executions are modeled as `lists` of states, as is customary in ACL2, then infinite executions are not objects inside ACL2. In fact, it is easy to prove in ACL2 that every `list` is finite, that is, for every `list` σ, there is a `list` π such that the length of σ is strictly less than the length of π. Thus, the CTL* property verified by the model-checker is not an ACL2 theorem based on the standard semantics of CTL*. In fact, if we posit the theorem as an axiom extending the ACL2 logic, the resulting logic is unsound.

Our approach to this dilemma is to define the semantics of the decision procedures as formal theories *inside* ACL2, and use the theorem prover to deduce properties of the theory. Such theorems can be treated as *characterizing theorems* of the decision procedures. In general, a characterizing theorem provides the formal statement of what can be deduced by a successful verification of a system model using the procedure. For example, in Section 6, we describe the characterizing theorem of Generalized Symbolic Trajectory Evaluation (GSTE). We discuss the theorem in Section 6, but roughly, it tells us that if we want to decide a specific property of the system model under consideration, as specified by *assertion graph*, then a successful verification of the model by GSTE is sufficient for the purpose.

We note that the characterizing theorems might sometimes be significantly different from the classical semantics of the procedure, and the proofs of such properties might differ from classical proofs. It often becomes tricky to realize what theorems can be deduced from such decision procedures. To understand this, let us examine an algorithm verified by Russinoff [49] for computing approximations of square roots of floating point numbers. A naive characterization for the procedure would have been to define a function `sqrt` that purports to compute the exact square roots of numbers, and then verify that the procedure computes an approximation of `sqrt`. However, a function like `sqrt` cannot be admitted to the logic of ACL2 as a formal theory, since the logic of ACL2 does not admit irrational numbers. Hence the proposed naive approach for verifying the square root procedure cannot be applied as is. In particular, it is simple to see that extending the ACL2 logic with an axiom characterizing the function `sqrt` leads to unsoundness in the logic. The solution provided in [49] is to characterize the square root procedure by the "square" of the value computed by the procedure. In particular, the theorem proved is of the following form: *Under appropriate conditions, difference between the square of the output of the procedure and the input is small.* We can think of the theorem as a characterizing theorem of the square root algorithm. In this case, of course, it is informally clear that the theorem does indeed reflect the semantics of square roots. In other more non-trivial situations, it is not so obvious, and justification that the characterization theorems do reflect the traditional semantics of the decision procedure might require considerable reflection and insight of an user well-versed in the logic of the theorem prover and the semantics of the procedure under consideration. For example, in Section 5, we discuss the verification of a compositional model checker based on LTL. For reasons we discuss, the semantics of LTL we formalize are considerably different from the traditional semantics of LTL, and are specified in terms of what we call *eventually periodic paths*. The justification that such characterization of the semantics of LTL is equivalent to the traditional semantics requires non-trivial observations regarding the traditional LTL semantics. On the other hand, our formalization shows precisely what is implied *in the logic of ACL2* by a successful verification of an LTL property by a model checker.

Admittedly, the specification of decision procedures inside a theorem prover like ACL2, and verification of characterizing theorems, is a non-trivial exercise. However, notice that such specification and verification are required only once for each decision procedure, and subsequent analysis of system models can then be done by simply invoking the verified decision procedure. In fact, once a characterizing theorem is proved for a decision procedure, the procedure itself can be thought of as integrated within the theorem prover to answer questions about decidable fragments of the logic. To see this, we consider the characterization theorem about Generalized Symbolic Trajectory Evaluation, that we prove in Section 6. Roughly speaking, the theorem takes the following form: *If we are given a system model and a temporal property to prove about the system, specified in a certain form of "assertion graph", then to prove the property, it is sufficient to invoke the GSTE algorithm on the system model and the graph and check if it answers* `true` *or* `false`*.* Recall that the

ACL2 theorem prover uses user-specified rules to prove theorems. Hence, once such a theorem is specified, the theorem prover on being asked to verify a property specified in the right form of assertion graphs, can simply invoke the above rule to invoke the GSTE algorithm, execute the algorithm on the system model and the graph, and check if the answer is `true` or `false`.[13] Since the logic of ACL2 is executable, and the executable portion of the logic is simply Common Lisp, the invocation of the GSTE algorithm amounts to an invocation of an *implementation* of a decision procedure to answer the question. Also, by virtue of the characterizing theorem verified by ACL2, we know that application of the GSTE algorithm in this case is sound and consistent with the logic of the theorem prover. Hence, we achieve the efficiency of using decision procedures inside the theorem prover without requiring any extra-logical tool or argument.

A question might be raised regarding the viability of implementing or reasoning about such decision procedures as formal theories inside a theorem prover. Based on our experience as documented in Sections 5 and 6, we think that it is possible, though not trivial. A further question might be asked regarding the efficiency of our implementations inside the theorem prover as compared to trusted tools implementing the procedures, as available in the market. Given that our concern is verification of large-scale systems, even the abstractions might still require highly efficient procedures to successfully circumvent the state explosion problem, and the procedures we verify are designed to simplify reasoning as opposed to efficient executability. We answer that our experience shows that once a simple implementation of an algorithm is verified mechanically with a theorem prover, it is often much less work to show functional equivalence between the simple implementation and a complex one. Further, in section 7, we propose ways of integrating trusted external procedures in the theorem prover, which can be used in the cases where efficiency of execution becomes a concern.

5 Compositional Model Checking

In this section, we discuss our experiences in verifying a simple compositional model checking algorithm in the ACL2 theorem prover. The algorithm uses *conjunctive reduction* and *cone of influence reduction* to reduce a large model checking problem into a collection of simpler problems, and we prove the soundness of composition of these reductions. The individual model checking problems are expressed in the Linear Temporal Logic. In Section 5.1, we review the classical syntax and semantics of LTL. In Section 5.2, we discuss the simple compositional model checking algorithm. In Section 5.3, we discuss the issues involved in such verification using the ACL2 theorem prover. We note that we are only presenting an overview of this work here. A detailed account of this verification effort can be found in [48].

5.1 LTL Model Checking

In this section, we briefly review the syntax and semantics of LTL, and provide an overview of LTL model checking. The purpose of this section is simply to make this paper self-contained, and we do not attempt an any way to be complete in our review of the large body of literature on model checking. The reader interested in model checking is referred to [11, 14, 16] for a detailed treatment of the ideas presented here.

The semantics of LTL are traditionally described with respect to "paths in a Kripke Structure". We, therefore, first describe a Kripke Structure and define the concept of a path through a Kripke Structure, and discuss how to represent system models as Kripke Structures.

Formally, given a collection of `atomic propositions` *(AP)*, a Kripke Structure M is given by the 4-tuple $M = \langle S, L, R, S_0 \rangle$, where S is a finite collection of *states*,[14] $L : S \rightarrow 2^{AP}$ is called a *state labeling function*, $R \subseteq S \times S$ is a left-total binary relation on S, called the *next state relation*, and $S_0 \subseteq S$ is called the *set of initial states*. Given a Kripke Structure $M = \langle S, R, L, S_0 \rangle$, a path in a Kripke Structure is an infinite sequence of states $s_0 s_1 s_2 \ldots$, such that for each $i \geq 0$, $s_i \in S$, and $(s_i, s_{i+1}) \in R$. A path $\pi = s_0 s_1 \ldots$ is

[13] The invocation of this theorem in the context of checking a GSTE property is slightly tricky in ACL2 and involves use of what are known as `:meta` rules.

[14] Recently, work has been done on Kripke Structures having infinite number of states. However, for the purpose of this and next sections, unless otherwise stated, we will concentrate only on Kripke Structures where the number of states is finite.

called an *initial path* if $s_0 \in S_0$. We now discuss a way in which system models as described in Section 2 can be represented as Kripke Structures. Recall that a state of a system model can be represented as some assignment to the set of `variables` in the system. If the number of states is finite, then the number of variables and the range of values of each variable must be finite. To represent such a system model by a Kripke Structure, we will assume that the set of atomic propositions is the set of variables in the system,[15] and the set of states to be the set of all possible evaluations of the variables. For now, we will assume without loss of generality, that the variables can only take up the two values `true` and `false`. Then the set S is the set of all possible Boolean assignments to the collection of variables in the system. Two states s and t are related by R if there is an input i such that $\texttt{next}(s, i)$ is t. Finally, the state labeling function maps every state of the system to the collection of variables that are assigned to `true` in the state. It is easy to note, now, that the set of all possible initial paths models the set of all possible executions of the system.

An *LTL formula* is either one of `true` or `false`, or a member of AP, or composed of other LTL formulas by using either (i) the boolean connectives `AND`, `OR`, and `NOT`, or (ii) unary temporal operators `G`, `F`, and `X`, or (iii) binary temporal operators `U` and `W`. A formula is `true` or `false` of a Kripke Structure and a formula is true of a Kripke Structure if and only if it is true of every initial path through the Kripke Structure. The formula `true` is true of every path, and the formula `false` is not true of any path. The formula $a \in AP$ is true of a path if a is in the label of the first state of the path. The semantics of an LTL formula created out of a boolean combination of other LTL formulas is specified by the standard semantics of boolean operators. The formula $\texttt{F}\phi$ is read "eventually ϕ" and is true of a path if and only if there exists a suffix of the path such that the formula ϕ is true. The formula $\texttt{G}\phi$ is read "always ϕ" and is true of a path if ϕ is true of every suffix of the path. The formula $\texttt{X}\phi$ is true of a path if ϕ is true of the suffix of the path starting from the second state. The operators `U` and `W` are called "until" and "weak until" respectively. The formula $\phi\texttt{U}\psi$ holds for a path $s_0 s_1 s_2 ....$ if there is an $i \geq 0$ such that ψ is true of the suffix of the path starting at s_i and for every $0 \leq j < i$, ϕ is true of the suffix of the path starting at s_j. The formula $(\phi \texttt{ W } \psi)$ is similar to $(\phi \texttt{ U } \psi)$, except that there is no requirement for ψ to hold eventually.

An *LTL model checker* is a procedure that, given a Kripke Structure and an LTL formula, returns `true` or `false` depending on the semantics of LTL. Modern LTL model checkers, for example SMV, also return a counter-example when the formula is not `true` of the Kripke Structure. The counter-example is a path through the Kripke Structure which does not satisfy the formula.

5.2 A Compositional Model Checker

A *compositional model checker* reduces a verification problem into a collection of "simpler" problems each of which can be solved by a model checker. It should be clear, that a verification problem in this context is a pair $\langle F, \psi \rangle$, where M is a finite state system and ψ is an LTL formula, and the question being asked is whether ψ is `true` of the Kripke Structure M built out of F. In this section, we consider a very simple compositional model checking algorithm composed of two *reductions*, namely, *conjunctive reduction* and *cone of influence* reduction. In this section, we first describe conjunctive and cone of influence reductions, and discuss their soundness proofs. Then we discuss our simple compositional algorithm.

Conjunctive Reduction

Conjunctive reduction is based on the observation that verification of a conjunction (`AND`) of LTL formulas can be decomposed into verification of individual formulas. The following lemma shows the soundness of conjunctive reduction, and is a standard exercise.

Lemma 6 *An LTL formula of the form* $(f_1 \texttt{ AND } f_2)$ *is true of a Kripke Structure* M *if and only if both* f_1 *and* f_2 *are true of* M.

[15] We remarked in Section 2.1 that the atomic propositions are taken to be values of a subset of state variables in the system. In the context of the reductions we analyze, it is simpler to take the propositions to be the set of all state variables in the system.

Cone of Influence Reduction

We now turn to the cone of influence reduction. Informally, cone of influence reduction can be thought of as elimination of redundant state variables. Let V be the variables in the system (and hence, by our approach, the set of atomic propositions). Assume that the LTL formula we are interested in checking is composed of a subset $V' \subseteq V$. (Recall that LTL formulas are built out of the atomic propositions (variables) using boolean and temporal operators.) We would like to simplify the description of the system by referring only to the variables in V'. However, the values of variables in V' might depend on variables not in V'. The cone of influence for V' is the minimal set of variables whose values could affect the variables in V'.

Formally, given a subset $V' \subseteq V$, the *cone of influence* of V' is the set C of variables having the following two properties:

- $V' \subseteq C$, and
- For every $v \in C$ and every input i, the value of v in $\texttt{next}(s, i)$ is a function of the values of variables in C in the current state s.

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 next state function only updates the values of the variables in V.
- 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.

The soundness of cone of influence reduction follows from the following lemma.

Lemma 7 *Let F be a finite state system and V be a subset of variables in F. Let C be the cone of influence of V and F' be the corresponding system formed by cone of influence reduction. Then any LTL formula f that refers only to the variables in C is true of the Kripke Structure built from F if and only if it is true of the Kripke Structure built from F'.*

The standard proof of Lemma 7 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 bisimulation arguments to prove soundness of reductions for LTL model checking.

Lemma 8 *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 9 *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 8 and 9, 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 8 and 9 are well-known results, and can be found in text-books on model-checking, for example [14].[16] The proof of Lemma 7 now follows by showing a bisimulation relation between the Kripke structure for the original system and the Kripke structure for the reduced system.

Compositional Algorithm

We now discuss our simple compositional algorithm. Given a finite state system F and an LTL formula φ, the algorithm performs the following steps:

1. Create a collection of verification problems $\langle F, \varphi_i \rangle$ corresponding to the conjunctive reduction of φ.
2. For each problem of the form $\langle F, \varphi_i \rangle$ let V_i be the set of variables in φ_i. Then replace the verification problem $\langle F, \varphi_i \rangle$ by the problem $\langle F_i, \varphi_i \rangle$, where F_i is produced from F by cone of influence reduction with respect to V_i.
3. Solve each verification problem $\langle F_i, \varphi_i \rangle$ by creating the Kripke Structure M_i for F_i, and model checking M_i for the formula f_i.

The soundness of our compositional model checker is given by the following theorem:

Theorem 2 *Let F be a finite state system and φ be an LTL formula. Let $\langle F_i, \varphi_i \rangle$, $i = 0 \ldots k$ be the collection of verification problems obtained by applying steps 1 and 2 above. Then φ is true of the Kripke Structure produced from F if and only if φ_i is true of the Kripke Structure produced from F_i, for each i, $i = 0 \ldots k$.*

Soundness of this algorithm follows trivially from Lemmas 6 and 7. Admittedly, the proof of this algorithm is trivial given the lemmas. We discuss this algorithm only as an illustrative example. Our experience shows that once the reductions are verified, it is relatively trivial to verify a compositional algorithm based on the reductions.

5.3 ACL2 Verification

We now discuss the issues in verifying the soundness of the compositional model checking algorithm in the ACL2 theorem prover. The standard approach in verifying the algorithm is to specify the algorithm as a formal theory in the theorem prover and verify Theorem 2. Notice however, that a formalization of Theorem 2 requires a specification of the semantics of LTL as a theory in ACL2. However, as we mentioned in Section 5.1, the standard semantics of LTL are specified with respect to infinite paths through a Kripke Structure. If paths are represented in ACL2 as `lists` of states, as is customary, then infinite paths are not objects that can be specified in the ACl2 logic. Indeed in ACL2, it is simple to prove that all lists are finite. There are ways to circumvent this restriction of the ACL2 logic, but we could not find any approach that was satisfactory for our purpose. Our workaround to this restriction is based on the observation that there is an infinite path through a Kripke structure satisfying φ if and only if there is an eventually periodic path satisfying φ. This observation is a standard theorem about the classical semantics of LTL. An *eventually periodic path* is one that can be decomposed into a finite path (called "prefix") followed by a finite cycle (that is repeated forever). Since both the prefix and the cycle are finite, they are representable inside the ACL2 logic. We can therefore, specify in a relatively natural way what it means for an eventually periodic

[16][14] presents a more general version of Lemma 9. The original theorem is stated in terms of formulas written in CTL* logic rather than LTL. However, the version we provided in this paper follows immediately from the theorem in [14].

path π to satisfy an LTL formula φ. We then stipulate that a Kripke Structure M satisfies an LTL formula φ if and only if every *eventually periodic path* through M satisfies φ.

While our specification of LTL with respect to eventually periodic paths enables us to define the semantics of LTL in a relatively natural manner in ACL2, it dramatically complicates the soundness proofs and bisimulation arguments. In particular, the arguments for proving Lemmas 8 and 9, restated in terms of eventually periodic paths, are far more complicated than the corresponding classical proofs. However, to the extent that we could needed eventually periodic paths to specify the semantics of LTL in ACL2, the complications are inherent in the proofs of soundness of the algorithms and need to be dealt with in such a verification effort.

We now discuss the ACL2 theorem of soundness of our compositional model checking algorithm in the light of our comments regarding characterizing theorems in Section 4. The precise ACL2 theorem we proved is the following:

```
(DEFTHM compositional-reduction-is-sound
   (implies (syntaxp (quotep C))
            (implies (and (finite-state-model-p C)
                          (ltl-formulap f)
                          (subset (variables-of f) (variables C)))
                     (equal (ltl-semantics-for-models C f)
                            (ltl-semantics-for-models* (compositional-reduction C f))))))
```

If we ignore the first hypothesis (beginning with `syntaxp`), then the theorem simply states that if `C` is a finite state system model written in ACL2, and `f` is an LTL formula, then checking if the Kripke Structure built from `C` satisfies `f` is equivalent to apply the compositional reduction algorithm and then checking each reduced verification problem. Further, since the theorem is used by the theorem prover as a **`rewrite rule`**, ACL2 will attempt to use this rule to check LTL properties of finite system models. (The first hypothesis actually evaluates to `true` in the ACL2 logic but provides a directive to the theorem prover to only consider applying this rule when provided with a concrete instance of a finite state model.) Hence, this theorem acts as a characterization theorem of our compositional model checking algorithm, both specifying what we can infer from a successful check of the algorithm, and directing the theorem prover to apply the algorithm under appropriate conditions.

6 GSTE

In this section, we discuss another automatic verification procedure, namely *Generalized Symbolic Trajectory Evaluation* (GSTE) [55, 56]. Fundamentally, GSTE is based on *Symbolic Trajectory Evaluation* (STE) [10], a lattice-based technique for analysis of properties of finite state system, based on symbolic simulation. STE has been found useful in the verification community, and has been in use in companies like Intel, IBM, and Motorola for efficient verification of hardware designs. In spite of its efficiency, STE is limited in the class of properties that it can specify and verify. GSTE extends STE to handle all ω-regular properties. In this section, we provide an overview of our experiences with verification of a simple GSTE procedure in ACL2. A more detailed account of this work can be found in [47].

The semantics of GSTE is specified in terms of a form of automaton, called assertion graphs, and notions of *satisfaction*. In Section 6.1, we study assertion graphs and different types of satisfaction criteria. In Section 6.2, we discuss a simple GSTE implementation that uses a specific satisfaction criterion, called *strong satisfiability*. Finally, in Section 6.3, we discuss our verification efforts.

6.1 GSTE Specification

GSTE is a linear time verification technique that checks that the set of all executions of the system obey their specification. The specification of GSTE is given by *assertion graphs*, which are a variety of automaton. The assertion graph specifies the set of "correct executions". The GSTE algorithm is essentially a *language containment* algorithm, checking that the set of executions of the system is a subset of the set of executions accepted by the assertion graph.

In general, an assertion graph is a directed graph with a distinguished initial vertex v_0, and the restriction that all vertices must have non-zero out-degree. Each edge e is labeled with an *antecedent* $\texttt{ant}(e)$ and a *consequent* $\texttt{cons}(e)$. The antecedents and consequents are propositional formulas over some set of atomic propositions AP. (Recall from Section 5 that the atomic propositions correspond to the state variables of the system, and hence the antecedents and consequents are simply boolean expressions over state variables.) The assertion graph also has *acceptance conditions* as specified below.

A *path*[17] in the assertion graph is a directed sequence of adjacent edges starting from the initial vertex v_0. Every path in the assertion graph specifies a temporal if-then assertion: if the antecedents hold then the consequents hold as well. Roughly, a path of length n (with n edges) specifies the system's behavior for executions of length n. If all the antecedents hold at the corresponding points of the execution, then in order for the assertion to be satisfied, the corresponding consequents must hold as well. If any of the antecedents does not hold, then the assertion is vacuously `true`. Formally, if ρ is a path of length n, with $\rho[i]$ denoting the i-th edge of ρ, $1 \leq i \leq n$, and if σ is an execution of the system of length n, with $\sigma[i]$ denoting the i-th state in the execution,$1 \leq i \leq n$, then σ *satisfies* or *is accepted by* ρ if and only if the following condition holds:

$$(\forall i : 1 \leq i \leq n : \sigma_i \models \texttt{ant}(\rho[i])) \Rightarrow (\forall i : 1 \leq i \leq n : \sigma_i \models \texttt{cons}(\rho[i]))$$

An assertion graph as a whole accepts a given execution, if and only if all "appropriate" paths in the assertion graph are satisfied. The notion of "appropriate paths" is defined differently for the four different kinds of acceptance in GSTE.

- In *strong satisfiability*, a *finite* execution is accepted if and only if it satisfies all the paths of the same length in the assertion graph.
- In *terminal satisfiability*, some edges in the assertion graph are marked as *terminal edges*, and a *terminal path* in the graph is a path that starts from v_0 and ends with a terminal edge. A *finite* execution is accepted if and only if it satisfies all terminal paths of the same length.
- In *normal satisfiability*, an *infinite* execution is accepted if and only if it satisfies all infinite paths.
- In *fair satisfiability*, there is a finite set of *fair edges.* A path is fair, if and only if it visits every fair edge infinitely often. An *infinite* execution is accepted if and only if it is satisfies all fair paths.

An assertion graph G defines the set of executions it accepts. Call that the language of G, denoted $L(G)$. A finite state system F defines the set of executions that it can produce, denoted $L(F)$. A system F satisfies an assertion graph G if and only if $L(F) \subseteq L(G)$.

6.2 GSTE Verification

In this section, we discuss a simple implementation of GSTE. The goal of verification of the GSTE algorithm is to show the following statement as theorem: *If the GSTE algorithm returns* `true` *on a model F and an assertion graph G, then $L(F) \subseteq L(G)$.* The notion of acceptance we use for this work is *strong satisfiability.*

[17] Recall that in Section 5 we referred to "paths" through a Kripke Structure. Since in this section, we will need to talk about two kinds of paths, namely sequences of edges in an assertion graph and sequence of states in a model, we make a distinction in the terminology. Whenever we refer to "paths" in the context of GSTE, we mean a sequence of edges in the assertion graph. We will consistently refer to a sequence of states in a model as an "execution" in this context.

A GSTE algorithm is *sound*, if and only if, whenever it returns `true`, the model from the circuit strongly satisfies the assertion graph. On the other hand, it is *complete*, if and only if it returns `true` whenever the model strongly satisfies the assertion graph. The GSTE implementation we examine is not complete, and we will only focus on verifying the soundness of the implementation.

The efficiency of a GSTE implementation arises from efficient representation of *sets of states* in terms of lattices [5]. Recall that a state is a boolean assignment to the variables of the system. The approach for the efficient representation is to map every set of states to a "point". A *point* is an assignment of one of the four values $\{\texttt{true}, \texttt{false}, \texttt{X}, \texttt{top}\}$ to the variables. The assignment is based on the function `map` following two conditions:

1. The empty set is mapped to the point in which every variable is assigned to `top`.

2. For a given set S of states, if a variable v is assigned to `true` (resp., `false`) in all the states of S, then v is assigned to `true` (resp., `false`), otherwise v is assigned to `X`.

The set of all possible ternary assignments of the variables of the system, together with the specific assignment of `top` to all the variables forms the set `points`. We define the partial ordering $\prec$ on the four values `true`, `false`, `X`, and `top` as follows: $\texttt{top} \prec \texttt{true}, \texttt{false} \prec \texttt{X}$. We also assume a total ordering relation $<<$ on all the variables. Based on $\prec$ and $<<$, we can now derive a partial order $\sqsubseteq$ for `points` as follows. To determine if for two points p and q, $p \sqsubseteq q$, first order the variables of both points according to the total order $<<$. Then $p \sqsubseteq q$ if and only if for every variable v, the assignment of v in p is either "less than" (according to $\prec$) or equal to the assignment of v in q.

It is clear that `points`, together with the partial order $\sqsubseteq$, forms a lattice with the assignment of `top` to all variables as the bottom point and the assignment of `X` to all variables as the top point. Hereafter, we will use the special symbol $\perp$ to refer to the bottom point of the lattice. Given a model F, we will call the pair $L = \langle \texttt{points}, \sqsubseteq \rangle$, the lattice model for F. We note that for a system with n variables, the number of sets of states is 2^{2^n}, while the number of corresponding lattice points is only $3^n + 1$.

To facilitate reasoning with lattices, it is convenient to extend the notion of *executions* of the model by defining trajectories. Given a set S of states and a natural number n, a trajectory of length n is the set of all executions of length n starting from some state in S. Further, we extend the notion of assertion graphs and satisfiability to deal with lattices as follows:

- Given a lattice $L = \langle \texttt{points}, \sqsubseteq \rangle$, an *abstract assertion graph* is given by $G = \langle V, v_I, E, ant, cons \rangle$ where $ant : E \to L$ maps each edge to a lattice point, and so does $cons : E \to L$.

- A trajectory τ in L *path-satisfies* a path ρ in an abstract assertion graph of the same length under some edge labeling function $\gamma : E \to \texttt{points}$, denoted by $\tau \models_\gamma^L \rho$, if for every $1 \leq i \leq len(\tau)$, $\tau[i] \sqsubseteq \gamma(\rho[i])$. The trajectory satisfies the path, denoted by $\tau \models^L \rho$ if $\tau \models_{ant}^L \rho \Rightarrow \tau \models_{cons}^L \rho$. Finally, a lattice model L *strongly satisfies* an abstract assertion graph G on a lattice $\langle L, \sqsubseteq \rangle$, denoted $L \models^L G$ if and only if for every finite initial path ρ in G and every finite trajectory τ of the same length in L, $\tau \models^L \rho$.

To reconcile the definition of satisfiability for lattice models with the strong satisfiability we discussed in Section 6.1, we note that assertion graphs have their antecedents and consequents labeled with abstract propositions. However, the abstract propositions can be characterized by the set of states that satisfy the propositions, and hence can be mapped to points in the corresponding lattice model. However, the introduction of abstraction weakens the notion of satisfiability in the following sense. Consider an edge e in the assertion graph, and assume that the set of states characterized by the assertions in the $\texttt{cons}(e)$ is S. In the corresponding abstract assertion graph, $cons(e)$ will be a point p in L, obtained by the mapping of S as described above. However, notice that p characterizes a set of states S' such that $S \subseteq S'$. It is possible for S' to be the set of all states in F. However, in such cases, there is no connection between $\texttt{cons}(e)$ in G and its abstraction $cons(e)$ in G'.

To reconcile such discrepancies and enable us to use the lattice model for implementing GSTE which enables us to verify properties of the original assertion graph, we use the notion of "consequent preserving abstractions". An abstract assertion graph G' is *consequent preserving* with respect to a concrete assertion graph G if and only if the following two conditions hold.

- The graphs G and G' have the same vertex, edge relation.
- For every edge e in G, the consequents of e in G refers to the same set of states as the consequents of e in G'.

The following lemmas then allows us to relate the abstract assertion graphs for lattice models with their concrete counterparts.

Lemma 10 *Let M be a Kripke Structure and G be an assertion graph. Consider the lattice model L for M, and let G' be a consequent preserving abstract assertion graph with respect to G with respect to L. Then M strongly satisfies G only if L strongly satisfies G'.*

Lemma 10 allows us to derive soundness theorems for concrete assertion graphs based on corresponding results for abstract assertion graphs. We now discuss the algorithm to determine strong satisfiability of abstract assertion graphs.

The GSTE algorithm works by computing a "simulation sequence" ψ_1, ψ_2,...., where $\psi_i : E \rightarrow L$, for $i \geq 1$. The computation of ψ_i is governed by the following recurrence. Note that the function ψ is computed in a mutually recursive manner with the function $\phi : E \rightarrow L$.

- $\psi_1(e) = ant(e)$ if e starts from the initial node, and $\perp$ otherwise. $\phi_1(e) = \perp$ for all e.
- Assume that we have computed all ψ and ϕ up to some iteration i. Then we show how to compute ψ_{i+1} and ϕ_{i+1}.
  - ψ_{i+1} is the least upper bound of ψ_i and ϕ_i for each e.
  - $\phi_{i+1}(e)$ is computed by taking the greatest lower bound of the `post` of $\psi_i(e')$ with $ant(e')$, and taking the least upper bound of this quantity over all neighbors e' of e.

Given this recurrence, we now define ψ^* as the least fix-point of ψ_1, ψ_2,...... Notice that from the properties of least upper bound, we know that $\psi_{i+1} \sqsubseteq \psi_i$. Also from the properties of assertion graphs we know that each ψ_i is a member of the lattice. This implies that the computation of ψ^* terminates. Given ψ^*, the GSTE algorithm checks if for every edge in G' whether the $\psi^* \sqsubseteq cons(e)$. The following lemma guarantees that the procedure guarantees strong satisfiability.

Lemma 11 *If the GSTE algorithm returns* `true` *on an abstract assertion graph G' and an abstract lattice model L, then L strongly satisfies G'.*

Lemmas 10 and 11 together guarantee that the GSTE algorithm is sound.

6.3 ACL2 Proof of GSTE

In this section, we make some brief comments on our effort verifying the GSTE algorithm in ACL2. Roughly, the proof goes by defining lattices, and exhibiting mappings from sets of states to lattices, and proving Lemmas 10 and 11. In fact, the presentation in Section 6.2 follows the progress of our ACL2 proof. Our experience suggests that verification of decision procedures is feasible in ACL2 as long the ACL2 user is clear about the basic points of the reasoning process.

We show the main theorem proved with ACL2. The statement should be clear, but if not, it merely says that if F is a finite state system and G is an assertion graph, and if the corresponding abstract assertion graph is consequent preserving, then, to check if the Kripke Structure for F strongly satisfies G, it is sufficient to check if the GSTE algorithm returns `true`.

```
(defthm gste-is-sound
  (implies (and (finite-state-systemp C)
                (assertion-graph-p g (create-kripke C))
                (lattice-model-p n (variables C))
                (consequent-preserving-p g (create-latice-assertion-graph
                                              g (variables c))
                                             (edges g)
                                             (variables c))
                (gste n (create-lattice-assertion-graph g (variables c)
                                                          (edges g)
                                                          (variables c))))
           (strongly-satisfiable (create-model c) g)))
```

We again note, that analogous to the compositional model checking verification work described in Section 5, that this approach allows the ACL2 theorem prover to use the GSTE algorithm to verify GSTE properties of finite state systems. There are technical differences between the way ACL2 can use our compositional reduction theorem, and the theorem stated above, which have to be considered in the context of the workings of ACL2 for matching "free variables". We do not go into the technical discussion here; the interested reader will find comments on this aspect in [47]. However, we note that it is still possible to use this theorem and verify properties of real finite state systems in ACL2, specified as assertion graphs.

7 External Oracles with a Theorem Prover

In this section, we address efficiency issues with the process of integration of decision procedures with a theorem prover. Recall that in Section 4, we argued that to apply decision procedures for automatic verification in a sound manner in conjunction with proofs done by a theorem prover, we need to insure that the logic implemented by the decision procedure is consistent with the logic of the theorem prover. Our approach to satisfy this insurance has been to implement the decision procedures as formal theories inside the theorem prover and justifying their semantics using characterizing theorems. This approach enabled us to use automatic verification procedures in combination with the theorem prover, without involving any extra-logical arguments. While such extra-logical arguments have been used to justify, for example, that our formal theory for compositional model checking do correspond to the traditional semantics of the decision procedure, such arguments have never been used to actually solve a model checking problem. In particular, from the perspective of the theorem prover, a compositional model checking algorithm will be invoked only if the goal is to verify that an LTL formula is *valid* for a system model, where the semantics of validity of the formula for system models is strictly in terms of eventually periodic paths. Hence, the soundness of the combination of decision procedures with the theorem prover is established strictly by the characterizing theorem, and the theorem justifies the use of the decision procedure to solve a verification problem.

A practical objection with this approach is efficiency. In particular, the practical utility of most of the decision procedures arises out of very efficient implementations. However, the implementations of decision procedures we considered in Sections 5 and 6 are naive, and targeted to make reasoning about the procedures easier rather than benefit application of the procedures in practice. In particular, the function representing the formal theory of semantics of LTL in section 5 has been implemented with quantification, and cannot be executed by the logic of ACL2. It is possible, now, to define more efficient implementations, show using the theorem prover that such implementations are logically equivalent to the inefficient implementations we verified, and then use the efficient implementations to verify properties of practical systems. However, we consider it unlikely that such implementations will be competitive in efficiency to the implementations already available in the market. Further, such an approach will be tantamount to "reinventing the wheel", since we then need to embark on an efficient implementation in ACL2 for every decision procedure that we need to use to analyze system models. From a practitioner's point of view, this is often unacceptable.

To solve this problem, we consider the possibility of integrating external trusted analysis tools with the theorem prover. It is important at this point to elaborate what we mean by "trusted". We note we can already bestow a trust on a decision procedure if we can implement it as a formal theory in the theorem prover, and then prove the characterization theorem. When a user now claims that an external tool is trusted, it means in our framework that (s)he believes that the procedure satisfies the characterization theorem of the formal theory. Notice however, since the external tools are implemented on a variety of languages and on a variety of software platforms, actual verification of the tool will involve formalization of both the languages and the software platforms on which they are implemented. Such a verification effort is normally impractical. However, nevertheless, it is often the case that even without such verification, for many tools available in the market, the users are willing to trust the analysis provided by the tool. Under such circumstances, we propose to implement ways of using such tools in conjunction with the theorem prover.

We will note that if a theorem prover is used with an external (unverified) analysis tool, then the theorem prover cannot take individual responsibility for the theorems proved by the composite tool. The soundness of such a theorem is based on (1) soundness of the theorem prover, and (2) soundness of the axiom positing that the external tool satisfies the characterization theorem. Hence integration of an external analysis tool with a theorem prover produces a composite analysis technique, which is sound only under soundness of the extra axiom. If the user has sufficient trust in this extra axiom then (s)he can therefore use the composite tool to verify large system models.

Practical considerations have played a part in verification tools. Design of efficient theorem provers like ACL2, and other automatic verification tools like model checkers has been driven primarily by the need for efficient techniques to verify large systems. Integration of the different techniques is necessary if one were to verify large systems, different parts of which can be analyzed efficiently by different analysis tools. As a case in the point, Moore [41] provides a "grand challenge" in formal methods, inviting researchers in formal verification to verify a complete stack from transistors to high-level program semantics. A soundness theorem satisfying the grand challenge can be stated in an expressive logic like ACL2. However, portions of the "verification stack" like analysis of transistors can be performed more efficiently by automatic tools designed for that purpose. Our approach simply provides a way of integrating the analysis tools with the theorem prover, with axioms characterizing precisely what is assumed by the theorem prover when an external tool successfully analyzes parts of the stack.

We further note that, in response to practical concerns, many modern theorem provers already introduce the concept of external oracles. The Isabelle theorem prover provides the concept of external checkers and allows the possibility of attaching external oracles to verify certain theorems [46, § 6]. For example, Isabelle 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 the following tools:

- An efficient μ-calculus model checker, as part of a logic for I/O-Automata [43].
- The Stanford Validity Checker (SVC), as an arithmetic decision procedure for the real-time interval logic Duration Calculus (Isabelle/DC) [22].
- the MONA model checker, as an oracle for deciding formulas expressed in the weak second-order monadic logic of one successor (WS1S) [3].

External oracles are also used in the HOL family of higher order logic theorem provers. Gunter [20] 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 [15] used 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 [26, 18].

In this section, we discuss our approaches with integrating a model checker with ACL2. Technical details of our work can be found in [48]. In Section 7.1, we discuss our current approaches to such integration. In Section 7.2, we discuss why our approach is not fully satisfactory, and propose extensions to the theorem prover to allow easier integration of external tools.

7.1 Integrating a Model Checker with ACL2

We have worked on integrating Cadence SMV [38] with ACL2. Technical details of our work are available in [48]. SMV is an LTL model checker widely used in the industry today. We, therefore, decided to use the model checker to efficiently decide problems like whether a finite state machine satisfies an LTL formula. We note that the characterization theorem of the model checker is described by the ACL2 definition of the semantics of LTL as we discussed in Section 5.3.

ACL2 does not currently support the use of an external oracle in conjunction with its theorem proving engine. Hence, our current integration is best described as an "implementation hack" to introduce the notion of oracles in the theorem prover. In particular, we introduce an executable function in the ACL2 logic, and direct the theorem prover to execute a corresponding definition in Common Lisp. To the ACL2 logic, we posit an axiom stating that the new function is logically equivalent to the semantics of LTL. In the underlying Lisp, the calls to the function are replaced by calls to the external model checker.

Our implementation has been used to check small exercises in [38]. We will note that the performance is worse than the performance of model checking the same problems on SMV directly, but is within acceptable limits. However, note that we only use the "model checking engine" of SMV, which, we contend, has been used often enough to be trusted, just as ACL2 is trusted. On the other hand, the compositional reductions are handled by executable functions in ACL2. In this manner, we can use the assurance of verified compositional procedures with the efficiency of a trusted model checker, with full knowledge of the assumptions required to believe in the soundness of the composite theorem.

7.2 Additional Remarks

As we noted above, our integration is best described as an implementation hack. In particular, notice that we needed to define an executable function in the ACL2 logic, so that we could direct the theorem prover to execute its Common Lisp implementation. Without going through the process of defining an executable model checker in ACL2, (which would not be used in any practical verification since we override the definition in the underlying Lisp), any (total) executable function we produce will be, by definition, logically different from the semantics of LTL. Hence the introduction of the axiom is unsound. In [48], we have introduced a way of using the axiom, such that the unsoundness is not manifest. In particular, the theorem prover has been asked not to use the logical definition of the executable function. However, it is possible for a user to override such directive, and hence prove "theorems" with the composite system, which are not `true`. At any rate, our approach of overriding definitions from the underlying Lisp is repugnant at best. We (and others) are considering approaches for extending the theorem prover, so that it is possible to integrate external oracles naturally with ACL2.

8 Proposed Work

We propose to enhance our current work by exploring the following:

- Our experiences with fair well-founded refinements has shown that it is an effective reasoning framework in practice. On the other hand, the systems we have verified so far have been written at the so-called "protocol level" or "algorithm level". Industrial systems models are implemented at the level of RTL, and are much more elaborate. We plan to use our techniques on RTL modules of processor designs of industrial interest. We plan to use our tool on models of systems written at that level. We have been asked to do so by several researchers. One advantage of applying our tool on such models is that ACL2 already has a collection of libraries providing rewrite rules involving operators in RTL. We are in the process of verifying a pipelined transaction system in this framework, and are using our invariant prover to strengthen the invariants for such verification. We plan to investigate the issues arising out of such RTL level verification of pipelined modules.

- Previous research in verification of pipelined systems involve verifying a Birch and Dill condition [50]. However, Manolios [36, 35] shows that variants of Birch and Dill conditions are flawed, and propose the use of stuttering bisimulation for verification of pipelined machines. We are currently working on an approach verifying pipelined processor models, using stuttering refinement. Our framework closely relates the stuttering bisimulation approach proposed by Manolios. However, we are able to derive the result by simulating Birch and Dill ideas in our mapping for the `rep` functions we discussed in Section 2. We plan to investigate the connection more thoroughly for practical pipeline implementations.

- The compositional model checking algorithm we discussed here is extremely simple, and not useful for reducing large-scale practical verification problems. We are in the process of verifying a more elaborate compositional algorithm based on symmetry and assume-guarantee reasoning. We will explore the issues involved in such verification, particularly in view of the obstacles in formalizing infinite paths in ACL2.

- We will write a paper on our work verifying GSTE in ACL2. To our knowledge, this is the first mechanized proof of a GSTE implementation. Although our proof is based on *strong satisfiability*, we believe the proof highlights the issues involved mechanically reasoning about trajectories in a formal framework. We are looking at other correctness criterion as well, and we believe that *terminal satisfiability* can be verified in a similar manner. A greater challenge is to verify *fair satisfiability*, which involves reasoning about infinite sequences. As we noted in Section 5, formalizing infinite sequences in ACL2 can be tricky, and naive modeling approaches normally do not work. To apply such a notion we need to either have a criterion that reduces checking infinite paths to finite ones, or provide a way of stating properties about infinite sequences in terms of "single steps", akin to stuttering refinements researches described in Section 2.

- Finally, we will look into the possibility of better integration between ACL2 and other external tools. As we noted in Section 7, our current approach is an intermediate hack at best. We and others are currently working on a simpler and more natural integration with external oracles, and contemplating recommendations for the ACL2 implementators for facilitating the possibility.

9 Summary and Conclusions

We are pursuing an approach to combining theorem proving with automatic verification procedures in the verification of large-scale system models. Our goal is to design sound and practicable techniques that can verify large classes of industrial designs. We specifically focus on synchronization protocols, cache coherence, and pipelined architectures. Implementations of such features on commercial systems are often non-trivial and error-prone. Further, automatic verification of temporal properties is normally infeasible in this context because of the state explosion problem. Our general methodology is to use a theorem prover to verify correspondence between executions of the concrete system model and an "abstract model". If the abstract model is simple enough, correctness of the concrete system then follows by simple inspection of the abstract model. Further, in many cases, the abstract model can be expressed in a decidable logic with a reasonable number of states, and automatic decision procedures like model checking can be invoked to verify the properties of the system. The methodology, therefore, provides a nice combination of deductive and automatic verification techniques for effective analysis of large-scale systems. However, as we pointed out in Section 1, the efficacy of the methodology critically depends upon (1) the design of tools and techniques to help reason about the correspondence between executions of a concrete system model and its abstraction with reasonably minimal user interactions, and (2) sound and efficient integration of automatic verification procedures with the theorem prover.

We have addressed the two issues and discussed our solutions. To address issue 1, we developed the framework of fair well-founded refinements in Section 2. Well-founded refinements enable us to relate infinite executions of the two models by proving theorems about single steps. Further, the integration of fairness constraints with the refinement framework often allows for simpler abstract models. We have tested the viability of the framework in the context of ACL2 theorem proving by verifying small but illustrative models of multiprocessor systems. Our initial verification efforts have shown that in a typical proof showing well-founded refinements, the manual effort is expended mostly in the construction and strengthening of invariants. To address this concern, we discussed an invariant proving procedure in Section 3, that provides automatic support in verifying and strengthening invariants. Our tool makes effective use of term-rewriting to "simplify" the original system model to a "finite model" which is then checked for invariants using automatic methods.

To address issue 2, we considered the soundness and efficiency problems arising from integration of decision procedures with the theorem prover. In Section 4 we discussed the soundness issues arising from the possible incompatibility between the logic implemented by the decision procedures used to verify abstract models and the logic of the theorem prover used in verifying the correspondence between the abstract and concrete models. We argued that the soundness of the composite system can be guaranteed up to the soundness of the theorem prover by producing effective embeddings of the semantics of the decision procedues as formal theories inside the theorem prover and proving characterization lemmas that formally specify the semantics of the decision procedures inside the theorem proving framework. We further demonstrated the viability of this approach in Sections 5 and 6, by specifying the semantics of two decision procedures, namely a compositional model checker and a procedure for GSTE, in terms of characterization lemmas inside the ACL2 theorem prover. In Section 7, we addressed the efficiency concerns by exploring the possibility of integrating efficient state-of-the-art implementations of decision procedures available in the market with a theorem proving system. In this case, the soundness of the composite system is guaranteed by (1) the soundness of the theorem prover, and (2) compatibility of the external procedure with the semantics specified by the characterization lemma for the procedure in the theorem prover. We have provided a mechanism that allows us to integrate an external model checker, namely Cadence SMV, with ACL2. We are working on extending the theorem prover in order to make the integration simpler and more natural with ACL2.

As we mentioned in Section 1, our initial experiments indicate that our approach is effective in the verification of practical multiprocessor systems. However, our current multiprocessor models are defined at the protocol level, with data structures like "sets" and "records". Our goal is to provide an effective framework for reasoning about realistic multiprocessor models at the level of RTL. Systems at that level are considerably more detailed and complicated than our current models. We are currently working on building and extending libraries of theorems in ACL2, that can help us effectively reason about such models. We propose to demonstrate our approach by verifying RTL modules of realistic models of pipelined processors and memory systems.

References

[1] M. D. Aagard, R. B. Jones, R. Kaivola, K. R. Kohatsu, and C. H. Seger. Formal verification of iterative algorithms in microprocessors. In *ACM/IEEE Design Automation Conference (DAC)*, 2000.

[2] F. Baader and T. Nipkow. *Term Rewriting and All That.* Cambridge University Press, 1998.

[3] 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.

[4] A. Beckmann, S. R. Buss, and C. Pollett. Ordinal Notations and Well-orderings in Bounded Arithmetic. *Annals of Pure and Applied Logic*, pages 197–203, 2003.

[5] G. Birkhoff. *Lattice Theory*. American Mathematical Society, 1940.

[6] N. Bjorner, A. Borwne, E. Chang, M. Colon, A. Kapur, Z. Manna, H. B. Sipma, and T. E. Uribe. STeP: Deductive-Algorithmic Verification of Reactive and Real-time Systems. In *Computer Aided Verification*, volume 1102 of *LNCS*, pages 415–418. Springer-Verlag, July 1996.

[7] R. S. Boyer, M. Kaufmann, and J S. Moore. The Boyer-Moore Theorem Prover and Its Interactive Enhancements. *Computers and Mathematics with Applications*, 29(2):27–62, 1995.

[8] R. S. Boyer and J S. Moore. Integrating Decision Procedures into Heuristic Theorem Provers: A Case Study of Linear Arithmetic. In *Machine Intelligence 11*, pages 83–124. Oxford University Press, 1998.

[9] B. Brock, M. Kaufmann, and J S. Moore. ACL2 Theorems about Commercial Microprocessors. In M. Srivas and A. Camilleri, editors, *Proceedings of Formal Methods in Computer-Aided Design (FM-CAD)*, pages 275–293. Springer-Verlag, 1996.

[10] Ching-Tsun Chow. *The Mathematical Foundation of Symbolic Trajectory Evaluation*. Springer-Verlag, 1999.

[11] E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic Verification of Finite State Concurrent Systems Using Temporal Logic. *ACM Transactions on Programming Languages and Systems (ACM TOPLAS)*, 8(2):244–263, April 1986.

[12] E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided Abstraction Refinement. In *Computer-Aided Verification (CAV)*, pages 154–169, 2000.

[13] E. M. Clarke, O. Grumberg, and D. E. Long. Model Checking and Abstraction. *ACM Transactions on Programming Languages and Systems (ACM TOPLAS)*, 16(5):1512–1542, September 1994.

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

[15] 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.

[16] E. A. Emerson. *Modal and Temporal Logics*, pages 995–1072. Handbook of Theoretical Computer Science, Volume B: Formal Models and Semantics. North-Holland Pub. Co./MIT Press, 1996.

[17] N. Frances. *Fairness*. Springer-Verlag, 1986.

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

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

[20] E. L. Gunter. Adding External Decision Procedures to HOL90 Securely. *LNCS*, 1479:143–152, 1998.

[21] H. Haidi and N. Shankar. Abstract and model check while you prove. In N. Halbwachs and D. Peled, editors, *Proceedings of International Conference on Computer-Aided Verification (CAV)*, volume 1633 of *LNCS*, pages 443–453. Springer-Verlag, 1999.

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

[23] R. Hosabettu, M. Srivas, and G. Gopalakrishnan. Proof of Correctness of a Processor with Reorder Buffer using the Completion Functions Approach. In N. Halbwachs and D. Peled, editors, *Computer-Aided Verification, CAV*, volume 1633 of *LNCS*, pages 47–59, Trento, Italy, 1999. Springer-Verlag.

[24] Christian Jacobi. Formal verification of complex out-of-order pipelines by combining model-checking and theorem-proving. In *Computer Aided Verification (CAV)*, volume 2404 of *LNCS*, pages 309–323. Springer, 2002.

[25] R. P. Jones. *Symbolic Simulation Methods for Formal Verification.* Kluwer Academic Publishers, June 2002.

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

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

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

[29] M. Kaufmann and J S. Moore. Design Goals of ACL2. Technical Report 101, Computational Logic Incorporated (CLI), 1717 West Sixth Street, Suite 290, Austin, TX 78703, 1994.

[30] M. Kaufmann and J S. Moore. An Industrial Strength Theorem Prover for a Logic Based on Common Lisp. *IEEE Transactions on Software Engineering*, 23(4):203–213, April 1997.

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

[32] M. Kaufmann and R. Sumners. Efficient Rewriting of Data Structures in ACL2. In *Third International Workshop on ACL2 Theorem Prover and Its Applications*, Grenoble, France, April 2002.

[33] L. Lamport. A New Solution of Dijkstra's Concurrent Programming Problem. *Communications of the ACM*, 17(8):453–455, August 1974.

[34] L. Lamport. Proving the Correctness of Multiprocess Programs. *IEEE Transactions on Software Engineering*, 3(2):125–143, March 1977.

[35] P. Manolios. Correctness of pipelined machines. In W. A. Hunt (Jr.) and S. D. Johnson, editors, *Formal Methods in Computer-Aided Design (FMCAD)*, volume 1954 of *LNCS*, pages 161–178. Springer-Verlag, 2000.

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

[37] 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.

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

[39] J. Misra. *A Discipline of Multiprogramming: Programming Theory for Distributed Applications.* Monographs in Computer Science. Springer, 2001.

[40] J S. Moore. Proving Theorems about Java and the JVM with ACL2. Markdoberdorf Summer School 2002 — Lecture Notes.

[41] J S. 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.

[42] J S. Moore and G. Porter. The Apprentice Challenge. *ACM Transactions on Programming Languages and Systems (ACM TOPLAS 2002)*, 24(3):1–24, May 2002.

[43] 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*, volume 1019 of *LNCS*, Aarhus, Denmark, May 1995. Springer-Verlag.

[44] K. Namjoshi. A Simple Characterization of Stuttering Bisimulation. In *17th Conference on Foundations of Software Technology and Theoretical Computer Science*, volume 1346 of *LNCS*, pages 284–296, 1997.

[45] 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.

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

[47] S. Ray. Verification of Generalized Symbolic Trajectory Evaluation in ACL2: A Progress Report. In Preparation.

[48] S. Ray, J. Matthews, and M. Tuttle. Certifying Compositional Model Checking Algorithms in ACL2. In W. A. Hunt (Jr.), M. Kaufmann, and J S. Moore, editors, *Fourth International Workshop on ACL2 Theorem Prover and Its Applications*, Boulder, CO, July 2003.

[49] D. Russinoff. A Mechanically Checked Proof of IEEE Compliance of the Floating Point Multiplication, Division, and Square Root Algorithms of the AMD-K7$^{\mathrm{TM}}$ Processor. *LMS Journal of Computation and Mathematics*, 1:148–200, December 1998.

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

[51] 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.

[52] R. Sumners. Correctness Proof of a BDD Manager in the Context of Satisfiability Checking. In *Second International Workshop on ACL2 Theorem Prover and Its Applications*, Austin, TX, October 2000.

[53] R. Sumners. Fair Environment Assumptions in ACL2. In W. A. Hunt (Jr.), M. Kaufmann, and J S. Moore, editors, *Fourth International Workshop on ACL2 Theorem Prover and Its Applications*, Boulder, CO, July 2003.

[54] The VIS Group. VIS: A system for Verification and Synthesis. In R. Alur and T. Henzinger, editors, *Proceedings of the 8th International Conference on Computer Aided Verification (CAV)*, LNCS, New Brunswick, NJ, July 1996. Springer.

[55] J. Yang and C. H. Seger. Introduction to Generalized Symbolic Trajectory Evaluation. In *Proceedings of the IEEE International Conference On Computer Design: VLSI In Computers & Processors (ICCD 2001)*, IEEE Computer Society, pages 360–367, Austin, TX, September 2001.

[56] J. Yang and C. H. Seger. Generalized Symbolic Trajectory Evaluation: Abstractions in Action. In M. Aagaard and J. W. O'Leary, editors, *Proceedings of the 4th International Conference on Formal Methods in Computer-Aided Design (FMCAD 2002)*, volume 2517 of *LNCS*, pages 70–87. Springer-Verlag, 2002.