Using Theorem Proving with Algorithmic Techniques for Large-scale System Verification

Ph.D. Oral Proposal

Sandip Ray

`sandip@cs.utexas.edu`

Department of Computer Sciences

The University of Texas at Austin

Motivation

- Design of modern computing systems is error-prone.
  - Simulation and testing cannot catch all the bugs.
  - Bugs discovered after manufacture can be extremely expensive.
  - Can we mathematically prove that systems behave correctly?
    - McCarthy's Dream (1962) [2]:
      "Instead of debugging a program, one should prove that it meets its specification, and this proof should be checked by a computer program."

Formal Verification

Formal Verification is a practical approach to realizing McCarthy's dream.

- Model the executions of the system under interest as formal objects in some logic.
- Prove the desired properties as formal theorems about the models in the logic.
    - Use a (trusted) computer program to assist in the proof generation process.

Formal Verification: Approaches

- Deductive Verification (theorem proving)
  - Logic used is expressive but undecidable.
  - A “theorem prover” is responsible for finding and checking proofs.
    - A user “guides” the theorem prover in proof search.
- Algorithmic Verification (Model Checking)
  - The logic used is decidable.
    - Checking properties in the logic is automatic (at least in principle).
- **Our Goal:** Combine the two approaches “effectively” for verifying large systems.

Why Combine Two Techniques?

Neither technique is effective as is for verification of large systems!

- Deductive Verification:
  - Requires substantial interaction from a “knowledgable” user.
  - The proof might change considerably as the design evolves!
- Algorithmic Verification:
  - Involves an intelligent but exhaustive search of the states of the underlying system.
    - For large systems, these techniques suffer from *state explosion*.

Approach to Combination

- Use Theorem proving to verify the correlation between the “concrete system” and an “abstract model”.
- The abstract system should have much fewer states.
  - Apply algorithmic verification techniques to verify such abstract models.
- Use the correlation proof and the algorithmic proof to conclude that the concrete system has the desired properties.

Basic Requirements

- Theorem proving aspect of the work must focus on lessening the manual effort.
    - Automatic (possibly heuristic) tools need to work with the theorem prover to help in verification of correlation.
- Algorithmic techniques should be carefully used so that state explosion can be avoided.
- The integration of the two techniques should be sound and efficient.

Domains of Interest

Our principal focus is on verification of implementations of multiprocessor system models:

- Synchronization protocols.
- Pipelined architectures.
- Cache coherence.

ACL2

ACL2 is the theorem proving system and logic that we use for our work.

- ACL2 is a programming language, logic, and a theorem prover for the logic.
  - It is relatively easy to code up decision procedures and tools in the ACL2 language, and possible to verify them.
- ACL2 has been successfully used for verification of large-scale system models.

We later discuss why we need to integrate “external” tools with ACL2 and how we propose to do it.

Modeling Systems

- Computing systems are traditionally modeled operationally in ACL2.
  - *"The meaning of a program is defined by its effect on the state vector."* (McCarthy, 1962) [2]
- A system model in ACL2 is defined by three functions:
  - A state function `step` that takes a "current state" s and "current input" i and returns the "next state" s'.
  - A predicate `init?` that recognizes if a state is an "initial state".
  - A function `label` that maps a state s to a collection of "observations" at s.

Framework: Well-founded Refinements

Introduced by Sumners [4] and follows from work by Manolios, Namjoshi and Sumners [1] on WEBs.

- Relates executions of two system models `impl` and `spec` at different levels of abstraction.
  - Define a function `rep` that maps a state of `impl` to a state of `spec`.
  - Show that for every "step" of `impl`, `spec` takes a "step" or "stutters".
  - Show that the observations are preserved by `rep`, *i.e.*,, a state of `impl` and the corresponding `spec` state have "equivalent" observations.
  - Use an argument based on well-foundedness to show that "stuttering" is finite.

Examining Well-founded Refinements

We used the framework to verify several distributed protocols, including a (simplified) Bakery Algorithm.

- Joint work with Rob Sumners.
- Our Observations:
  - We needed to extend the framework to incorporate the notion of “fair executions”.
    - The “fairness constraints” have been subsequently extended and improved by Sumners [5].
  - Most of the human effort is expended in the process of defining and proving “invariants”.
    - Similar conclusion has been reached by others (independently) in verifying computing models.

Inductive Invariants

Inductive invariants are predicates that are true along every "step" of the system model.

- `(init? s)` $\Rightarrow$ `(inv s)`, and
- `(inv s)` $\Rightarrow$ `(inv (step s i))`

For example, assume that a component of a state is a `counter` that is incremented at each "step" starting from `0`. Then, an inductive invariant is:

- *The value stored in the "*`counter`* variable", is a natural number.*

Invariants and Refinement

In well-founded refinement, we need to show that when `impl` takes a “step”, `spec` either takes a “step” or “stutters”.

$$\texttt{(rep (impl s i))} = \begin{cases} \texttt{(rep s)} \\ \texttt{(spec (rep s) (pick s i))} \end{cases}$$

If `inv` had been shown to be an inductive invariant, then we can assume `(inv s)` for this proof!

- Determine a predicate `inv` such that:
  1. `inv` is an inductive invariant.
  2. Assuming `inv` you can prove the conditions of well-founded refinement above.

An Invariant Prover

We have implemented a tool with the ACL2 system to generate and prove inductive invariants.

- Joint work with Rob Sumners.
- Basic Idea:
  - Start with a predicate `suff` that is strong enough to prove the conditions of well-founded refinement.
  - "Strengthen" `suff` using a refinement procedure to get an inductive invariant `inv`.
    - Procedure uses *term rewriting*, and lightweight *model checking*.
    - If it cannot strengthen `suff` to `inv` it produces a "counterexample".

Invariant Strengthening Example

PC	Impl Program	PC	Spec Program
1	`j=init;`	1	`j=init+3;`
2	`j++;`		
3	`j++;`		
4	`j++;`		

An invariant we might like is: *The value of* `j` *in the* `impl` *system in a state where* `PC` *is* `4` *is equal to* `(init + 2)`.

- But this is not an inductive invariant!

Informally, to prove the property as an invariant for the states where `PC` is `4`, we need to know something about the states for which `PC` is `3`.

Example: Continued

We strengthen the invariant by a simple rewriting technique:

- *(*`PC = 4`*) implies (*`j = init + 2`*)* simplifies to:
- *(*`PC = 3`*) implies (*`j = init + 1`*)*.

The "simplification" is obtained using ACL2's simplification engine along with built-in rewrite rules verified by the theorem prover.

- **Note:** Our tool critically depends on the availability of libraries of rewrite rules to help in the simplification process.

Applications of Invariant Prover

We have modeled a fairly complex multiprocessor memory system with caches and directories.

- We can prove well-founded refinement between the memory system and a simple spec.
- The invariant prover is used to generate invariants for this proof.
- Experience shows that the prover is useful.
- Observations:
  - Invariant prover critically depends on built-in libraries of “good” rewrite rules.
  - The system model we have used is at protocol level.

Decision Procedures

- Decision procedures (like model checking) implement some (decidable) logic.
  - Logic of the theorem prover (ACL2) might not be compatible with the logic of the decision procedure.
    - The semantics of LTL model checking is specified in terms of infinite sequence of states.
    - If sequences are modeled as `lists`, it is easy to prove in ACL2 that all sequences are finite.
- How do we then use decision procedures on abstract models and compose them with the refinement proof relating concrete and abstract models?
- *Note:* Ruben Gamboa faced similar problems trying to verify square root algorithms in ACL2.

Verifying Decision Procedures

- Decision procedures are programs too!
  - You can model them in ACL2, and prove properties about them.
  - We refer to such theorems about decision procedures as *characterizing theorems*.
    - They tell you exactly what can be derived in ACL2 if a decision procedure returns `true` or `false` on some verification problem.
  - The characterizing theorems sometimes turn out to be different from the traditional semantics of the decision procedure, in order to be expressible to the theorem prover.

Verifying Decision Procedures: Feasibility

Is verification of decision procedures to generate characterizing theorems feasible?

- We have verified two decision procedures:
  1. A (simple) compositional model checking procedure.
  2. A (simple) implementation of Generalized Symbolic Traejctory Evaluation, using *strong satisfiability*.
- Observations:
  - The approach is feasible, though non-trivial.
  - Characterizing theorems and their proofs can be very different from traditional ones.
  - But, the proof is “once-off” per decision procedure to be integrated.

Verifying Compositional Model Checking

- Joint work with John Matthews and Mark Tuttle.
- Uses a composition of two (trivial) model checking reductions.
  1. Conjunctive reduction
  2. Cone of Influence reduction
- The model checking logic used is LTL.
- Observations:
  - The reductions are really trivial, but their verification turned out to be complicated.

Notes on Our Proof

The chief road-block was in specifying the semantics of LTL!

- We could not specify the semantics in terms of infinite paths.
- We used *eventually periodic paths*, that is, infinite paths composed of a finite “prefix” followed by a finite “cycle”.
  - **Known Result:** If there is an infinite path violating an LTL property, then there also exists an eventually periodic path violating the property.
- All proofs had to be cast into this framework. Proofs turned out to be different and sometimes complicated.
- Full details in our paper [3].

Verification of GSTE

Joint work with Warren A. Hunt (Jr).

- GSTE is an efficient lattice-based automatic verification technique.
- Properties are specified not as formulas but in terms of *assertion graphs*.
- Several notions of correctness exist in the GSTE literature, namely *strong*, *terminal*, and *fair* satisfiabilities.
  - Fair satisfiability can express any ω-regular property.
- We verified an algorithm that implements *strong satisfiability* which is normally used for verification of safety properties.

Notes on Our Proof

- Our implementation is not terribly efficient, but we anticipate that a more efficient implementation will be verified along the same lines.
  - More efficient implementations have been done in ACL2 itself by Erik Reeber.
- Proof involves mechanically verifying results from lattices and partial order relations.
- To our knowledge, this is the first mechanical verification of GSTE in a general-purpose theorem prover.
  - We are on the way to verify a slightly more sophisticated implementation that satisfies *terminal satisfiability*.

Characterizing Theorems

- Compositional Model Checking:
  - The compositional algorithm decomposes a verification problem to a collection of “smaller” verification problems. (In this context, a verification problem is a pair $\langle M, \phi \rangle$.)
  - **Theorem** (proved by ACL2): The original verification problem returns `true` if and only if every verification problem produced by the compositional algorithm returns `true`.
  - Truth of a verification problem is defined according to model checking semantics for LTL (in terms of eventually periodic paths).

Using Characterizing Theorems

Given specific M and ϕ, we use the theorem to enable us to decompose the verification problem.

- Such verification has actually been done on relatively simple system models.
  - ACL2 has used the characterizing theorem to deduce that it can decompose the problem into a number of simpler problems.
  - Each of the simpler problems could now be verified by a model checker. (Some issues there, we will discuss them later.)
  - Soundness of the combination guaranteed by the soundness of ACL2 and characterizing theorem.

External Tools

Do we need to implement every decision procedure efficiently in ACL2?

- We did NOT implement an efficient model checker, but used an external model checker (Cadence SMV).
- ACL2 does not have a mechanism of hooking an external tool.
  - We had to hack into ACL2 to get the hook.
  - Guarantee from composition: *If the external tool satisfies the characterizing theorem, then the verificaion using the composite structure is correct.*
- Better approaches for integrating external tools with ACL2 are explored by Erik Reeber.

Summary of Proposals

- Use theorem proving to verify correlation between executions of a “concrete system” and an “abstract model”.
  - Abstract system must be “simpler” than the concrete system under consideration.
  - Design automatic tools to lessen manual effort in this task.
- Design a framework to integrate algorithmic procedures to verify properties of the abstract system.
  - The integration should be sound and efficient.
- Use the composite framework to verify models of multiprocessor systems of practical complexity.

Our Accomplishments

- Examined and extended well-founded refinements.
  - Verified several distributed synchronization protocols, including a (simplified) Bakery Algorithm.
  - Built a tool for checking and strengthening invariants in this framework.
- Used the framework and our tool to verify some properties of a complex multiprocessor cache system.
- Explored the feasibility of integrating decision procedures with ACL2, using characterizing theorems.
  - Verified a compositional model checking algorithm and an algorithm for GSTE.
- (Roughly) integrated external procedures (Cadence SMV) with ACL2.

Proposals

1. Verification of RTL level designs.
   - Such designs are much “lower level” than the models we verified.
   - We are looking to building better libraries for reasoning about such systems.
2. Verification of Pipelined systems.
   - We are working on an approach to verify pipelined machines using well-founded refinements.
     - We are using quantified first-order predicates to simplify the definition of invariants.
3. Building and integrating more efficient decision procedures and external tools.

References

[1] 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.

[2] J. McCarthy. Towards a Mathematical Science of Computation. In *Proceedings of the Information Processing Congress*, volume 62, pages 21–28, Munich, West Germany, August 1962. North-Holland.

[3] 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.

[4] 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.

[5] 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.