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 late in design can be extremely expensive.
- Can we mathematically prove that systems behave correctly?

  McCarthy, 1962 "Instead of debugging a program one should prove that it meets its specification, and the proof should be checked by a computer program."

Formal Verification

Basic Approach:

- Model the system of interest as a formal object 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)
  - The logic is expressive but undecidable.
    - A user guides the theorem prover in proof search.
- Algorithmic Verification (Model Checking, Equivalence Checking etc.)
  - The logic is decidable.
    - Checking properties in the logic is automatic (at least in principle).
- **Our Goal:** Combine the two approaches “effectively” for verifying large systems.

Large Systems

- Multiprocessor systems:
  - Synchronization Protocols
  - Cache Coherence Protocols
- “Low-Level” Implementations:
  - Pipelined Architecture
  - RTL level hardware modules

Why Combine Two Techniques?

Neither technique is effective individually for verification of large systems!

- Deductive Verification:
  - Requires substantial interaction from a knowledgeable 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 system”.
  - Apply algorithmic verification techniques when relevant to verify such abstract systems.
- 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.
- Algorithmic techniques should be used carefully so that state explosion can be avoided.
- The integration of the two techniques should be sound and efficient.

ACL2

ACL2 [KMM00b, KMM00a] 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.
  - Easy to code up decision procedures and model systems.
- ACL2 has been successfully used for verification of large-scale system models.
  - Libraries of lemmas about common software/hardware operations available.
- Local expertise.

Current Work

- Examined a refinement framework to verify correlations.
  - Verified several concurrent protocols, including an implementation of the Bakery Algorithm.
  - Built a tool for “automatically” establishing invariants.
  - Currently using the framework and our tool to verify a multiprocessor cache system.
- Explored the integration of decision procedures with ACL2.
  - Verified a compositional model checking algorithm and an algorithm for GSTE.
  - Made prototype connection of Cadence SMV with ACL2.

Proposed Work

1. Verification of RTL level designs.
   - Such designs are much “lower level” than the models we verified.
2. Verification of Pipelined systems.
3. Exploration of better integration strategies between decision procedures and ACL2.

Broad Outline

- Verifying Correlation with Theorem Proving
  - Tools helping Theorem Proving
- Integration with Decision Procedures
  - Soundness Issues
  - Efficiency Issues
- Summary

Verifying Correlation

Theorem proving for verifying correlations involves:

- Modeling the implementation system `impl`.
- Defining the abstract system `spec`.
- Proof rules which guarantee correlation between `impl` and `spec`.

Modeling Systems

- A system model in ACL2 is defined by three things:
  - A state transition function `step` that takes a "current state"s and "current input" i and returns the "next state".
  - An initial state `*init*`.
  - A function `label` that maps a state s to a collection of "observations" at s.
    - Think of the observations as the visible components of the states.

Correlation: Approach

We show correlation using a refinement framework.
An implementation `impl` is a refinement of `spec` if:

- Every state of `impl` has the same observation as the corresponding state of `spec`.
  `label(s) = label(rep(s))`
- When `impl` takes a step, `spec` either takes a corresponding step or "stutters".

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

- Stuttering is finite (established using well-foundedness).

Examining Stuttering Refinements

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

- Salient Features:
  - We can relate infinite executions of systems by reasoning about single steps.
  - We can reason about both safety and liveness properties.
  - Fairness constraints can be easily integrated [Sum03].
  - Determining invariants requires considerable user interaction.

This is joint work with Rob Sumners.

Invariants and Stuttering Refinement

Invariants are properties that are true of every reachable state of the implementation.
In stuttering refinement, we need to show that when `impl` takes a “step”, `spec` either takes a “step” or “stutters”.

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

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

Can we make invariant proofs easier in the context of stuttering refinement proofs?

Broad Outline

- Verifying Correlation with Theorem Proving
  - **Tools helping Theorem Proving**
- Integration with Decision Procedures
  - Soundness Issues
  - Efficiency Issues
- Summary

Invariant Proving

Goal: Given a predicate `inv` (which is sufficient to show stuttering refinement), determine whether `inv` is an invariant or not.

- We have implemented a tool in ACL2 that can check if `inv` is an invariant.
  - Tool uses term rewriting and reachability analysis.
  - Facilities provided for counterexample generation and useful feedback to the user when it does not succeed.

This is joint work with Rob Sumners.

Invariant Proving: Trivial Example

1. `step(s,i)`
   $\triangleq$

   ```
   if integer(i)
      list(second(s),i)
    else list(second(s),0)
   ```

2. `*init*` $\triangleq$ `list(0,0)`
3. `inv(s)` $\triangleq$ `integer(first(s))`

We determine that `inv` is an invariant, by reducing this system to a finite abstract system and using reachability analysis on the abstract system.

Example: Continued

The intuition: For a state `s`, to determine that `step(s,i)` satisfies `inv`, we might need to know something more about `s` than that `s` satisfies `inv`.

We determine that by term-rewriting:

```
inv(step(s,i))
→ {rewrites to}
if integer(i)
   integer(second(s))
 else integer(second(s))
```

So we need to know something about `integer(second(s))` as well!

Example: Continued

Do we need to know something more for `integer(second(s))`?

```
integer(second(step(s,i)))
→
if integer(i)
   T
 else T
```

Nothing further required!
We now create an abstract model where a state is a pair of bits and every input can be taken to be a Boolean.

Example: Continued

1. `abs_step(s,i)`
   $\triangleq$

   ```
   list((if i second(s) else second(s)),
        (if i T else T))
   ```
2. `*abs_init*` $\triangleq$ `list(T,T)`

Here we can treat `s` to be composed of a pair of Booleans, and `i` to be a Boolean.

- So we now have a finite model. We now test if the bit corresponding to our invariant (the first component) is always `T`.
  - We can do this automatically by reachability analysis on this model.

Applications of Invariant Prover

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

- We are proving stuttering refinement between this memory system and a simple spec.
  - Work in progress.

The invariant prover is used to establish invariants for this proof.

- Experience shows that the prover is useful.
- General Observation:
  - Invariant prover critically depends on built-in libraries of “good” rewrite rules.

Broad Outline

- Verifying Correlation with Theorem Proving
  - Tools helping Theorem Proving
- **Integration with Decision Procedures**
  - **Soundness Issues**
  - Efficiency Issues
- Summary

Decision Procedures

- Decision procedures (like model checking) verify properties in some (decidable) logic.
  - The logic of the theorem prover (ACL2) might not be compatible with the logic of the decision procedure.

How do we then use decision procedures on abstract models and compose them with the refinement proof relating the concrete and abstract models?

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 us exactly what can be derived in ACL2 if a decision procedure returns `true` or `false` on some verification problem.

Feasibility

Is verification of decision procedures feasible?

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

Compositional Model Checking

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.

This is joint work with John Matthews and Mark Tuttle.

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:** For finite state systems, 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 [RMT03].

GSTE

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*, *normal*, *terminal*, and *fair* satisfiabilities.
- We verified an algorithm that implements *strong satisfiability* (normally used for specification of safety properties).

This is joint work with Warren A. Hunt (Jr).

Notes on Our Proof

- Our implementation is not 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 properties of 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 for *terminal satisfiability*.

Characterizing Theorems

- Compositional Model Checking:
  - The compositional algorithm decomposes a verification problem into a collection of “smaller” verification problems.
  - ACL2 verifies that solving the original problem is equivalent to solving all the smaller problems. (Characterizing Theorem)
  - A Model checker now can be used to solve the smaller problems.

Note: The characterizing theorem is in terms of eventually periodic paths.

Broad Outline

- Verifying Correlation with Theorem Proving
  - Tools helping Theorem Proving
- Integration with Decision Procedures
  - Soundness Issues
  - **Efficiency Issues**
- Summary

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 for naturally integrating an external tool.
  - Some “hacking” into the theorem prover was necessary.
- If the external tool satisfies the characterizing theorem, then the property proved by the integrated tool is valid.

Note: Erik Reeber is working on better integration approaches.

Broad Outline

- Verifying Correlation with Theorem Proving
  - Tools helping Theorem Proving
- Integration with Decision Procedures
  - Soundness Issues
  - Efficiency Issues
- **Summary**

Goals

- 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.

Completed Work

- Examined a refinement framework with provision for stuttering.
  - Verified several distributed synchronization protocols, including a (simplified) Bakery Algorithm.
  - Built a tool for establishing invariants.
- Used the framework and our tool to verify some properties of a multiprocessor cache system.
- Explored the integration of decision procedures with ACL2, using characterizing theorems.
  - Verified a compositional model checking algorithm and an algorithm for GSTE.
- Made prototype connection of Cadence SMV with ACL2.

Continuing Work

1. Verification of RTL level designs.
   - Such designs are much “lower level” than the models we verified.
   - We are working to build better libraries for reasoning about such systems.
2. Verification of Pipelined systems.
   - We are working on an approach to verify pipelined machines using stuttering refinements.
     - We are using quantified first-order predicates to simplify the definition of invariants.
3. (Speculative) Explore possibilities for better integration of external tools with ACL2.

Questions?

References

[KMM00a] M. Kaufmann, P. Manolios, and J S. Moore, editors. *Computer-Aided Reasoning: ACL2 Case Studies*. Kluwer Academic Publishers, June 2000.

[KMM00b] M. Kaufmann, P. Manolios, and J S. Moore. *Computer-Aided Reasoning: An Approach*. Kluwer Academic Publishers, June 2000.

[RMT03] 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.

[Sum03] 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.