Certifying Compositional Model Checking Algorithms in ACL2

Sandip Ray

John Matthews

Mark Tuttle

ACL2 Workshop Presentation

July 14, 2003

Outline

- Motivation and Goals
- Technical Background
- Comments on Our Work
- Issues and Proposals

Model Checking

- A procedure for automatically deducing temporal properties of reactive computer systems.
  - The temporal properties are specified in some *temporal logic* (CTL, LTL etc.)
  - A computer system is specified as a *Kripke Structure*.
  - The properties are verified by intelligent and systematic graph search algorithms.

Model Checking: Good, Bad, & Ugly

- **Good:**
  - *If it works*, model checking (unlike theorem proving) is a push-button tool.
- **Bad:**
  - If the system is too large, model checking cannot be applied because of *state explosion*.
- **Ugly**
  - The system (and/or property) then needs to be suitably “abstracted” in order to use model checking.

Compositional Model Checking

- Replace the original verification problems by one or more “simpler” problems.
  - Exploit characteristics of the system like symmetry, cone of influence etc.
- Solve each simpler problem using model checking.

Can be used to verify considerably larger systems.

Verifying Compositional Algorithms

- Implementations of compositional algorithms are often complicated.
  - How do we insure that the algorithms themselves are sound?
- A plausible solution:
  - Use theorem proving to verify the algorithms.
- End Result:
  - A *verified* tool that can be effectively used to model check temporal properties of *large* systems.

Our Work

- A feasibility test for verifying compositional algorithms in ACL2.
- **Goals:**
  - Implement and verify a simple compositional algorithm based on two simple reductions.
  - Integrate the compositional algorithm with a state-of-the-art model checker (Cadence SMV) for efficiently solving the reduced problems.

Outline

- Motivation and Goals
- **Technical Background**
- Comments on Our Work
- Issues and Proposals

How Do we Verify Compositional Algorithms?

- Specify what it means to verify a temporal property of a system model.
  - Implement the semantics of model checking.
- Implement the compositional algorithms.
  - Recall that a compositional algorithm decomposes a verification problem into a number of "simpler" problems.
- Use theorem proving to show that solving the original problem is equivalent to solving all of the simpler problems (with respect to the semantics of model checking).

System Models

- A System is modeled by:
  - A collection of *state variables*. The *states* of the system are defined as the set of all possible assignments to these variables.
  - A description of how the variables are updated in the next state.
  - A set of *initial states* corresponding to the collection of possible evaluations at reset.

System Model Example

A very simple system:

boolean *v1, v2, v3*;

Repeat forever in parallel

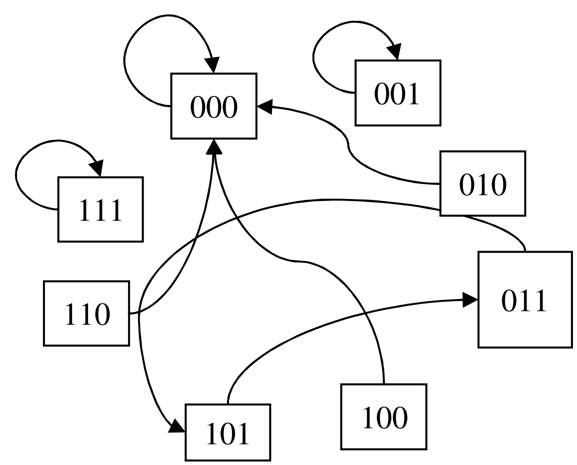
v1 = v2 & v3

v2 = v1 & v3;

end.

Initial states: <000, 111>

Corresponding state representation.



Modeling Temporal Properties

- We use LTL formulas to model properties.
- An LTL formula is either:
  - Some state variable or the constants True, False.
  - A Boolean combination of LTL formulas.
  - The application of a temporal operator G, F, X, U, or W to an LTL formula.
- Example property for the simple system:

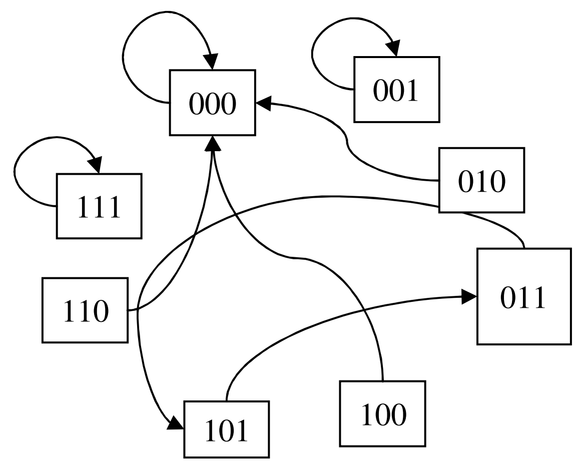
  F (~ v1)

Semantics of LTL

- The semantics of LTL is specified with respect to (infinite) paths through the system model.
  - v is true of some path if *v* is assigned to true in the first state of the path. (True is true of every path.)
  - F stands for *eventually*:
    - (F p) is true of some path iff p is true of some suffix of the path.
  - G stands for *globally*:
    - (G p) is true of some path iff p is true of every suffix of the path.
- A formula is true of a model iff it is true of every path through the model.
- We will call the pair <*f*, *M*> as a *verification problem*, if *f* is an LTL formula and *M* is a system model, and the verification problem is *satisfied* if *f* is true of *M*.

LTL Model Checking Example

- An Example Property:
  - Eventually v1 becomes false.
- Counterexample!!!
  - Path through <111>



Our Simple Model

Compositional Algorithm

- Based on two simple reduction:
  - Conjunctive reduction
  - Cone of Influence Reduction

Conjunctive Reduction

- Replace the verification problem
  - (f1 Λ f2) is true of *M*.
- With the two problems:
  - f1 is true of *M*.
  - f2 is true of *M*.

Cone of Influence Reduction

A Simple System Model

Boolean v1, v2, v3, v4, v5, v6;

Repeat forever in parallel:

```
v1  =  v2;
v2  =  v1 & v3;
v3  =  v1 & v2;
v4  =  v5 & v3;
v5  =  v4 & v6;
```

End.

A Simple LTL property

(F (~ v1)): *v1* will eventually become False.

Cone of Influence Reduction

Boolean v1, v2, v3;

Repeat forever in parallel:

```
v1  =  v2;
v2  =  v1 & v3;
```

End.

Soundness of Reductions

- Conjunctive Reduction
  - The verification problem <(*f1* & *f2*), *M*> is satisfied if and only if <*f1*, *M*> is satisfied and <*f2*, *M*> is satisfied.
- Cone of Influence Reduction
  - If *f* is an LTL formula that refers only to the variables in *V*, and *C* is the cone of influence of *V*, then <*f*, *M*> is satisfied if and only if <*f*, *N*> is satisfied, where *N* is the reduced model with respect to *C*.

Compositional Algorithm

- Input: A verification problem: <*f*, *M*>
- Algorithm:
  - Apply conjunctive reduction to the formula, thus producing a collection of "simpler" verification problems: <*fi*, *M*>
  - Apply cone of influence reduction to each of the simpler problems thus producing problems: <*fi*,*Mi*>
- **Soundness theorem:**
  - If *f* is an LTL formula, and *M* is a model, then <*f*, *M*> is satisfied if and only if each <*fi*, *Mi*> is satisfied.

Note: Soundness of this algorithm follows from the soundness of the reductions.

Outline

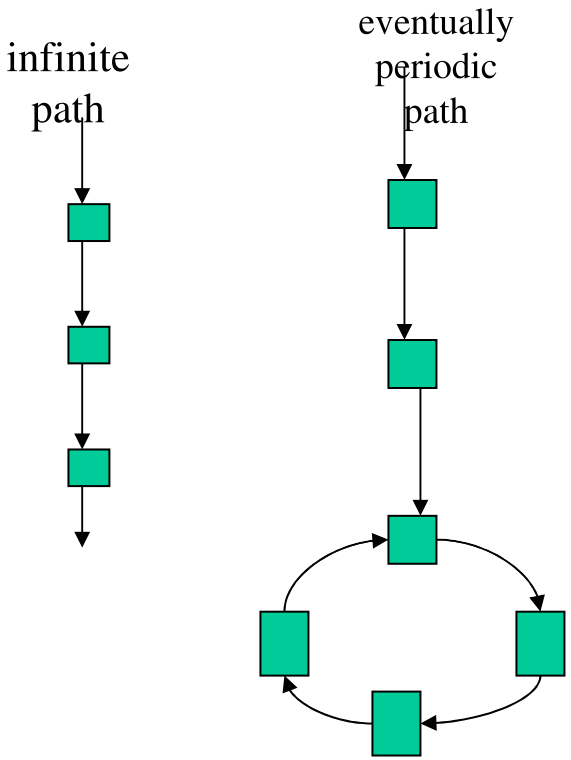
- Motivation and Goals
- Technical Background
- **Comments on Our Work**
- Issues and Proposals

Proving Compositional Algorithms

- The biggest stumbling block is the definition of the semantics of LTL.
  - LTL semantics are classically defined with respect to infinite sequences (paths).
  - The definitional equations require the use of recursion and quantification.
- We could not define the classical semantics of LTL in ACL2.

Eventually Periodic Paths

- These are special infinite paths with a *finite* prefix followed by a *finite* cycle (which is repeated forever).
- Known result:
  - If an LTL *f* property does not hold for some infinite path in some model *M*, there is an eventually periodic path in *M* for which *f* does not hold.



Modeling Semantics of LTL in ACL2

- Eventually periodic paths are finite structures.
  - We can represent them as ACL2 objects.
  - We define the semantics of LTL with respect to such structures.
  - We define the notion of a formula being true of a model by quantifying over all eventually periodic paths consistent with the model.
  - The known result guarantees this is equivalent to the standard semantics.

Issues with the Definition

- We verified our compositional algorithm to be sound using this definition.
- Observations on the proof:
  - The definition is more complicated to work with than the traditional definition.
  - The proofs of the reductions are very different from the standard proofs.
  - Some proofs, for example soundness of cone of influence, get *much* more complicated than the standard proofs.

Note: Details of the complications are in the paper.

Outline

- Motivation and Goals
- Technical Background
- Comments on Our Work
- **Issues and Proposals**

Principal Proposals

1. Addition of External Oracles
2. Reasoning about infinite sequences in ACL2

External Oracles

- We proved that the original verification problem is satisfied if and only if each of the “simpler” verification problems is satisfied.
- For a particular verification problem we want:
  - To use the algorithm to decompose it into a simpler problem.
  - To use an efficient model checker to model check each of the simpler problems.
- But we do not want to implement an efficient LTL model checker in ACL2.
  - There are trusted model checkers in the market to do the job.
  - As long as we believe that the external checkers satisfy the semantics we provided in ACL2, we should be allowed to invoke them.

Intermediate hack

- Define an executable function ltl-hack with a guard of T.
- Define axiom positing ltl-hack is logically equivalent to the logical definition of semantics of LTL.
- In the Lisp, replace the definition of ltl-hack to a syscall that calls the external model checker (Cadence SMV).
- We have used the composite system to check simple LTL properties of system models using our compositional algorithm.

Proposal: External Oracles

- Note that if `ltl-hack` is not an LTL model checker then the axiom posited makes the logic unsound.
  - We have never used the logical body of `ltl-hack`, but a `:use` hint expanding the body will enable you to prove `nil`!
- Can ACL2 give us a better way of integrating an external tool?
  - It is important for ACL2 not to be monolithic.
  - Other theorem provers like Isabelle have such capability.

Infinite Sequences: Recursion with Quantifiers

- To define the natural semantics of LTL, we need quantification with recursion (plus some axiomatization of infinite paths).
  - ACL2 does not allow recursion with quantification.
  - The addition of such facility violates "conservativity" of the logic.
- We have claimed that having addition of such facility is sound, though not conservative.
- Is it possible to reduce the restrictions imposed by ACL2?
  - Is such an extension possible with ACL2(R)?

Questions?