Certifying Compositional Model-checking Reductions in ACL2

Sandip Ray

ACL2 meeting presentation

Feb 12, 2003

(Joint Work with John Matthews)

Outline

- Background and motivation
- Framework
- Goal Theorems
- Proofs
- Issues and Proposals

Background: Model Checking

- Model-checking is a procedure for automatically deducing "temporal properties" of "reactive computer systems".
  - The properties are specified in terms of a "*Temporal Logic*" (CTL, LTL *etc.*)
  - The system is specified in terms of a "*Kripke Structure*".
  - The properties are often analyzed by an intelligent, but extensive "graph-search" algorithm.
  - When it works it is a push-button tool.
  - For large systems it does not work because of *state explosion*.
    - The system then needs to be "reduced" or "abstracted" in some way so that the reduced system can be analyzed.

Model-checking: Good, Bad and Ugly

- The Good:
  - It is a push-button tool (unlike theorem-proving) when it works.
  - If the property is violated by the system, then model-checking produces a counter-example automatically.
- The Bad:
  - Has problems with scaling to large systems because of *state explosion*.
- The Ugly:
  - The system (or the property) needs to be abstracted sufficiently so that a model-checker can dispatch the problem.

Compositional Reductions

- Replace the original verification problem into one or more simpler problems.
  - Exploit system characteristics like symmetry, cone of influence, *etc.,* and characteristics of property, like conjunction.
  - Strong reductions can be performed by effective knowledge of the domain.
  - Reductions are often non-trivial.
  - Each of the reduced problems are then dispatched by a model-checker or further reduced.

Problems

- Compositional reductions are often complex.
  - Need to insure that the reductions are sound.
- Possibilities:
  - Use theorem-proving to verify soundness of reductions.
    - Requires human interaction, but we have a verified tool.

Our Work

- Verify using ACL2, two very simple compositional reductions.
  - Our basic model-checking logic is Linear Temporal Logic (LTL).
  - We model the semantics of LTL in ACL2.
    - Our formalization of LTL is non-conventional, to say the least (I will discuss why....)
  - Our proof is based on ***our formalization*** of LTL.
  - We face a surprising amount of difficulty trying to specify what we want.
    - I expect discussions, suggestions, and comments on these problems as I go.

Outline

- Background and motivation
- **Framework**
- Goal Theorems
- Proofs
- Issues and proposals

(Really) Broad Overview of our work

- Define a representation of finite state systems (circuit) in ACL2.
- Define functions that translate such representations to Kripke Structures.
- Define the semantics of LTL for Kripke Structures.
- An LTL proposition will be said to be true of a circuit if it is true of the Kripke structure derived from the circuit.
- A verification problem is a pair:
  - A circuit
  - An LTL proposition.
- Reduction algorithms take a verification problem and return a collection of verification problems.
- We prove that:
  - The original verification problem is "equivalent" to the collection of reduced verification problems.

Finite State Systems in ACL2

- We chose our own home-grown model of circuits that was sufficient for our purpose.
  - A circuit is defined by three things:
    - A collection of *state variables* representing 1-bit latches in the circuit.
    - Corresponding to each variable is a list of *Boolean functions (equations)*. These equations tell us, given a current binding of each variable to a Boolean, the possible next bindings.
    - A collection of initial bindings of the variables which correspond to the legal reset or initial states of the system.

Circuits: An example

An example of circuits with two variables v and w:

```
'(;; variables
  (v w)
  ;; equations
  ((v .  ((v & w) (~ v))
   (w . ((v + w))))
  ;; initial-states
  (((v . T ) (w . nil))
   ((v . nil) (w . T))))
```

Thus, given a current binding.

- '((v . T) (w . nil))
- We get a possible next binding '((v . nil) (w . T))

The predicate *circuitp*

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

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

Kripke Structures in ACL2

- Traditional Semantics of LTL model-checking is defined with respect to "execution paths" in *Kripke Structures*.
- To formalize LTL we therefore need
  - A formalization of Kripke Structures.
  - Functions to translate a circuit model into the language of the Kripke Structures.

Kripke Structures

- Given a collection AP of "abstract propositions", a Kripke Structure is specified by a 4-tuple:
  $\langle S, R, L, I \rangle$
- S is a set of ***states***.
- $R \subseteq S \times S$ is called the ***transition relation***.
- $L: S \rightarrow 2^{AP}$ is called the ***state labeling function***.
- $I \subseteq S$ is called the set of ***initial states***.

Paths in a Kripke Structure

- A ***path*** $\langle s_0 s_1 \ldots \rangle$ in a Kripke Structure is an infinite sequence of states such that:
  - $s_i \in S$, for all i.
  - $R(s_i, s_{i+1})$ is true for all i.
- A path is an ***initial path*** of the Kripke Structure if the first node s_0 is in I.
- The semantics of LTL are defined with respect to such paths in the Kripke Structure.
- Given a path $P = \langle s_0 s_1 \ldots \rangle$, P_i is the suffix of P from state s_i.

Kripke Structures from Circuits

- The set AP is the set of variables of the circuit.
- The set of states is the set of possible Boolean bindings to all variables of the circuit.
- Two states p and q are related by R if there exists a collection of functions E, one associated with each variable of the circuit, such that applying the corresponding function of E on each variable produces the assignment q.
- The label function maps each state to the set of variables that are true in that state.

The Predicate circuit-modelp

```
(defun circuit-modelp (m)
  (and (only-evaluations-p (states m) (variables m))
       (all-evaluations-p (states m) (variables m))
       (only-all-truths-p (states m) (label-fn m) (variables m))
       (labels-subset-of-vars (label-fn m) (states m) (variables m))
       (transitions-always-subset-p (states m) (transition m))
       (consp (states m))))
```

We define function create-kripke which has the property

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

Some Technical Problems with the theorem

- The theorem is non-trivial. For example, we define ***all-evaluations-p*** as:

```
(defun evaluation-eq (p q vars)
  (if (endp vars) T
      (and (equal (g (first vars) p) (g (first vars) q))
           (evaluation-eq p q (rest vars)))))

(defun evaluation-p (st vars)
  (if (endp vars) T
      (and (booleanp (g (first vars) st)) (evaluation-p st (rest vars)))))

(defun-sk all-evaluations-p (states vars)
  (forall st (implies (evaluation-p st vars)
                      (evaluation-eq-memberp st states vars))))
```

Challenge: Define function (create-all-evaluations vars) and prove
(all-evaluations-p (create-all-evaluations vars) vars)

The LTL Language: Syntax

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

The Traditional Semantics of LTL

- Given a path *P* of a Kripke Structure *M*, and an LTL formula *f*, when do we claim that *f* ***satisfies*** *M*?
  - If *f* is *true* then *P* satisfies *f*.
  - If f is *false* then *P* does not satisfy *f*.
  - If *f* is a variable *v* then *P* satisfies *f* if and only if *v* is assigned to T in *s0*.
  - If f is of the form *(f1 & f2)* then *P* satisfies *f* if and only if *P* satisfies *f1* and *P* satisfies *f2*.
  - If f is of the form *(f1 + f2)* then *P* satisfies f if and only is *P* satisfies *f1* or *P* satisfies *f2*.

Traditional Semantics of LTL (Contd.)

- If f is of the form $(G\ f')$ then P satisfies f if and only is for every suffix P_i of P, P_i satisfies f'.
- If f is of the form $(F\ f')$ then P satisfies f if and only if there exists some suffix P_i of P such that P_i satisfies f'.
- If f is of the form $(X\ f')$ then P satisfies f if and only if $P1$ satisfies f'.
- If f is of the form $(f1\ U\ f2)$ then P satisfies f if and only if there exists a suffix P_i of P such that P_i satisfies $f2$, and for every j, $0 \leq j < i$, P_j satisfies $f1$.
- If f is of the form $(f1\ W\ f2)$ then P satisfies f if and only if either P satisfies $(f1\ U\ f2)$ or P satisfies $(G\ f1)$.

Traditional Semantics of LTL (Contd.)

❑A Kripke Structure M satisfies f if and only if for every initial path P of M, P satisfies f.

Reminder:

A path is an **infinite** sequence of states.

Semantics of LTL in ACL2: the objective

- If we can define a function which we (hypothetically) call ltl-path-semantics that allows us to install the recursive equations we showed, then we can define the semantics of LTL with respect to a Kripke Structure as:

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

Question: Can we define the function ltl-path-semantics?

Problems with ltl-path-semantics

- To implement the function as is, we need recursion with quantification.

```
(ltl-path-semantics (list 'F prop) path)
=
(exists i (ltl-path-semantics
              prop (suffix i path))
```

- ACL2 does not allow recursion and quantification together (for important logical reasons, as we will see).
- We cannot implement the semantics of LTL as is….

Possible Alternatives: 1

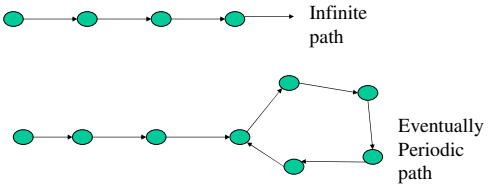
- Implement the semantics of LTL directly with respect to a Kripke Structure. (This is similar to what Pete Manolios did for mu-calculus.)
  - This amounts to implementing an LTL model-checker in ACL2.
- Problems:
  - (This was not considered a worthwhile approach….)
  - The model-checker is not simple enough (unlike mu-calculus) to be called a spec. for LTL.
  - Proofs of some of the reductions actually depend on arguments regarding equivalence of paths (*bisimilarity*) and such proofs would be very different in the new definition.
    - This is discouraged, especially because there is no way to "prove" that the model checker is correct.

Possible Alternatives: 2

- Use of encapsulate.
  - Define *path* as an encapsulated function, and define the semantics of LTL with respect to that encapsulated function.
    - Rob Sumners did proofs of bisimulation and stuttering bisimulation with this notion.
  - Problems:
    - I could not see an easy way of defining the semantics of LTL using encapsulate.
    - I did not pursue this long enough because
      - The crux of the bisimulation argument is in constructing an "equivalent" path, and showing that ltl-path-semantics returns the same value on equivalent paths.
      - Encapsulating *path* and functionally instantiating it with two concrete instances also provides two different instances of ltl-path-semantics.
      - We cannot describe the property that the same function returns identical results on two different paths.

Eventually Periodic Paths

- These are restricted types of paths which end in a cycle.



A Key Observation

- Theorem (Private Communication with Ernie Cohen):
  - If there is an infinite path that does not satisfy an LTL formula, then there is an eventually periodic path that does not satisfy the same formula.
- However,
  - Eventually periodic paths are finite structures representable in ACL2, and
  - A relatively simple (not too simple) version of the semantics of LTL can be modeled with respect to such structures.
    - The function is in the handout.

Reductions

- A compositional reduction algorithm based on two very simple reductions.
  - Conjunctive Reduction
  - Cone of Influence Reduction.

Conjunctive Reduction

- Reduce the checking of a property of the form (f1 & f2) to the checking of two properties:
  - f1
  - f2

```
(defthm conjunctive-reduction-is-sound
   (implies (and (ltl-formulap f)
                 (equal (len f) 3)
                 (equal (second f) '&))
            (equal (ltl-semantics f m)
                   (and (ltl-semantics (first f) m)
                        (ltl-semantics (third f) m)))))
```

Trivial except for trouble with quantifiers in ACL2.

Cone of Influence Reduction

- Basically elimination of unnecessary states.
- Consider a subset *V'* of the set *V* of variables. The cone of influence *K* of *V'* has the property:
  - (subset *V' K)*
  - If for some *v* in *K*, the equations corresponding to *v* depend on *w*, then *w* is in *K*.

Cone of Influence reduction (Contd.)

- Now create a new circuit as follows:
  - The variables are only the variables in *K*.
  - Initial states are deduced from the original initial states by setting every variable not in *K* to be *nil*.
  - The equations are the equations corresponding to the variables in *K*.

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

Details of a "simple" Implementation

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

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

Details of a "simple" Implementation (Contd.)

```
(defun find-all-variables (vars variables equations)
  (if (or (not (uniquep vars))
          (not (uniquep variables))
          (not (subset vars variables)))
      vars
    (let ((new-vars (set-union (find-all-variables-1-pass vars equations)
                               vars)))
      (if (not (subset new-vars variables)) nil
          (if (set-equal vars new-vars) vars
              (find-all-variables new-vars variables equations))))))

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

Outline

- Background and motivation
- Framework
- **Goal Theorems**
- Proofs
- Issues and Proposals

Simple Compositional Reduction

- This algorithm is only for demonstration purposes. The proof of the algorithm is not hard at all, given the proofs of individual reductions, and this seems to be the case for all the compositional algorithms we looked at.
  - Apply conjunctive reduction to the problem and obtain a collection of sub-problems.
  - Reduce each of the sub-problems using cone-of-influence reduction.

Details of Simple Compositional Reduction

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

(defun reduce-problem-cone (f C)
   (let ((vars (variables-in-formula f)))
     (cone-of-influence-reduction C vars)))

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

Details of Simple Reduction (Contd.)

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

Now let us look at the theorems:

```
(defun ltl-ckt-semantics (C f) (ltl-semantics f (create-kripke C)))

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

(defthm compositional-reduction-is-sound
  (implies (and (circuitp C)
                (ltl-formulap f)
                (subset (variables-in-formula f) (variables C)))
           (equal (ltl-semantics-for-circuits* (compositional-reduction C f))
                  (ltl-ckt-semantics C f))))
```

Outline

- Background and motivation
- Framework
- Goal Theorems
- **Proofs**
- Some Issues

Verification of Reductions

- We will only concentrate on cone of influence reduction for present, since the other reduction is simple, and the proof of compositional reduction follows simply from the proofs of the individual reductions.

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

Standard Proof of Cone of Influence

- The standard proof proceeds using bisimulation.
  - We next discuss bisimulation and how it can be employed in the proof.

Bisimulation

- Given two Kripke Structures *M* and *M'* with respect to a set of abstract propositions *AP,* a relation *B* over *S*×*S'* is called a bisimulation relation if and only if for every *s* in *S* and *s'* in *S',* such that *B(s,s'),* we have:
  - *L(s)=L'(s')*
  - For every *s1* in *S* such that *R(s,s1),* there exists *s1'* in *S'* such that *R(s',s1')* and *B(s1,s1').*
  - For every *s1'* in *S'* such that *R'(s',s1'),* there exists *s1* in *S* such that *R(s,s1)* and *B(s1,s1').*
- Two Kripke Structures are bisimilar if and only if there is a bisimulation relation B, such that
  - For every initial state *s0* in *I*, there is a state *s0'* in *I'* such that *B(s0,s0').*
  - For every initial state *s0'* in *I',* there is a state *s0* in *I* such that *B(s0,s0').*

Traditional Theorems about Bisimulation

- **Definition:** Two (infinite) paths P1 and P2 are ***corresponding***, if for every i ≥0, the *i*-th state of P1 is bisimilar to the *i*-th state of P2.
- ***Lemma 1:***
  - Let *s* and *s'* be two states such that *B(s,s').* Then for every path *P* starting from *s* there is a corresponding path *P'* starting from *s'* and for every path *P'* starting from there *s'* is a corresponding path *P* starting from *s*.
- ***Lemma 2:***
  - Let *f* be an LTL formula and let *P* and *P'* be corresponding paths. Then *P* satisfies *f* if and only if *P'* satisfies *f*.

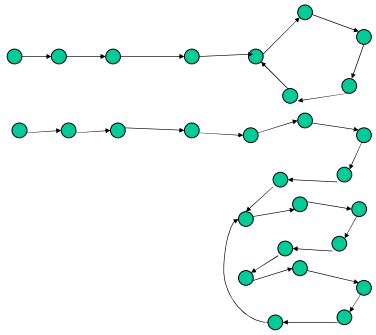
Traditional Proof of Lemma 1

- Let *P=<s0,s1,….>,* and *B(s0,s0').*
- We construct a path *P'* starting from *s0'* by induction.
  - Base case: *B(s0,s0').*
  - Assume $B(s_i, s_i')$ and $R(s_i, s_{i+1})$. We will show how to choose s_{i+1}'.
  - By definition of bisimilarity, there exists a state *t* such that $R'(s_i', t)$, and $B(s_{i+1}, t)$. We choose s_{i+1}' to be *t*.
- ***Comments:*** Is this a proof?

ACL2 equivalent of Lemma 1

- Let *B* be a bisimulation relation, and *B(s,s').* Then for every eventually periodic path starting from *s*, there is a corresponding eventually periodic path starting from *s'*. Further for every eventually periodic path starting from *s',* there is a corresponding eventually periodic path starting from *s*.
  - The statement is similar to Lemma 1, but the proof is complicated by the presence of eventually periodic paths.

Intuitive Idea for ACL2 Proof



Proof Overview

- The proof makes use of Pigeon-hole arguments.
  - It is complicated to justify such an argument in ACL2.
  - We defined a function that returns a sequence of corresponding states until either all states in the system are exhausted, or there has been a duplication.
  - We then justify why there must be a duplication.
- Our approach:
  - Use of mv to return a number of things in addition to the path.
  - Proof showing relation between all the different things returned by mv to justify that we have an eventually periodic path.
  - The function is in the handout.

Proof of Cone of Influence Reduction from Bisimulation Proof

- We simply define a "bisimulation relation" between the Kripke Structure for the original system and the Kripke Structure for the reduced system.
  - Problems:
    - The *AP* (variables) of the two systems are not the same.
    - We have to define what is meant by bisimilarity with respect to a collection of variables.
    - (Aside) This is a problem (or advantage) of formalism. I did not find any discussion regarding this issue in the literature, and never thought of this before I did the proof. (End Aside)

What is the bisimilarity Relation for COI?

- Two states are bisimilar if and only if they have the same assignment for the variables in cone of influence.
  - This is shown to be a "bisimilarity relation".
  - The remaining results follow from the lemmas 1 and 2.

Bisimulation Arguments in ACL2: Previous Approaches

- Pete Manolios, Kedar Namjoshi, Rob Sumners defined the notion of bisimulation (and stuttering bisimulation) in ACL2.
  - They used this approach to verify a pipelined machine at different levels of abstraction, and some concurrent protocols.
  - Why could I not use their approach in my work?

What did they do?

- They justified bisimilarity by showing single-step theorems.
- They proved such single-step theorems to justify that the actual implementations are "bisimilar" to some very simple programs.
- They sometimes model-checked these "simpler programs".

What did they do?

- But
  - These verifications (Ref: Pete, Kedar and Rob's paper in CAV 99, and Rob's paper in ACL2 Workshop 2000) do not attempt to show a final ACL2 theorem that states every infinite execution of the two programs have certain property.
    - Their's is a pragmatic approach.
    - Often, in their case, the states are ACL2 terms because of a shallower embedding of Kripke Structures in ACL2 (ref Rob's ACL2 workshop 2000 paper).
    - The book I saw of their showing an "equivalent" of our theorem is in a bisimilarity book Rob showed me.

Why Couldn't I use it?

- Rob's book shows that if there is a bisimulation relation then for every path there is a matching path.
  - It defines two encapsulated functions path and path+ where the first state of path is bisimilar to the first state of path+.
  - It shows that (path i) is bisimilar to (path+ i).
  - But of course, there is encapsulation in path, which in our case will cause problems since ltl-semantics needs to be functionally instantiated if we produce an instance of path.
    - Hence we could not state what we wanted.

Outline

- Background and motivation
- Framework
- Goal Theorems
- Proofs
- **Issues and Proposals**

Some Efficiency Considerations

- In theory we can implement an LTL model-checker in ACL2 and show it returns the same value as ltl-semantics.
  - This was not considered worthwhile utilization of my time because:
    - We were focused on verifying reductions and not implementing model-checkers.
    - There are "trusted" model-checkers in the market and my sponsors were willing to believe in the soundness of the basic model-checker as much as they will trust ACL2.
    - We did not believe I can implement a model-checker in ACL2 that is nearly as efficient as a commercial model-checker.
- We wanted to find a way of hooking an external algorithm into ACL2.

Unsound Use of ACL2

- We decided to "cheat".
  - Defined a function *ltl-semantics-hack* with a guard of T.
  - Defined Axioms showing that the function *ltl-semantics-hack* is equal to

    (ltl-semantics f (create-kripke C))
  - In the underlying lisp, we redefine the function to a function calling an underlying system (Unix) script, using syscall.
  - The script parses the current circuit description into something a standard model-checker (SMV) understands and calls the external model-checker.
  - Returns a value based on whether the external model-checking succeeded.

Unsound Use (Contd.)

- The underlying script is in ACL2 and perl.
- I judiciously avoided doing I/O in ACL2 (we do it in the script underlying syscall) because I was not sure ACL2 will allow me to pass the real state for that purpose in the middle of a proof.
  - The script seems to be working but is not well-tested yet.
  - Thanks to Robert, Bogo and Rob for helping out with the idea.

Why is it important to have "Unsound Use"

- ACL2 is too monolithic.
  - We need to "reinvent the wheel" every time we want to add a procedure to ACL2.
  - It is nice to be able to hook an external trusted algorithms into ACL2.
  - J Moore suggested in his proposal for a "verified stack" that:
    - Several analysis tools might have to work in conjunction to make such a verification possible.
  - If that is the case, we then need a way for several analysis tools to talk to ACL2.

Proposal 1

- There should be a certified way of hooking an external decision procedure into ACL2.
  - Currently, with my hack, I cannot certify books (of course!!!)
- Such actions might be tantamount to adding a new *defaxiom* event with a note that the axiom has been checked by an external tool and not ACL2.
  - Of course the external logic needs to be consistent with ACL2 (for example it cannot be ACL2 (R), right?)

Why is it Important to Verify Reductions

- ACL2 has the semantics of a programming language.
  - We can represent models of hardware and other systems as ACL2 objects.
  - We can define functions that analyze such objects, like a model-checker.
  - We want to be able to run such analyzers and analyze properties of such models.
  - If the analyzer returns an answer we want to interpret such an answer as a theorem about the object.
- How can we do that?

Importance to ACL2 (Contd.)

- If we can prove a theorem characterizing such an analyzer:
  - That theorem will tell us what we can deduce if the analyzer returns an answer.
- Consider a tautology checker function that analyzes whether a Boolean expression is a tautology.
  - We can have a theorem that says: "If the checker returns *true*, then any evaluation of the input expression will evaluate to *true*."
- We want to do the same thing about other decision procedures for finite state machines.
  - If we can do that, we can claim that we have all the power of automatic analysis plus the power of theorem-proving, which I think is a win.

Why is Our Result not satisfactory

- Our modeling of semantics of LTL is in terms of eventually periodic paths.
  - This is not how the model-checking people visualize LTL properties.
  - This is based on theorems (relating infinite and eventually periodic paths) that cannot be stated (and hence cannot be proved) in ACL2.
  - We will eventually wish to extend our work to model-checking (parametric) infinite state systems.
    - But the theorems are not true for such systems.
- Plus the proofs are more complex and sometimes (as we saw) very different from traditional (and hence anticipated) proofs.

Proposal 2: What Did I need?

- Some way of having recursions with quantified expressions.
  - ACL2 does not allow this.
- Why?

Conservativity

- ACL2 has a property called *conservativity*.
  - Assume we have defined functions *f1, f2,…, fn*.
  - We prove a theorem *TH* which talks about these functions.
  - In order to prove the theorem we introduce auxiliary functions *g1,…gm.*
  - We then put the whole theory in a book and mark all events other than *TH* and definitions of *f1,…, fn* local.
  - So if someone includes this book, he should expect that TH follows from definitions of *f1,…,fn* in first order logic (with epsilon-0 induction).
  - This property is known as the ***conservativity property***.
  - It allows use of encapsulate and local events.

Problems with Recursion and Quantification

- We can define the predicate *true-formula* that declares whether a first-order formula is true.
  - The definition is in the handout.
  - Note: Thanks to Matt for pointing this out to us.
- Well known:
  - If we have the axioms of Peano arithmetic together with the axioms for *true-formula*, then we can prove the consistency of Peano arithmetic.
  - Hence the definition of *true-formula* is not conservative by Godel's Incompleteness Theorem.
- Adding recursion and quantification is going to be a strictly stronger, non-conservative extension of ACL2.

Is ACL2(R) also non-conservative?

- **Disclaimer:** I have very little knowledge about the logic of ACL2 (R) or about non-standard analysis.
  - My exposure is talks from Matt, Ruben, and Brittany in this meeting.
- We believe, yes.
  - Consider a model of Peano Arithmetic in ACL2(R) as follows.
    - The natural numbers and arithmetic operators of PA are corresponding operators in ACL2(R).
    - Formulas of PA are encoded using real numbers. (Sorry if this is B.S….)

Encoding PA functions as Real Numbers

- Consider the range R1=[-1,1]
- The domain is the set of functions P=Nat→Bool.
- For every predicate *p* in *P*, map p to a real number *r* in *R1*.
  - The binary expansion of r is the truth table of p.
  - This means that *p(i)* is encoded as 2^{-(i+1)}.
  - Using Godel encodings, we can encode predicates from tuples of numbers to *R1* as well.
  - It should be possible to define every PA operator, including quantification.
  - We can then define the function true-formula as the ACL2(R) function.

How Can we claim consistency in spite of non-conservativity

- We think a non-conservative extension like having recursion with quantification is still consistent *w.r.t.* a higher order logic.
  - Assume there is a collection *M* of models of ACL2 in *ZFC*.
  - For every model *m* in *M*, the class of ACL2 objects is an element of some class *k* of *ZFC*.
  - Every function in ACL2 is mapped to some function in the set $k \rightarrow k$.
  - Then the axioms and inference rules are theorems in *ZFC* regarding model *m*.
  - However, for every such model *m*, we can have quantification and recursion in *ZFC* using the axiom of choice.
  - Hence consistency of augmented models with quantification and recursion will follow from consistency of *ZFC*.

Is such an extension consistent with ACL2(R)?

- Every model of ZFC can be transformed to a model of non-standard analysis.
  - Justification of recursion and quantification is given for every model *inside* ZFC.
  - Each such model, then, can be transformed into a non-standard model which satisfies all axioms of quantified recursion (for standard objects) as well as non-standard axioms.
- Can we have recursion and quantification inside ACL2(R) right now?