# Integrating External Deduction Tools with ACL2

Matt Kaufmann, J Moore, **Sandip Ray**, Erik Reeber

Department of Computer Sciences

University of Texas at Austin

Austin, TX 78712

Presentation for IWIL 2006

# A Sincere Apology

I am sorry I could not be in the conference to give this talk in person. My re-entry visa to the United States was unexpectedly refused pending security clearance and background checks and hence I cannot make international travels until the checks are over. I am disappointed at this unfortunate event, and sincerely apologize to the conference organizers and the audience for the inconvenience.

# Motivation

**We are interested in increasing the effectiveness of interactive theorem proving and its application to the formal verification of large-scale computing systems.**

There has been significant enhancement in the scalability of automated reasoning procedures in recent years.

- SAT and SMT solvers
- Model checkers
- Resolution theorem provers
- Decision procedures for theories like Presburger Arithmetic

It is desirable for interactive theorem provers to exploit the automation provided by these procedures.

# Motivation

But different reasoning tools implement different logics!!

- **Theorem provers:** FOL or HOL, constructive type theory, etc.
- **Model checkers:** Flavors of temporal logic, $\mu$-calculus, etc.

Different tools also have different formal syntax.

**To make an effective connection between a theorem prover and another reasoning tool, one must know how to interpret a formula proven by the tool as a theorem in the logic of the theorem prover.**

# Our Work

**We provide an interface for sound connection of external reasoning tools with the ACL2 theorem prover.**

- Provides a precise description of the requirements that the developer of an external tool connection with ACL2 must meet.
- Provides facilities and constructs so that a developer can meet these requirements.

We also provide a modest enhancement to the ACL2 logic to enable connection with certain types of external tools.

# ACL2

"A Computational Logic for Applicative Common Lisp"

Co-authored by **Matt Kaufmann** and **J Moore**, with early contributions from **Bob Boyer**

- An industrial-strength version of the Boyer-Moore theorem prover.

Provides

- An applicative programming language based on Common Lisp
- A first-order logic of recursive functions (with induction)
- A mechanical theorem prover for the logic

**Has been used for some of the largest formal verification projects.**

- AMD K5™ floating point division microcode algorithm
- Floating point RTL designs of AMD processors
- Motorola CAP DSP
- Rockwell Collins AAMP™ series of microprocessors
- ....

# Our Immediate Motivation

These projects involve several man-years of work!!

Yet, many of the lemmas proven could have been stated in a decidable theory, and discharged by a decision procedure for the theory.

- **Key example:** Proving invariants for hardware models

Other theorem provers (HOL, PVS, etc.) provide interfaces to external tools or oracles.

# Non-Trivialities

The ACL2 logic is complicated by several constructs provided to structured proofs.

- The complexity manifests itself in some aspects of the interface.

**This also makes developing a precise interface very important.**

- A tool developer without adequate understanding of complications might inadvertently produce an unsound connection.

# Deduction Tools

**We are developing a *generic* interface to connect ACL2 with external deduction tools.**

- The interface is under development and will be available with the next release of the theorem prover.

## Three classes of Deduction Tools are supported

- Tools verified by the theorem prover itself
- Unverified tools using a theory fully formalized in ACL2
- Tools implementing theories that are incompletely formalized

# Why Verified Tools?

The formal language of ACL2 is essentially an applicative subset of Common Lisp.

- It is possible to write efficient code using this language.
- ACL2 itself is substantially implemented in ACL2.

**Ideally, a user should be able to control proofs by**

- implementing domain-specific reasoning code
- verifying with ACL2 that the code is sound
- invoking such code for proving theorems in the target domain

This enables the use of ACL2 as a customized reasoning engine for the domain, while requiring no code other than the theorem prover itself to be trusted.

# ACL2: User's View

Definitions:

*rev1(l, a)* = *if consp(l) then rev1(cdr(l), cons(car(l), a) else a*

*rev(l)* = *rev1(l, nil)*

*true_listp (l)* = *if consp(l) then true_listp(cdr(l)) else (l=nil)*

Goal Theorem:

true_*listp(l)* ⇒ *rev(rev(l)) = l*

# ACL2: User's View

Definitions:

*rev1(l, a)* = *if consp(l) then rev1(cdr(l), cons(car(l), a) else a*

*rev(l)* = *rev1(l, nil)*

*true_listp (l)* = *if consp(l) then true_listp(cdr(l)) else (l=nil)*

*app(p,q)* = *if consp(p) then cons(car(p), app(cdr(p),q)) else q*

*rev0(l)* = *if consp(l) then app(rev0(cdr(l)), cons(car(l), nil)) else nil*

Theorem

*rev1(x,a) = app(rev0(x), a)*

**hints** At Goal induct *rev1(x,a)*

rule classes rewrite

Theorem *true_listp(l) ⇒ rev(l) = rev0(l)*

Theorem

*true_listp(l) ⇒ rev0(rev0(l)) = l*

rule classes rewrite

Theorem

*true_listp(l) ⇒ rev(rev(l)) = l*

# Book Certification

Definitions:

*rev1(l, a)* = *if consp(l) then rev1(cdr(l), cons(car(l), a) else a*

*rev(l)* = *rev1(l, nil)*

*true_listp (l)* = *if consp(l) then true_listp(cdr(l)) else (l=nil)*

Local Definitions

*app(p,q) = if consp(p) then cons(car(p), app(cdr(p),q)) else q*

*rev0(l) = if consp(l) then app(rev0(cdr(l)), cons(car(l), nil)) else nil*

Local Theorem

*rev1(x,a) = app(rev0(x), a*)

hints At Goal induct rev1(x,a)

rule classes rewrite

Local Theorem *true_listp(l) ⇒ rev(l) = rev0(l)*

Local Theorem

*true_listp(l) ⇒ rev0(rev0(l)) = l*

rule classes rewrite

Theorem

*true_listp(l) ⇒ rev(rev(l)) = l*

# Book Inclusion

**If we include the book in a subsequent ACL2 session, we get only the non-local definitions and theorems.**

Definitions:

*rev1(l, a)* = if *consp(l)* then *rev1(cdr(l), cons(car(l), a)* else *a*

*rev(l)* = *rev1(l, nil)*

*true_listp (l)* = if *consp(l)* then *true_listp(cdr(l))* else *(l*=nil*)*

Theorem

*true_listp(l)* ⇒ *rev(rev(l)) = l*

**Why is this sound?**

**Theorem (Kaufmann and Moore, 2001):** *Every formula proven by ACL2 is in fact first-order derivable (with induction) from the initial boot-strap theory together with (hereditarily) only the axioms that involve the function symbols in the formula.*

# Deduction Tools as Clause Processors

Internally, ACL2 stores a goal as a *clause*: A list of formulas.

$P \Rightarrow Q$ is stored as the clause **(¬P Q)**

**A deduction tool takes such a clause and returns a list of clauses.**

- It can be invoked as a hint to simplify a goal conjecture.

Our interface allows the user to issue a command like:

Theorem

*true_listp(l)* ⇒ *rev(rev(l))* = *l*

**hints** At Subgoal 2 use *foo* as a **clause processor**

**If *foo* has been registered with ACL2 as a clause processor** then

ACL2 applies the function *foo* on the internal representation of Subgoal 2, and then attempts to prove each clause in the list returned by *foo*.

# Registering a Clause Processor

A function *foo* can be registered as a clause processor in two ways.

**Verified Case:** By proving a certain theorem about *foo* with a special (newly implemented) rule class.

**Unverified Case:** By declaring it as an unverified clause processor.

# Verified Clause Processors

Informal Statement:

Suppose the internal representation of a formula is clause C, and that when foo is applied to C the result is a list L of clauses. If the formula corresponding to each clause in L is provable in the current theory then so is C.

**Key Observations:**

- The clause processor operates on the internal representations of formulas (clauses), which are objects in the logic.
- It is possible to define a function in the logic that specifies an evaluation semantics for a clause.
- A sufficient condition for the provability of the formula is that the corresponding formula never evaluates to nil.
- This condition can be stated as a formula in the logic.

See the paper for details.

**(Aside)** ACL2 currently provides a "meta-reasoning" facility which is also based on a similar argument.

# Unverified Tools

We provide an interface in which a user can declare a function to be an unverified clause processor.

- ACL2 provides a logic-free programming facility (also an operating system interface for invoking arbitrary executables) which can be used for the implementation of an unverified clause processor.

Expectation from the developer of an unverified tool:

- The implementation satisfies the provability condition.

**Certification of a book with a declaration of an unverified clause processor requires an acknowledgement tag.**

- Any subsequent book that includes a tagged book (even locally) inherits the acknowledgement tag from that included book.

**(Aside)** Currently a more general tagging mechanism is being developed with ACL2 which will serve as a foundation for tagging of unverified clause processors.

# Complication: Local definitions

Consider the file book1 with the following commands.

Local Definition *fn(x) = x*

Definition *tool0(C) = …*

Declaration of *tool0* as unverified clause processor

Assume that *tool0* replaces terms **fn(A)** with **A**.

Now consider book2:

include book1

Definition *fn(x) = cons(x,x)*

**Then anything we do in book2 with *tool0* is unsound!!**

# Solution: Supporters

We allow the implementer of *tool0* to name a list of axiomatic events, such as the definition of fn, as supporters to an unverified clause processor.

**Any declaration installing an unverified clause processor causes ACL2 to check that the supporting events have already been defined.**

- This will cause the certification of book1 to fail in our example, as desired.

Local Definition *fn(x) = x*

Definition *tool0(C) = …*

Declaration of *tool0* as unverified clause processor

# "Advanced" Unverified Tools

We will now consider tools that implement their own theory which is not formalized in ACL2.

**"Why would you want to do that?"**

# Motivating Remarks for Advanced Tools

**A key application of formal methods is in reasoning about hardware designs.**

- Such designs are implemented in HDLs (VHDL or Verilog)
- To formalize them in ACL2 we must **define** a formal interpreter of the HDL in ACL2.

An alternative is to merely **constrain** the properties of the interpreter as an incomplete theory.

- Use HDL-level tools to deduce low-level properties of the actual design.
- Use these properties together with the constraints introduced to deduce the final theorem in ACL2.

# Example: ACL2 and SixthSense

**Sawada and Reeber verify a floating-point VHDL multiplier with a combination of ACL2 and SixthSense.**

- Interpreter partially formalized with two functions sigvec and sigbit, axiomatized to only return a bit and bit vector.
- SixthSense proves key properties of different multiplier stages.

By combining these properties with ACL2 they prove that

*If the interpreter is given a requisite design and two bit vectors of the right size, then after 5 cycles, the output of a specific signal is the product of the two inputs.*

# Encapsulation: Introducing Constraints

ACL2 allows introduction of constrained functions via its ***encapsulation principle***.

Introduce *f* with constraint:

*natp(f()) = T*

For soundness ACL2 requires the user to provide a local witness function that satisfies the constraints.

ACL2 also has a derived rule of inference, ***functional instantiation***.

If *bar(f())* is a theorem then so is *bar(10)*.

# Incomplete Theories

We cannot use encapsulation to introduce incomplete constraints.

- If we introduced *sigbit* and *sigvec* as encapsulated functions and used SixthSense to prove the correctness of the multiplier as a theorem, then functional instantiation would immediately produce inconsistency.

## Our solution

- A new extension principle, **templates**.
- Just like encapsulations, except that we acknowledge that the set of constraints explicitly specified might be incomplete.
  - **Functional instantiation is not permitted** to instantiate functions introduced as templates.

# Templates and Generalized Tools

**A clause processor can have a template function as a supporting event.**

The tool implementer must guarantee

- The template could in principle be extended to an encapsulation which would have been permissible in place of the template.
- Any theorem proven by the external tool is a theorem in the theory in which the template were replaced by the promised encapsulation.

- ACL2 does not allow a template function to be a supporter of two different clause processors, to enable the tool implementer to provide such guarantee.

More details in the paper.

# What Is ACL2's Logical Story?

Transform each template in the ACL2 session with an admissible encapsulation. (If some tool has it as a supporter we must transform the template to the encapsulate promised by the tool developer.) Every alleged theorem in the session is then first-order derivable (with induction) from the axiomatic events of the session produced after this transformation.

# Related Work

Theorem provers such as HOL, PVS, Isabelle, etc., provide a notion of external oracles to connect the theorem prover with other tools.

For Isabelle and HOL, this is done by a concept of "theorem tagging" introduced by Elsa Gunter.

- A tag is a function **in the logic**.
- Each formula certified by the oracle is viewed as an implication with a tag for the oracle in the hypothesis.
- This approach tracks unverified tools at the theorem level.

Our tagging mechanism tracks use of untrusted tools at the book level.

- This is a design decision.
- Our initial work attempting to introduce tag as theorem level revealed that the logical foundations were complicated in the presence of local events.

**We believe our approach will be effective for ACL2 since users typically find it easy to move events between books.**

# Random Concluding Remarks

## Does the builder of the connection need to be an ACL2 expert?

- She must understand the ACL2 logic and the logic of the other tool.
  - Anyone interested in building sound connection between two formal tools must clearly understand the individual logics.
- No knowledge of the internals of ACL2 reasoning engine is necessary for soundness.

## We hope that the ACL2 users will benefit from the interface.

## A Future Challenge:

Can we create a logical foundation so that the user of the interface can make use of more expressive logics (say HOL) for discharging some ACL2 obligation?

- **Note:** There is already work in progress in the other direction (Gordon et al., 2006) to use ACL2 as an oracle for HOL4.

# Questions?

Please feel free to email us questions at:

kaufmann@cs.utexas.edu

moore@cs.utexas.edu

sandip@cs.utexas.edu

reeber@cs.utexas.edu

**Again, I apologize for not being able to give this talk in person.**