ACL2: A Program Verifier for Applicative Common Lisp

Matt Kaufmann, AMD, Austin, Tx
J Strother Moore, CS Dept, UT, Austin, Tx

Our Hypothesis

The "high cost" of formal methods (when real) is a *historical anomaly* due to the fact that virtually every project formally recapitulates the past.

The use of mechanized formal methods will ultimately

- *decrease* time-to-market, and
- *increase* reliability.

ACL2 Pre-History: Nqthm

- CLI stack

- C String Library (verified gcc -o 68020 binaries)

stressed Nqthm and led to ACL2.

Goals:

- ANSI standard language and logic;
- execution efficiency;
- scale.

Some Interesting Software Verified

- Motorola CAP DSP (Brock)

$$\neg hazards(s) \rightarrow microarch(s) = isa(s)''$$

  - more than 100 pages of models
  - established correctness of isa as a simulator
  - isa (in ACL2) executes **3 times faster** than Motorola's own $microarch$ simulator (in C).

- FDIV on AMD K5 (Moore, Kaufmann, Lynch)
- elementary fp RTL for AMD Athlon, Opteron, and other FPUs (Russinoff, Flatau)
- FDIV and FSQRT on IBM Power 4 (Sawada)

ACL2 fp models (mechanically translated from RTL) are run on hundreds of millions of fp test vectors.

- soundness of the Ivy proof checker (McCune, Shumsky)

- correctness of BDD implementation (60% CUDD speed) (Sumners)

- Union Switch & Signal post-compiler checker (Bertolli)

This ACL2 code is run and trusted.

- Rockwell instruction set equivalence theorems (Greve, Wilding)
- Rockwell AAMP7 separation kernel in microcode (Greve, Wilding)
- Rockwell / aJile Systems JEM1 (Hardin, Greve, Wilding)

ACL2 model of JEM1 replaced the C simulator for requirements testing.

- Sun CLDC JVM model (700 pages) (Liu)
- properties of various Java classes and methods via javac (Moore, Smith)
- correctness of Sun JVM class loader (Liu)
- correctness of Sun JVM bytecode verifier (Liu) (in progress)

- Bryant’s Y86 Verilog (microarchitecture correspondence) (Ray)
- Dijkstra’s shortest path (Moore, Zhang)
- Unicode reader (Davis)

Proof Techniques

These proofs all involve lemmas about inductively defined objects and concepts.

- induction: appropriate inductions for many simple theorems can be discovered by analysis and manipulation of the recursive functions used in the conjecture.

- simplification:

- congruence-based rewriting
- conditional rewriting
- use of recursively defined function definitions
- use of previously proved theorems as rules
- efficient representation of constants
- very efficient evaluation
- fixed-point based typing procedure
- back- and forward chaining
- integrated linear arithmetic decision procedure
- extensions supporting non-linear arithmetic

- integrated equality decision procedure
- integrated BDD decision procedure
- integrated SAT decision procedure (in progress)
- reflection (metafunctions)
- single-threaded objects ("monads")
- beta-reduction avoidance
- decades of engineering

Operational Semantics

Our proofs are often based on operational semantics.

We define interpreters (for netlists, bytecode, etc.).

We map programs into this code via conventional compilers (gcc, javac).

Via this “deep embedding” we move up and down the abstraction hierarchy while staying in one logic and proof system.

By posing the right theorem to prove, we can make the simplifier act like a VCG for any given operational semantics.

We have methods of mixing operational semantics and inductive assertion-style proofs.

We do both partial and total correctness proofs.

Present - Why are we succeeding?

Reason 1: Our modeling language and logic is an ANSI standard executable programming language:

- widely supported on many platforms
- extremely good compilers
- editors and debugging tools provided by others

Reason 2: We have invested 34 years

- supporting efficient execution
- integrating a wide variety of proof techniques
- engineering for industrial scale formulas
- developing reusable "books" (lemma collections)

Reason 3: We have chosen the right problems. In our applications, the models

- are useful software artifacts
- are often executed to convince designers and management that verification is relevant
- are useful as simulation engines
- permit mathematical abstraction and proof

Drawbacks

The ACL2 user must be trained to *use* the tool, not *fight* it.

The language is not as expressive as many would like (but it is executable).

Finding appropriate lemmas is often a hard (mathematical) problem.

Developing compatible collections of theorems takes system design (not just mathematical) talent.

Short Term Future

- Create better books (especially for arithmetic).
- Integrate FSM decision procedures.
- Do more (non-combinatoric) search.
- Exploit static analysis tools: import lemmas proved by shape analysis, etc.
- Import lemmas proved by other theorem provers.

Medium Term Future

- Incorporate heuristics for discovering invariants.
- Exploit examples to guide search.
- Parallelize the programming language.
- Call on other theorem provers.
- Support interactive steering and visualization.

Long Term Future

- Discover how to mechanize mathematical
  - decomposition,
  - abstraction, and
  - generalization.
- Build a powerful and mechanically verified theorem prover.