Industrial Use of a Mechanical Theorem Prover

J Strother Moore
Department of Computer Science
University of Texas at Austin

July, 2017

About “Big Proof”

This 6 week program focuses on mechanically checked proofs of conventional mathematical challenges.

But there is another important opportunity for mathematicians and mechanized provers.

A New Era

Formal mathematical logic was a cumbersome solution in search of a practical problem

A New Era

Formal mathematical logic was a cumbersome solution in search of a practical problem *until the invention of the digital computer.*

- Digital artifacts are formal systems.
- Computers allow construction of trustworthy formal proofs.

This is an extraordinarily young development (wrt the history of formalized mathematics) and offers many opportunities for discovery.

Theorems about These Artifacts Matter to Everyone Who. . .

- owns a mobile phone, TV, modern appliance
- rides in an automobile, airplane, or train
- visits a doctor or hospital
- uses online banking
- wishes to avoid an accidental nuclear exchange
- . . .

How can one check a routine in the sense of making sure that it is right?

In order that the man who checks may not have too difficult a task the programmer should make a number of definite assertions which can be checked individually, and from which the correctness of the whole programme easily follows.

— *Alan Turing, "Checking a large routine," 1949*

Instead of debugging a program, one should prove that it meets its specifications, and this proof should be checked by a computer program.

— *John McCarthy, "A Basis for a Mathematical Theory of Computation," 1961*

We can formalize the semantics of computer hardware and software.

We can formalize the semantics of computer hardware and software.

We can prove properties of these formal models.

We can formalize the semantics of computer hardware and software.

We can prove properties of these formal models.

We can check – or even discover – these proofs with mechanical theorem provers.

We can formalize the semantics of computer hardware and software.

We can prove properties of these formal models.

We can check – or even discover – these proofs with mechanical theorem provers.

But how practical is this goal?

ACL2

A Computational **L**ogic for
Applicative **C**ommon **L**isp

A fully integrated verification environment for a practical applicative subset of an ANSI standard programming language

{kaufmann,moore}@cs.utexas.edu
http://www.cs.utexas.edu/users/moore/acl2

Idea: Take a functional programming language, axiomatize it, build a theorem prover for it, and use it to model other computational artifacts.

Demo 1

A Few Bullets About ACL2

- Common Lisp (an ANSI standard language)
- prefix syntax of Lisp
- data types: integers, rationals, characters, strings, lists and trees (ordered pairs)
- first order
- untyped syntax
- induction up to $\epsilon_0 = \omega^{\omega^{\omega^{\cdots}}}$

- conservative principle of definition
- recursive definitions must be proved total
- formulas are implicitly univerally quantified
- no explicit quantification
- support for enforcing "type-like" discipline on function calls (like PVS type conditions)
- support quantification via Hilbert-like `defchoose`

- support for the reals via Robinson's Non-Standard Analysis in ACL2(R)
- support some second-order reasoning via the derived rule of *functional instantiation*
- support for user-defined equivalence relations
- support for congruence-based reasoning via rewriting
- support for verified metafunctions

- . . .

Tomorrow's talk will explore more about ACL2.

Example: Formalize a Simple Machine

- $state : \langle pc, registers, stack, program \rangle$
- $instr : \langle opcode, arg_1, \ldots, arg_n \rangle$
- $step(instr, state) \Rightarrow state$
- $m1(state, n) \Rightarrow state$

Demo 2

In the following demonstration we show *one* way to formalize semantics and prove program properties.

There are many, including inductive assertions, McCarthy's functional semantics, Hoare logic, weakest preconditions, denotational semantics, etc.

All of these can be formalized in ACL2. We chose this method for its simplicity.

But can this be scaled up to problems of industrial interest?

An elusive circuitry error is causing a chip used in millions of computers to generate inaccurate results

— *NY Times, "Circuit Flaw Causes Pentium Chip to Miscalculate, Intel Admits," Nov 11, 1994*

Intel Corp. last week took a $475 million write-off to cover costs associated with the divide bug in the Pentium microprocessor's floating-point unit — *EE Times, Jan 23, 1995*

IEEE 754 Floating Point Standard

Elementary operations are to be performed as though the infinitely precise (standard mathematical) operation were performed and then the result rounded to the indicated precision.

AMD K5 Algorithm FDIV$(p, d, mode)$

1.	$sd_0 = \text{lookup}(d)$	`[exact 17 8]`	
2.	$d_r = d$	`[away 17 32]`	
3.	$sdd_0 = sd_0 \times d_r$	`[away 17 32]`	
4.	$sd_1 = sd_0 \times \text{comp}(sdd_0, 32)$	`[trunc 17 32]`	
5.	$sdd_1 = sd_1 \times d_r$	`[away 17 32]`	
6.	$sd_2 = sd_1 \times \text{comp}(sdd_1, 32)$	`[trunc 17 32]`	
...	... = ...	...	
29.	$q_3 = sd_2 \times ph_3$	`[trunc 17 24]`	
30.	$qq_2 = q_2 + q_3$	`[sticky 17 64]`	
31.	$qq_1 = qq_2 + q_1$	`[sticky 17 64]`	
32.	$fdiv = qq_1 + q_0$	$mode$	

Using the Reciprocal

```
      3 6.
+       -.1 7
+         .0 0 3 4
+       -.0 0 0 0 6 6
      ---------------
      3 5.8 3 3 3 3 4
   ___________________
1 2 | 4 3 0.0 0 0 0 0 0
      4 3 2.
      -----
        -2.
        -2.0 4
        -----
          .0 4
          .0 4 0 8
          ---------
        - .0 0 0 8
        - .0 0 0 7 9 2
        - -------------
        - .0 0 0 0 0 8
```

Reciprocal Calculation:

$1/12 \;=\; 0.083\overline{3} \approx 0.083 = sd_2$

Quotient Digit Calculation:

$0.083 \times 430.0000 = 35.6900000 \approx 36.000000 = q_0$

$0.083 \times \text{-}2.0000 = \text{-}.1660000 \approx \text{-}.170000 = q_1$

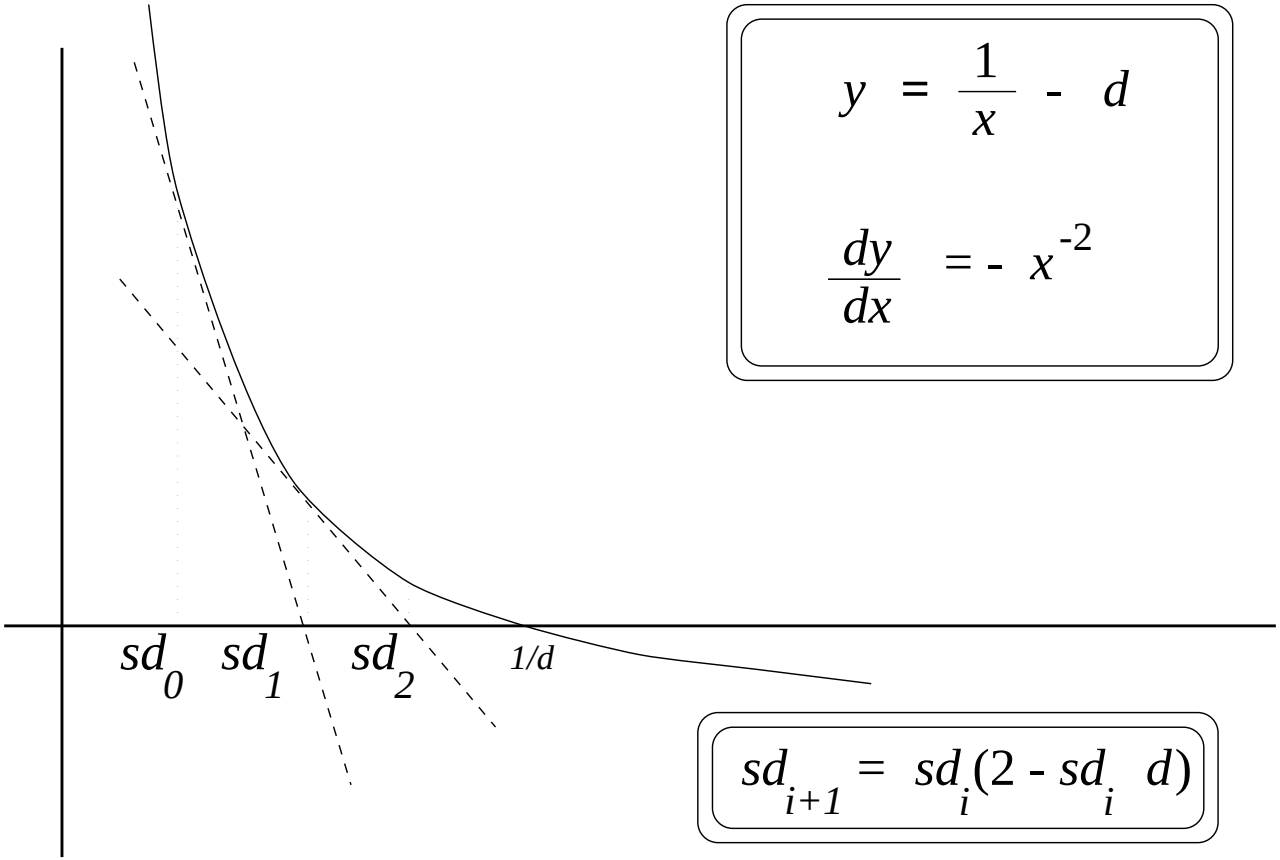
$0.083 \times .0400 = .0033200 \approx .003400 = q_2$

$0.083 \times \text{-}.0008 = \text{-}.0000664 \approx \text{-}.000067 = q_3$

Summation of Quotient Digits:

$q_0 + q_1 + q_2 + q_3 = 35.833333$

Computing the Reciprocal



$$y = \frac{1}{x} - d$$

$$\frac{dy}{dx} = -x^{-2}$$

$$sd_{i+1} = sd_i(2 - sd_i\ d)$$

top 8 bits of d	approx inverse	top 8 bits of d	approx inverse	top 8 bits of d	approx inverse	top 8 bits of d	approx inverse
1.0000000_2	0.11111111_2	1.0100000_2	0.11001100_2	1.1000000_2	0.10101010_2	1.1100000_2	0.10010010_2
1.0000001_2	0.11111101_2	1.0100001_2	0.11001011_2	1.1000001_2	0.10101001_2	1.1100001_2	0.10010001_2
1.0000010_2	0.11111011_2	1.0100010_2	0.11001010_2	1.1000010_2	0.10101000_2	1.1100010_2	0.10010001_2
1.0000011_2	0.11111001_2	1.0100011_2	0.11001000_2	1.1000011_2	0.10101000_2	1.1100011_2	0.10010000_2
1.0000100_2	0.11110111_2	1.0100100_2	0.11000111_2	1.1000100_2	0.10100111_2	1.1100100_2	0.10001111_2
1.0000101_2	0.11110101_2	1.0100101_2	0.11000110_2	1.1000101_2	0.10100110_2	1.1100101_2	0.10001111_2
1.0000110_2	0.11110100_2	1.0100110_2	0.11000101_2	1.1000110_2	0.10100101_2	1.1100110_2	0.10001110_2
1.0000111_2	0.11110010_2	1.0100111_2	0.11000100_2	1.1000111_2	0.10100100_2	1.1100111_2	0.10001110_2
1.0001000_2	0.11110000_2	1.0101000_2	0.11000010_2	1.1001000_2	0.10100011_2	1.1101000_2	0.10001101_2
1.0001001_2	0.11101110_2	1.0101001_2	0.11000001_2	1.1001001_2	0.10100011_2	1.1101001_2	0.10001100_2
1.0001010_2	0.11101101_2	1.0101010_2	0.11000000_2	1.1001010_2	0.10100010_2	1.1101010_2	0.10001100_2
...	...	...	...	...	...	...	...
1.0010110_2	0.11011010_2	1.0110110_2	0.10110100_2	1.1010110_2	0.10011001_2	1.1110110_2	0.10000101_2
1.0010111_2	0.11011000_2	1.0110111_2	0.10110011_2	1.1010111_2	0.10011000_2	1.1110111_2	0.10000100_2
1.0011000_2	0.11010111_2	1.0111000_2	0.10110010_2	1.1011000_2	0.10010111_2	1.1111000_2	0.10000100_2
1.0011001_2	0.11010101_2	1.0111001_2	0.10110001_2	1.1011001_2	0.10010111_2	1.1111001_2	0.10000011_2
1.0011010_2	0.11010100_2	1.0111010_2	0.10110000_2	1.1011010_2	0.10010110_2	1.1111010_2	0.10000011_2
1.0011011_2	0.11010011_2	1.0111011_2	0.10101111_2	1.1011011_2	0.10010101_2	1.1111011_2	0.10000010_2
1.0011100_2	0.11010001_2	1.0111100_2	0.10101110_2	1.1011100_2	0.10010101_2	1.1111100_2	0.10000010_2
1.0011101_2	0.11010000_2	1.0111101_2	0.10101101_2	1.1011101_2	0.10010100_2	1.1111101_2	0.10000001_2
1.0011110_2	0.11001111_2	1.0111110_2	0.10101100_2	1.1011110_2	0.10010011_2	1.1111110_2	0.10000001_2
1.0011111_2	0.11001101_2	1.0111111_2	0.10101011_2	1.1011111_2	0.10010011_2	1.1111111_2	0.10000000_2

The Futility of Testing

If AMD builds this, will it work?

A bug in this design could cost AMD hundreds of millions of dollars.

On Sunway TaihuLight (93 petaflops $= 93 \times 2^{50}$ operations per second), testing all possible cases would take

272611252374672254716119 9

$\sim 2.7 \times 10^{24}$ *years*.

The K5 FDIV Theorem (1200 lemmas)

"If p and d are $64, , 15^{+}$ floating point numbers, $d \neq 0$, and $mode$ is an IEEE rounding mode, then $\texttt{FDIV}(p, d, mode) = \mathrm{round}(p/d, mode)$."

```
(defthm FDIV-divides
  (implies (and (floating-point-numberp p 15 64)
                (floating-point-numberp d 15 64)
                (not (equal d 0))
                (rounding-modep mode))
           (equal (FDIV p d mode)
                  (round  (/ p d) mode))))
```

A Few of the 1200 Lemmas

Trunc Trunc: If $i \leq j$, then
$\mathrm{trunc}(\mathrm{trunc}(x, i), j) = \mathrm{trunc}(x, i)$.

Sticky Enough: If $mode$ is an IEEE rounding mode with size $n < i$, then
$\mathrm{round}(\mathrm{sticky}(x, i + 2), mode) = \mathrm{round}(x, mode)$.

Sticky Plus: Let $x \neq 0$, such that $\mathrm{trunc}(x, n) = x$ and $1 + e(y) < e(x)$, and $n + e(y) - e(x) < k$.
Then $\mathrm{sticky}(x + y, n) = \mathrm{sticky}(x + \mathrm{sticky}(y, k), n)$.

(Some standard hypotheses have been omitted for brevity.)

The library of floating point lemmas and the main theorem were proved with ACL2 under the direction of two ACL2 users and the designer of the FDIV algorithm.

The proofs took 9 weeks starting from Peano's axioms.

The proofs were completed before the K5 was fabricated.

9 weeks $<$ 27261125237467225471611 99 years

The library was used in subsequent proofs.

By 1997, AMD had

- built software to translate their in-house hardware design language to ACL2
- used the tool to generate ACL2 functions modeling all the elementary floating point arithmetic on the soon-to-be fabricated AMD Athlon microprocessor

- tested the ACL2 functions by running them on AMD's standard floating-point test suite ($>$ 100 million arithmetic problems) and compared the answers to AMD's design simulator

- proved the ACL2 functions compliant with the IEEE Standard

- found (and fixed) 3 design errors not exposed by the 100 million tests

Other Early Industrial Users of ACL2

- Motorola: DSP and microcode proofs
- AMD: floating-point on Opteron
- Rockwell-Collins: silicon JVM chip, AAMP7 crypo-box, Greenhills OS
- IBM: Power 4 FDIV and SQRT
- Sun Microsystems (via contract): Sun JVM class loader and byte-code verifier

Integration into Design Workflow

In 2007, Centaur Technology, Inc., challenged the ACL2 community to verify its floating-point adder:

- handles single (32-bit), double (64-bit) and extended (80-bit) additions
- pipelined to deliver 4-results per cycle
- 33,700 lines in 680 Verilog modules
- 1074 input signals (including 26 clocks) and 374 output signals

Integration into Design Workflow

In 2007, Centaur Technology, Inc., challenged the ACL2 community to verify its floating-point adder:

Done! After exposing and fixing one very rare bug.

(The bug occurred on exactly one pair of 80-bit inputs, i.e., 1 case of 2^{160} cases.)

ACL2 at Centaur Today

ACL2 is an indispensable part of the Centaur design process

Centaur FV team consists of 3 full-time employees and a couple of interns

Centaur's current family of x86-based microprocessors is called VIA Eden

Centaur has an ACL2 specification of the Eden subset of the x86

Validated by routinely running millions of tests comparing ACL2 x86 to Intel, AMD, and Centaur hardware

The ACL2 tool-chain translates the entire Eden design (700,000+ lines of Verilog) into a formal object in a few minutes

The translated model is validated by running millions of tests against Cadence NC Verilog and Synopsys VCS Verilog simulators

Two main applications:

- proving correctness of parts of the RTL design wrt behavioral spec
- proving correctness of microcode (typically no longer than several hundred lines)

All proofs are re-run nightly (on a cluster of 154 CPUs with a total of 2TB RAM)

"Bugs introduced today are found tonight and fixed tomorrow."

Centaur uses ACL2 to build custom tools for Verilog designers

Mechanized formal reasoning and theorem proving are taken for granted

Centaur's Verilog tool-chain is distributed with ACL2 and is used by Intel and Oracle

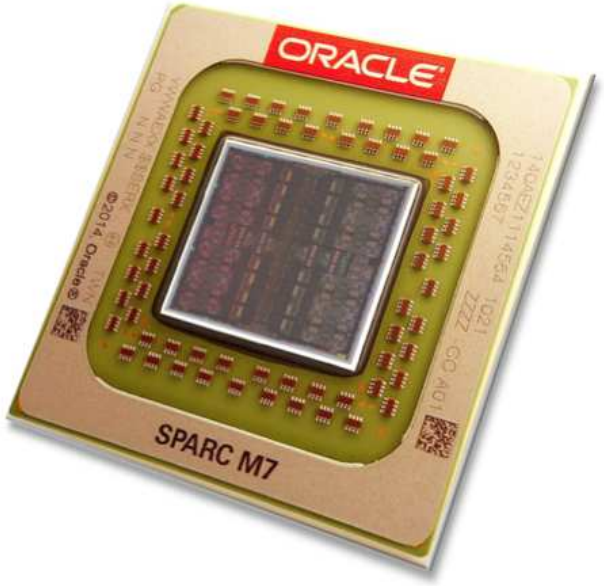
Other Ongoing Industrial Projects

- AMD (transaction protocols)
- Intel (Elliptic curve crypto: computationally surveyable proofs that $2^{255} - 19$ is prime and *Curve25519* is abelian)
- Kestrel Institute (Android apps)
- Oracle (floating point)
- Rockwell-Collins (LLVN)

Verifying Oracle's SPARC processors with ACL2

Greg Grohoski
VP, Hardware Development
Oracle Microelectronics

May 23, 2017



Summer 2014

- Wrote 1st draft of ACL2 spec for
  - IEEE 754 Standard on Floating-Point Arithmetic
  - Integer division variants
- Verified
  - Floating-point implementations wrt significand
  - Integer division implementations
- Found improvements for SPARC core
  - Reduced look-up tables by 60%
  - Improvements based on careful error analysis
  - Simplified square-root implementation

ORACLE

Summer 2015

- Validated ACL2 spec of IEEE 754 Standard against 8M vectors
- Verified all 4 floating-point implementations wrt
  - Sign, exponent, and significand
  - All exceptions
- Found more improvements for future SPARC core
  - Reduced latency further for
    - fdivs, fdivd
    - fsqrts, fsqrtd
    - idiv
- Improvements were all proven correct first and then implemented
  - No extensive 24x7 testing needed

ORACLE

Summer 2016

- Applied ACL2 to other areas
  - Temporal properties
  - Proved absence of starvation for certain instructions
- Looked at verification of complex crypto instructions
  - Verified Montgomery algorithms implementations wrt their math specification
  - Saved 25% latency in one instruction
  - Generated test vectors to exercise special cases for these instructions
- Verified and improved fault-tolerant cache-coherency protocol (like CCIX)
  - "Bake off" between ACL2 (TP) and Murphi/PReach (MC)

ORACLE

Summer 2017

- Found more improvements for integer division for future SPARC core
  - Further, significant latency reduction for integer division
  - Again, optimization was first verified, then implemented
- Verified another function using Cellular Automata Shift Registers
- Applied ACL2 to prove absence of starvation for certain instructions
  - Another “bake off” between ACL2 (TP) and SVA model checking

ORACLE

Rockwell Collins

Our Uses of ACL2 to Date

- Microcode Modeling and Proofs
- AAMP7 Information Flow Proofs (GWV Theorem)
  - NSA MILS Accreditation
- Green Hills Information Flow Proofs (GWVr2 Theorem)
  - EAL6+ Accreditation
- AAMP7 Instruction Set Modeling and Proofs
  - Interface to Eclipse-based Debugger
- MicroCryptol Runtime
- Proofs for Guard Prototype (AAMP7 code, vFAAT)
- Data Flow Logic (DFL) for C code
- LLVM Modeling and Proofs
- Other things we can't talk about…

Themes:

- ***Automated High-Level Property Verification for Low-Level Artifacts***
- ***Validation Enabled by Executable Formal Models***

X86 ISA in ACL2 (Hunt and Goel)

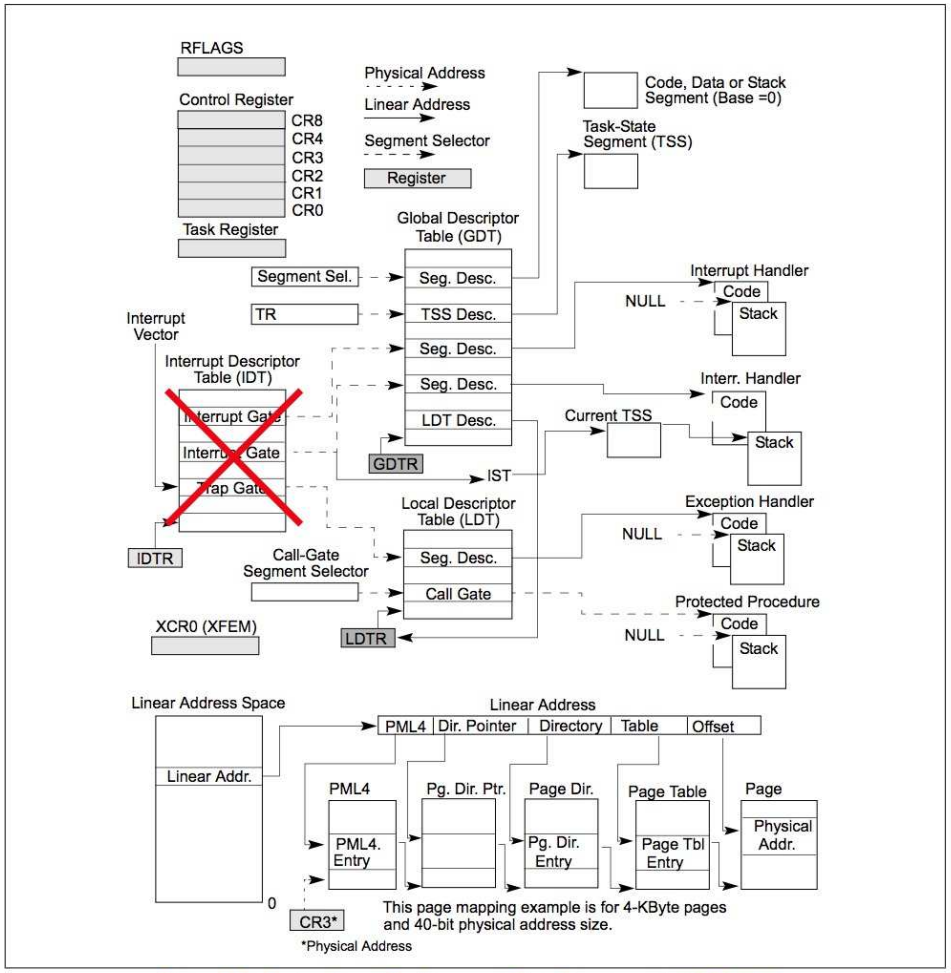


Figure 2-2. System-Level Registers and Data Structures in IA-32e Mode

x86 ISA in ACL2

Purpose of Model: Verifying binary machine code and "build-to" spec

Performance:

user level: $\sim$ 3.3 million ips

system level: $\sim$ 912,000 ips

ACL2 Support for Industrial Projects

There have been over 1000 changes to ACL2 since Centaur started using ACL2 in May, 2009. Of those, these were requested by Centaur:

Changes to Existing Features	95
New Features	44
Heuristic and Efficiency Improvements	22
Bug Fixes	72
Changes at the System Level	18
Total due to Centaur	251

Industrial Wish List

- faster execution speed of models in the logic
- more automation (esp in lemma/defn discovery)
- better ways to view large formulas
- scripting capabilities
- ability to build GUIs

Things Our Industrial Users Haven't Asked For

- quantifiers
- higher-order functions
- strong typing

General Challenges

What is the spec of given piece of software?

What does "security" mean?

Can concurrency be made tractible?

Can modern computing devices be virtualized?

Designers are faced with many problems besides functional correctness (e.g., timing, power use, area, memory, pin-out): how can formal methods contribute?

General Challenges

What is the spec of given piece of software?

What does "security" mean?

Can concurrency be made tractible?

Can modern computing devices be virtualized?

Designers are faced with many problems; how can formal methods contribute?

Because digital artifacts are formal mathematical systems these are mathematical questions.

General Challenges

What is the spec of given piece of software?

What does "security" mean?

Can concurrency be made tractible?

Can modern computing devices be virtualized?

Designers are faced with many problems; how can formal methods contribute?

Defining concepts and stating conjectures is as worthy a mathematical task as proving theorems.

What Mechanical Reasoning Requires of the User

- accepting the constraints of its formal logic/language
- precision
- ability to digest new ideas and express them in formal terms
- good proof-strategic judgement

- mastery of your own complex theories (lemmas and definitions)
- creativity, especially in connection with inductive arguments and generalization and abstraction of new concepts
- collaboration, especially with designers and other verification team members
- respect for the designers and engineers producing remarkably fast, remarkably power-efficient, and *almost perfect* designs

What It Doesn't Require of the User

- accuracy in manipulating formulas
- perfect recall of the side-conditions for use of a lemma or strategy
- deep understanding of electronics or hardware design

Why ACL2 is Successful in Industry

- that was the goal of the project
- efficient, executable logic/programming language with native verifier
- dual-use bit- and cycle-accurate models
- access to Common Lisp programming (via trust tags)
- automatic prover with "a human in the loop"

- encourages development of domain-specific automatic provers allowing *proof maintenance* as designs evolve

- rugged, well documented, free, open source form, many useful books, and a fairly unrestrictive license

- coherent user community devoted to making mechanized verification practical

- industry needs help: people with these skills are in demand, they are paid well, and become “mission critical” as soon as new bugs are exposed by their efforts

Summary

Industry is using a mechanized theorem prover to prove important theorems about critical hardware and software.

The fact that the prover supports a programming language means models have dual use: for prototyping and establishing properties via proof.

Upcoming Talk: Tomorrow I will give a tutorial on how ACL2 works and how people use it.