RACSAM
Rev. R. Acad. Cien. Serie A. Mat.
VOL. **98 (**1), 2004, pp. 181–195
Ciencias de la Computación / Computational Sciences

Some Key Research Problems in Automated Theorem Proving for Hardware and Software Verification

Matt Kaufmann and J Strother Moore

Abstract. This paper sketches the state of the art in the application of mechanical theorem provers to the verification of commercial computer hardware and software. While the paper focuses on the theorem proving system ACL2, developed by the two authors, it references much related work in formal methods. The paper is intended to satisfy the curiosity of readers interested in logic and artificial intelligence as to the role of mechanized theorem proving in hardware and software design today. In addition, it points out some of the key research topics in the area. These topics transcend any one particular theorem proving system.

Algunos problemas claves de investigación en la demostración mecánica de teoremas para la verificación de hardware y software

Resumen. En este artículo se resume el estado del arte de la aplicación de demostradores mecánicos de teoremas a la verificación de hardware y software comerciales. Aunque el trabajo se centra en el sistema de demostración de teoremas ACL2, que desarrollaron los autores, se citan trabajos en métodos formales muy relacionados. Este artículo tiene por objetivo satisfacer la curiosidad de aquellos lectores que se interesan por la lógica y la inteligencia artificial en lo relativo al papel que tiene la demostración mecánica de teoremas en el diseño de software y hardware hoy en día. Por otra parte, se señalan algunos de los temas claves de investigación en este campo. Éstos temas van más allá de las capacidades de cualquier sistema de demostración de teoremas particular.

1. Applied Formal Logic

Until the invention of the digital computer, there were few applications of formal mathematical logic outside the study of logic itself. In particular, while many logicians investigated alternative proof systems, studied the power of various logics, and formalized the foundations of mathematics, few people used formal logic and formal proofs to analyze the properties of other systems.

The lack of applications can be attributed to two considerations: (i) the very formality of formal logic detracts from its clarity as a tool of communication and understanding, and (ii) the "natural" applications of mathematical logic in the pre-digital world were in pure mathematics and there was little interest in the added value of formalization. Both of these considerations changed with the invention of the digital computer. The tedious and precise manipulation of formulas in a formal syntax can be carried out by software operating under the guidance of a user who is generally concerned more with the strategic direction of the proof. As for artifacts to study, digital computers are intended to be abstract discrete state machines

Presentado por Luis M. Laita.
Recibido: December 3, 2003. *Aceptado*: October 13, 2004.
Palabras clave / Keywords: mathematical logic, artificial intelligence, verification, formal methods
Mathematics Subject Classifications: 68T15,68Q60, 03B35,68T27,68N18,68N30.

and such machines and their software are naturally formalized in mathematical logic. Given the formal descriptions of such systems, it is then natural to reason about the systems by formal means. And with the aid of software to take care of the myriad details, the approach can be made practical. Indeed, given the cost of bugs and the complexity of modern hardware and software, these applications cry out for mechanical analysis by formal mathematical means.

To drive these points home, consider the famous Intel Pentium "FDIV" bug. In an article entitled "Circuit Flaw Causes Pentium Chip to Miscalculate, Intel Admits," the *New York Times* on November 11, 1994, reported "An elusive circuitry error is causing a chip used in millions of computers to generate inaccurate results." Three months later, on January 23, 1995, the *EE Times* reported "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." In part due to the publicity surrounding the Pentium FDIV, Advanced Micro Devices, Inc., an Intel competitor, decided to carry out a machine checked proof of the correctness of the FDIV microcode in the **AMD-K5** processor, which was slated for release in the summer of 1995. [1] The proof, constructed under the direction of this paper's authors and Tom Lynch, a member of the design team for the floating point unit, was completed 9 weeks after the effort commenced. About 1200 definitions and theorems were written by the authors and accepted, after appropriate proofs were completed by the ACL2 [27, 6] theorem prover. Among the first theorems written was one that allowed the theorem prover to rewrite $x + x$ as $2x$, which was proved by appeal to the axioms of addition and multiplication over the rationals. The last lemma proved stated that the algorithm implemented by the FDIV microcode implemented floating point division as specified by the IEEE standard [41]. The formal expression of that result, in the logic used, was as follows.

```
(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))))
```

The formula may be paraphrased as follows: If p and d are floating-point numbers with 15-bit exponents and 64-bit significands, d is non-0, and $mode$ is an IEEE rounding mode, then the output of the `FDIV` algorithm when applied to p, d, and $mode$, is the infinitely precise p/d rounded to a floating point number as specified by $mode$. Among the 1200 definitions and theorems are formalizations of the unfamiliar concepts just mentioned; these concepts are all described in the IEEE standard [41]. A description of the machine checked proof of the theorem above may be found in [34].

Machine-checked formal proofs of hardware designs are still not commonplace. But they are increasingly common. For example, register-transfer level models of the elementary floating point operations for the **AMD Athlon**^{TM} processor have been mechanically checked to be IEEE compliant [38, 39] using ACL2. These proofs were carried out before that processor was first fabricated. The initial proof attempts uncovered bugs in the design – bugs that had survived extensive testing. Intel has used HOL Light [22, 21] and other tools [1] to verify various algorithms and hardware designs for the IA-64 architecture. In addition to AMD and Intel, IBM, Sun, Rockwell and other manufacturers are advancing, applying, and supporting research into theorem proving. Among the mechanical theorem proving systems used to prove commercially interesting theorems about hardware designs, in addition to ACL2, are Coq [18], HOL [20], HOL Light [22], Isabelle [36], and PVS [37].

These systems vary quite a lot. Collectively they support several different mathematical logics and a wide variety of proof techniques and user interfaces. It is best to think of all of these systems as *proof checkers*: the user describes a particular proof and the system checks that it is correct. Proofs can be "described" a variety of ways, e.g., by giving the inference steps, by specifying tactics or strategies to try, by stating the "landmark" subgoals or lemmas to establish, etc. Often, combinations of these styles are

[1] AMD, the AMD logo, AMD Athlon, and combinations thereof, are trademarks of Advanced Micro Devices, Inc.

used within a single large proof project. But all proofs of commercially interesting theorems completed with mechanical theorem proving systems have one thing in common: they require a great deal of user expertise and effort.

The question we address in this paper is "how can the demands on the user be reduced?" More precisely, what are the key research problems in automatic theorem proving, as it applies to establishing the correctness of computer hardware and software?

2. Setting the Stage

If we are mainly interested in the correctness of computing systems, one might ask whether a mechanical theorem prover is an appropriate tool to use, since the systems being studied are always finite-state. Why not focus instead on finite state methods such as model checking [19, 14, 15], binary decision diagrams (BDDs) [12], and satisfiability solvers [44]? Such methods are surprisingly effective for a surprisingly large number of applications. Machine capacities and automatic abstraction and reduction techniques make it possible to handle system descriptions with very large state spaces. For example, the number 10^{120} is cited in [16]. This does not mean today's checkers explore 10^{120} states explicitly! It means they can sometimes verify some properties of systems with that many states, by using appropriate representation, abstraction and reduction techniques. Consider a system with 8-bits of state: the state space is of size 2^8. But suppose the property being checked is the conjunction of two properties, one of which is a function of the first three bits and the other of which is a function of the last two bits. Then the first property can be checked by visiting just 2^3 states and the second by visiting 2^2 states. So in this example a total of twelve states must be considered to verify a property of a system with 256 states.

Today's satisfiability checkers can often recognize propositional tautologies containing hundreds of thousands of propositional symbols. However, reducing finite problems to propositional form often introduces many new variable symbols and so the capacity of SAT solvers can be rapidly consumed.

The previously shown theorem about FDIV could be checked by running the microcode on about 10^{30} examples and checking it. As far as we know, all such examples would have to be checked – no reduction techniques are obvious. Assuming no reduction techniques are applicable (a good research question) and assuming that the simulator could check one example in one femtosecond (10^{-15} seconds) (the cycle time of a 1 petahertz processor) – an extremely unlikely assumption since it will take many cycles to simulate the microcode of any yet-to-be-built processor – it would still take more than 10^7 years.

While finite state tools should not be ignored or underestimated, typical hardware and software systems contain many more states than can be practically explored by exhaustive methods. Detailed application-specific formal analysis is required to invent appropriate abstraction and reduction techniques. That is the essence of the theorem proving problem.

Given our interest in theorem proving, another question might be "how does the choice of application affect the key research problems?" That is, what aspects of hardware and software verification stress theorem proving technology? In hardware and software verification one is almost constantly dealing with a mixture of inductively defined mathematical objects of essentially arbitrary size: the natural numbers, the integers, sequences, trees, graphs, etc. Thus, techniques based entirely on instantiating, chaining and substitution of equals for equals such as resolution (see, for example, the Otter System [43]) and term rewrite systems ([26]) unless those systems are augmented with some inductive capability and the techniques for dealing with the pervasive presence of integer arithmetic.

Another characteristic of hardware and software verification is the large size of the formal system descriptions. While in traditional mathematics the definitions of the relevant concepts might take a few lines or at most a few pages of formulas, in this area the relevant concepts may consume hundreds of pages. A device as complicated as a single floating point unit, much less a microprocessor or operating system, cannot be accurately described in a few well-chosen lines of formalism. When a complicated digital artifact is so-described it begs the question: does the artifact actually conform to this simple description? And to answer that question one must describe the artifact in more detail so that, ultimately, one can recognize

the actual implementation in the lowest level formal description. The proofs themselves are also large: hundreds or thousands of formally proved lemmas are assembled to reach the ultimate conclusion. Mechanically produced intermediate subgoals in these proofs can also be extremely large; in the verification of a Motorola digital signal processor design [11] some of the formulas produced (and proved) required 25 megabytes to print. (Such subgoals typically contain duplicated expressions: the propositional tautology $p \rightarrow p$ takes twice as many bytes to print as p!)

Because we are the authors of ACL2, examples from that system will be used to illustrate the research problems we identify. But the problems transcend any one particular theorem proving system.

3. A Little Background on ACL2

"ACL2" stands for "A Computational Logic for Applicative Common Lisp." It is the name of a programming language, a first-order mathematical logic based on recursive functions, and a mechanical theorem prover for that logic. We describe each component briefly. See [27] or the documentation, source code and examples at the URL [30] for details. We then cite a few example applications to establish the general applicability of the system.

3.1. The Programming Language

As a programming language, ACL2 can best be thought of as an applicative ("side-effect free" or "functional") subset of Lisp. The choice of a functional programming language as the basis for a mathematical logic of computation was McCarthy's invention [33] and was one of the original motivations behind the design of Lisp in the late 1950s and early 1960s. That Lisp is one of the oldest programming languages still in everyday use is a testament to its elegance and expressive power.

We assume the reader is familiar with the basic prefix syntax of Lisp. For example, `(+ 3 (f x y))` is the Lisp notation for $3 + f(x, y)$.

ACL2 supports the following basic data types: the complex rationals (including the rationals, integers and naturals as subsets), characters, symbols, strings, and ordered pairs. The symbols `t` and `nil` are used for the Boolean values "true" and "false." Lists are represented by ordered pairs, with the successive elements of the list in the `car` ("head") of the successive pairs found in the `cdr` ("tail"). The `cdr`-chain in lists are typically terminated with the non-pair `nil`. Ordered pairs are constructed by the function `cons`. An example constant is `'(1 "see Doc" (A . 2/3))`. This list contains three items, the natural number `1`, a string, and an ordered pair whose `car` is the symbol `A` and whose `cdr` is the rational number `2/3`. This constant could be constructed with `(cons 1 (cons "see Doc" (cons (cons 'A 2/3) nil)))`.

Systems are built by defining functions. No iteration primitives are provided; recursion is used extensively. Here is a definition of the function `len` for determining the length of a list.

```
(defun len (x)
  (if (endp x)
      0
      (+ 1 (len (cdr x)))))
```

This expression defines `len` to be a function of one argument, `x`, so that if `x` is empty (not a cons pair), its `len` is `0` and otherwise its `len` is one more than the `len` of the `cdr` of `x`. If this definition were made, then the user could type `(len '(a b c))` and the system would evaluate it and print `3`.

ACL2 includes versions of almost all of the function symbols in Common Lisp [42] that are free of side-effects and global variables and are specified in an implementation-independent fashion. We say "versions" because we restrict the domains of some of our functions so that their definitions agree with the corresponding Common Lisp program on the restricted domain. ACL2 provides a means of specifying and verifying that functions operate on their intended domains (see the discussion of "guards" in [27]).

We also support the definition of new constant symbols, new "symbol packages" (which allow users to control name spaces), and macros. The latter are like functions but produce values that are then treated as expressions. Macros thus permit the user to extend the syntax.

Because it is a functional programming language, ACL2 is executable: terms composed entirely of defined functions and constants can be reduced to constants by Lisp calculation. This is very important to many applications. For example, ACL2 models of floating point designs have been executed on millions of test cases to "validate" the models against industrial design simulation tools, before subjecting the ACL2 models to proof.

3.2. The Logic

As a mathematical logic, ACL2 may be thought of as first-order predicate calculus with equality, recursive function definition, and mathematical induction. The primitives of applicative Common Lisp are axiomatized. For example, one axiom is `(car (cons x y))` = `x` and another is `x` $\neq$ `nil` $\rightarrow$ `(if x y z)` = `y`. After axiomatizing the basic data types, including natural numbers, integers, rationals, complex rationals, ordered pairs, symbols, and strings, a representation of the ordinals up to ε_0 is introduced and the ordering relation "less than" on such ordinals is defined recursively.

The principle of mathematical induction, in ACL2, is then stated as a rule of inference that allows induction up to ε_0. To prove a conjecture by induction one must identify some ordinal-valued measure function. The induction principle permits one to assume inductive instances of the conjecture being proved, provided the instance has a smaller measure according to the chosen measure function.

A principle of definition is also provided, by which the user can extend the axioms by the addition of equations defining new function symbols. To admit a new recursive definition, the principle requires the identification of an ordinal measure function and a proof that the arguments to every recursive call decrease according to this measure. Only terminating recursive definitions can be so admitted under the definitional principle. ("Partial functions" can be axiomatized; see [32].)

While the syntax does not permit one to write explicit quantifiers ("$\forall$" and "$\exists$") the power of quantification is provided via a definition-like principle that permits the introduction of a new function symbol (a "Skolem function") whose value satisfies a given predicate if such a value exists.

A mechanism for introducing new function symbols constrained to satisfy arbitrary formulas is provided, but the mechanism requires a proof that at least one such function exists.

A "second order instantiation" derived inference rule permits one to replace the function symbols in any theorem by other function symbols, provided the latter can be proved to satisfy the constraints on the former.

For more details of the logic, see [27, 28, 29].

3.3. The Theorem Prover

The ACL2 theorem prover is an example of the so-called Boyer-Moore school of inductive theorem proving [5, 6]. ACL2 is, in fact, the successor of Nqthm [9], coded by Kaufmann and Moore (with some early help by Boyer) to support applicative Common Lisp and largely coded within that same applicative language. It is an integrated system of ad hoc proof techniques that include simplification, generalization, induction and many other techniques.

Simplification is, however, the key technique. The `typewriter` font keywords below refer to online documentation topics accessible from the ACL2 online user's manual [30]. The simplifier includes the use of (1) evaluation (i.e., the explicit computation of constants when, in the course of symbolic manipulation, certain variable-free expressions, like `(expt 2 32)`, arise), (2) conditional rewrite rules (derived from previously proved lemmas), cf. `rewrite`, (3) definitions (including recursive definitions), cf. `defun`, (4) propositional calculus (implemented both by the normalization of if-then-else expressions and the use of BDDs), cf. `bdd`, (5) a linear arithmetic decision procedure for the rationals (with extensions to help with integer problems and with non-linear problems [23]), cf. `linear-arithmetic`, (6) user-defined

equivalence and congruence relations, cf. `equivalence`, (7) user-defined and mechanically verified simplifiers, cf. `meta`, (8) a user-extensible type system, cf. `type-prescription`, (9) forward chaining, cf. `forward-chaining`, (10) an interactive loop for entering proof commands, cf. `proof-checker`, and (11) various means to control and monitor these features including heuristics, interactive features, and user-supplied functional programs.

The induction heuristic, cf. `induction`, automatically selects an induction based on the recursive patterns used by the functions appearing in the conjecture or a user-supplied hint. It may combine various patterns to derive a "new" one considered more suitable for the particular conjecture. The chosen induction scheme is then applied to the conjecture to produce a set of new subgoals, typically including one or more base cases and induction steps. The induction steps typically contain hypotheses identifying a particular non-base case, one or more instances of the conjecture to serve as induction hypotheses, and, as the conclusion, the conjecture being proved. The system attempts to select an induction scheme that will provide induction hypotheses that are useful when the function applications in the conclusion are expanded under the case analysis provided. Thus, while induction is often considered the system's *forte* most of the interesting work, even for inductive proofs, occurs in the simplifier.

All of the system's proof techniques are sensitive to the database of previously proved rules and, if any, user-supplied hints. By proving appropriate theorems (and tagging them in pragmatic ways) it is possible to make the system expand functions in new ways, replace one term by another in certain contexts, consider unusual inductions, add new known inequalities to the linear arithmetic procedure, restrict generalizations, etc.

3.4. A Few Applications

ACL2 has been applied to a wide range of commercially interesting verification problems. We recommend visiting the ACL2 home page [30] and inspecting the links on Tours, Demo, Books and Papers, and for the most current work, The Workshops and Related Meetings.

Among the items there documented are floating point proofs for AMD and IBM hardware, including the **AMD-K5** processor, **AMD Athlon**^{TM} processor, and IBM Power4^{TM} processor, the verification that a pipelined microarchitecture for a Motorola digital signal processor implements a certain microcode engine when a certain predicate relating to the absence of pipeline hazards approves of the microcode [11], the verification of microarchitectures for Rockwell avionics microprocessors, the verification of security properties for an IBM secure co-processor, the verification of a post-compiler checker for an on-board railroad control system for Union Switch and Signal, the verification of a proof-checker used by Otter to certify its proofs, and the verification of properties of Java bytecode and the Java Virtual Machine.

Many of these projects require the definition within ACL2 of one or more interpreters for other languages (e.g., the JVM is formalized as an interpreter [35, 31]). Proving theorems about such interpreters, especially establishing relationships between such interpreters, is a key aspect of system verification.

4. Key Research Problems

We tend to give relatively few references in this section. Most of our references are of historic note only. Anybody undertaking research on these topics should search the web (at least) for the latest developments.

4.1. Invention of Lemmas and New Concepts

The fundamental problem in automatic theorem proving is, of course, the theorem proving problem itself: how can a given formula be constructed from the axioms by the rules of inference? In a logic supporting induction this is crucial. The top-level theorems in which we are interested are typically of insufficient generality to permit inductive proofs. Instead, inductively provable lemmas must be formulated and proved.

For example, given the recursive definition of list concatenation ("append") `(defun app (x y) (if (endp x) y (cons (car x) (app (cdr x) y))))` try to prove `(equal (app (app a a) a) (app a (app a a)))`. To prove this theorem one must distinguish some of the variable symbols. The obvious lemma is the associativity of `app` but a weaker lemma will suffice. How do we find such lemmas?

More often, new concepts must be invented to prove theorems by induction. For example, suppose `isort` is defined to be the insertion sort function. Then it is a theorem that `(isort (isort a))` is `(isort a)`.[2] However, the proof that springs to mind is that `isort` is stable, i.e., if `x` is ordered then `(isort x)` is `x`, and that `(isort a)` is ordered. These two lemmas require the introduction of the concept of "ordered." We call "ordered" a "eureka" concept. Why did it spring to mind? How do we know how to define it?

To see how common the problem is, take any interesting mechanically checked theorem in an inductive setting, and compare the number of defined concepts used in the statement of the theorem (including such "ancestral uses" as when a concept is used in a definition of a concept used in the theorem) with the total number of defined concepts developed in the proof. If they are not the same, a eureka concept was probably discovered.

This problem is widely recognized and has received some study. One might start by considering [5, 2, 13, 24, 25]. The problem is closely related to the invention or completion of inductive invariants to prove code correct, the characterization of the reachable states of a system, and abstraction/refinement methods. See the work on abstract interpretation, which was first described in [17]. There is enormous interest in these problems because of the state explosion problem in model checking. See for example the SLAM project [3].

4.2. Examples and Counterexamples

How can a machine productively use examples to guide the discovery of a proof? How can a machine productively use failed proofs to guide the discovery of counterexamples? When we (the authors) try to prove formulas we are not sure are theorems, we often spend considerable time looking for counterexamples. When the proof attempt is stymied, i.e., when it seems impossible to show that a subgoal is valid, we may switch to finding a counterexample. Often, just when we think we have a counterexample, careful testing shows that the formula is still true under that assignment. This often exposes a subtle relation between the hypotheses that, once recognized, can be exploited to move the proof forward. On the other hand, when we find a counterexample for a subgoal it may mean we are pursuing a bad proof strategy or it may mean our main goal is not a theorem. In the first case, the example helps guide subsequent backtracking and search; in the second case, the counterexample often helps us reformulate the main theorem.

There has been enormous investment by industry in test suites for hardware and software designs. Those test suites could, conceivably, provide examples to help guide formal proofs. Let us say a formula is "plausible" if it evaluates to true on many examples. Clearly, before trying to prove a formula, one might first check that it is plausible. A more sophisticated definition of plausibility is actually required. The non-theorem "if `x` is `123` then `x` is even" is plausible over an arbitrarily large number of examples – as long as `123` is not among them! To test plausibility one must have the ability to generate or find test cases that satisfy the hypotheses so that one can test the conclusion. But even so, how can this guide search? Suppose you are trying to prove `(Q x)` and you have two lemmas, one that says `(P x)` implies `(Q x)` and another that says `(R x)` implies `(Q x)`. To apply such a lemma you must "backchain" to establish its hypothesis. But if your tests establish that `(P x)` is not plausible (i.e., one of your examples falsifies `(P x)`) that branch of the search space can be pruned. SAT solvers make such use of propositional counterexamples.

Another common use of examples in ordinary technical work is related to the comprehension and invention of ideas. Generalizing from examples seems to be an exceptionally well-developed skill among many

[2] In all of our examples, we ignore "type-like" hypotheses. The `isort` identity to which we just alluded is in fact an identity in ACL2, for all `a`. But some readers may be more comfortable restricting `a` to proper lists of sortable objects, i.e., whatever objects `isort` can compare; the essential problems are the same.

people. How many times have conference speakers, for example, presented a new idea or methodology by using it to solve an example problem? How many times have professors presented "proofs by example"? Often, when confronted with the choice between a formal description of a new concept and a few examples of it, we choose to look at the examples early.

Probably because of this phenomenon, some people prefer to think about the creation of new lemmas and concepts by looking at examples. For example, define `ffib` as follows.

```
(defun ffib (i j k)
  (if (zp i)
      j
      (if (equal i 1)
          k
          (ffib (- i 1) k (+ j k)))))
```

Then `(ffib n 1 1)` is the n^{th} Fibonacci number, as that concept is traditionally defined,

```
(defun fib (i)
  (if (zp i)
      1
      (if (equal i 1)
          1
          (+ (fib (- i 1)) (fib (- i 2))))))).
```

Prove `(ffib n 1 1)` = `(fib n)`. To do so requires a generalization. Find it. Some people find it helpful to run a few examples of `(ffib n i j)` with `i` and `j` different from `1`. Often, after creating a purported generalization, people use examples to "confirm" its plausibility as a theorem.

4.3. Analogy, Learning, and Data Mining

"Proof by analogy" is a commonly used phrase in mathematical parlance. Often it means that the proof of the indicated lemmas follows some general pattern exemplified by the earlier proof. More often it means "Surely a proof can be constructed along those lines (but I haven't actually done it)." There has been some work on this [10].

A user of ACL2 frequently has the sense that the system is repeating old proofs with new symbols. This can be a sign that some earlier lemma was not sufficiently general. But often it is extremely tedious to produce the sufficiently general result. Indeed, people seem often to look for the precise general statement of a result by proving special cases (here we may see the important role that examples play in human reasoning). If the proof engine can be made to prove rapidly all the special cases one ever needs, then it is often more efficient never to state and prove the general result. However, it would be nice to provide an interface so that the user could signify this is what is happening and so guide ACL2 to follow the old proof even though ACL2's database might now suggest completely different attacks on the problem.

In a similar vein, can a theorem prover not recognize when an analogy might be appropriate? Or better yet, can it not learn from past behavior when to revert to a given proof pattern or plan?

Finally, note that users are often distributed and each has his or her database of definitions and lemmas. Therefore it would be extremely helpful to build, first, a tool that would allow a user to ask a search engine "has anybody ever proved that `gcd` distributes over multiplication?" and get an answer like "User smith defined the function `gd` which is reminiscent of your `gcd` and proved that it distributes over multiplication of rationals. The definition of `gd` and the theorem follow." Once such an interactive search engine were in use, it would be challenging to try to get the theorem prover to try to search for and speculatively import such results so proofs could draw on the larger body of work of the whole community.

4.4. An Open Architecture

How can we design a theorem prover that allows the user easy but sound access to the inner workings of the system? Experienced ACL2 users occasionally want to reconfigure ACL2, e.g., so that it skips some proof technique. Examples include the desire to skip ACL2's "pre-processing" step, to skip or delay the use of `IF`-normalization, or to rewrite the various parts of a formula in a different order. The authors of ACL2 have provided a plethora of hints that allow the selective disabling of certain commonly problematic heuristics. But the result feels somewhat ad hoc. It might be better if the system were designed in a way that the user could easily program it, without worrying about soundness. The use of tactics and tacticals in HOL [20] and Nqthm's metafunctions [7] were suggestive — indeed, ACL2's "proof-checker" utility has a form of tactics and tacticals, and ACL2 has a sophisticated capability for metatheoretic reasoning — but we seek an entirely open architecture that would allow easy seamless reconfiguration of the basic proof engine. For example, ACL2 users have expressed an interest in more control of the distribution of calls of the function `if`, the order of rewriting literals in a clause, and the handling of cases.

A suitably ambitious attempt at providing an open architecture might be the SAL [40], which includes the design of a language for describing the combination of proof techniques. Another valuable resource is the "QED Quo Pro" (www.qpq.org) project.

4.5. Parallel and Collaborative Theorem Provers

We have earlier mentioned (subsection 4.3.) the idea of a prover speculatively drawing on the results proved by other users of the system. Still more challenging is drawing on the capabilities of other theorem provers.

How can we parallelize a single theorem prover? The theorem proving task is embarrassingly parallel. Often, ACL2 spawns thousands of subgoals, each of which must be proved. Since each is attacked in the same logical theory, each could be farmed out to a copy of ACL2 loaded with the current theory. Interprocess communication could be minimized by having each processor initially loaded with the current database. Then, all that would be required to spawn a new subgoal would be to transmit its formula to a free processor and await a brief success or failure signal. Such parallelization is currently being investigated by our group in at the University of Texas at Austin. But finer grained parallelization is also possible. Why not try multiple induction schema simultaneously? Why not try alternative rewrite theories or other simplification strategies simultaneously? Why not simultaneously attempt to relieve all the hypotheses of a lemma when backchaining? Why not rewrite all argument subexpressions of an expression simultaneously? The answer to many of these questions is "because you would almost certainly swamp the communication network and tie up many processors with pointless work when you could save them for necessary work." But the judicious non-explosive use of parallelism is appealing and, we believe, practical in the theorem proving context.

How can several theorem provers and/or decision procedures be combined so that they can collectively prove theorems more automatically than any one of them could prove in isolation? There is much work in the combination of theorem proving with model checking and with SAT solving. Experience [8] shows that combining powerful heuristic theorem provers with decision procedures is very difficult. The problem is that it is impossible to know what problems can be solved by the decision procedure, so most of the time it reports failure. Thus, the time it takes to interact with another tool is often dominated by the time it takes to convert a problem into the representation used by the "foreign" tool. This argues for the design of tools that sacrifice internal stand-alone efficiency for ease of communication. But no one would use such a tool except as part of a larger collaborative system (if a more efficient stand-alone tool were known to be sufficient).

Even more problematic is the differences between the logics supported by different theorem provers. For example, PVS and ACL2 have different syntax. To use PVS from within ACL2, we would have to build a translation tool that could translate formula Φ of ACL2 into some formula Φ' of PVS such that if Φ' is a theorem of PVS then Φ is a theorem of ACL2. To make this precise, the theory (i.e., definitions, lemmas, etc.) must also be exported as well. Because of the differences in the two logics, such translators are extremely difficult to define correctly and will almost certainly be only partial. The likelihood of

success by the foreign system is low, thus increasing the importance of fast translation and well-considered submissions. But these issues have the effect of opening up the "black box," i.e., to use PVS to help ACL2, ACL2 would benefit from being able quickly (1) to predict when PVS will succeed and (2) to generate the appropriate proof scripts for success.

This line of reasoning points to meta-tools with which complex system expectations, behaviors and capabilities can be described and learned. Note also that the many of the problems in this section on collaborative theorem proving strategies might impact the previous section on learning and data mining. For example, perhaps an ACL2 user could make use of the fact that a PVS user has proved a certain lemma.

4.6. User Interface and Interactive Steering

Until the artificial intelligence problem is solved, human interaction will be important in theorem proving. In fact, we do not think this need be a major problem. Designers and programmers are much more willing to explain why their designs work than to be bothered by a lot of mindless details (unless a bug is indicated). How can a theorem prover present itself to its users (who are possibly distributed and working collaboratively) so that it (1) is often effective while being fully automatic, (2) presents its users with a manageable amount of information, (3) presents its users with sufficient information to allow them to help it when it seems to be stymied, and (4) accepts real-time interactive advice from its users without forcing them to take over low level control of the entire proof effort?

A related issue is the challenge of integrating automated or interactive reasoning engines in more routinely-used tools such as compilers, CAD systems, databases, etc. Here the interface challenge is to cast the user-supplied guidance into a form (e.g., type declarations or runtime assertions) acceptable to the user of the high-level tool and to limit the expressible problems so that the reasoning engine succeeds often enough to be perceived as adding value.

4.7. Education

Until the artificial intelligence problem is solved, the hardware and software industry will need people trained in the use of mechanized reasoning tools. How can theorem proving be taught effectively at the undergraduate level?

It is insufficient simply to teach an undergraduate course on predicate calculus or logic, sets and functions. Experience with using a mechanized theorem prover is necessary, because it drills the student in the details of formalism and symbolic manipulation. But there is a gap between tools designed for beginners and tools useful to experts. The former are typically dead ends and teach the student only that mathematical logic is tedious and impractical. The latter (in which we would include most of the "industrial strength" theorem provers mentioned in this article) have large, complex mathematical logics, a plethora of user controls, and very steep learning curves to reach moderate skill levels.

Clearly, we need industrial strength theorem provers supporting user profiles, active advice and debugging, with simple but layered error messages and documentation, which provide tutorial, beginner, and intermediate levels consistent with developing the skill-set necessary to become a professional.

In addition, we need course material suitable for undergraduates that can take them from a rudimentary understanding of formal logic to a "self-sufficient" level of competence with some industrial-strength formal reasoning tool. By self-sufficient competence we mean that the successful student, interested in going forward, ought to be able to do so using the documentation and help facilities available to the research community using the tool. An example course, oriented around the Isabelle theorem proving system, has been developed by David Basin (CS Department, ETH Zurich) and is available from his web page.

4.8. A Verified Theorem Prover

Can you prove a theorem prover correct? Obviously, yes. A propositional tautology checker was proved correct in [6] in 1978. Can you prove a theorem prover correct with a mechanical theorem prover that

you trust more than the theorem prover being verified? That is harder. Naively, using a theorem prover to prove itself correct seems to catch you in the Liar's Paradox. But this need not be so. If a theorem prover optionally produces a formal proof as its output, then a very complicated AI based heuristic theorem prover might be used to produce a formal proof of its own correctness and then that formal proof might be checked by simple, trusted proof checker. Once such a proof is successfully checked, the option of generating the formal proof can be turned off in the knowledge that the system is correct.

Our feeling is that this particular attack will not succeed. One reason is that the generated proof is probably too large for a simple proof checker. For example, what if the constant 2^{31} is used (as it is in ACL2)? If the proof checker is sufficiently simple, natural numbers are represented in the Peano style, e.g., 3 is $s(s(s(0)))$. But then it is probably impractical to represent and manipulate 2^{28}. But representing constants more efficiently changes the logic and the simple proof checker. Would you still trust the more efficient one? Such thinking leads to the idea of a hierarchy of theorem provers, each more powerful than the previous one, each producing proofs that are much more succinct, and each by itself and checked by the previous one.

We believe this is viable but raises another question. Do you trust the implementation of the lowest level theorem prover? A grand challenge would be to build a "verified stack" [4] – a mechanically verified microprocessor design hosting mechanically verified software on it – with the top-most software being a verified theorem prover of practical value. Once built, use the fabricated design – as well as other platforms – to check the proof of its correctness

5. Some Concluding Advice

We believe that substantial progress is possible on many of the problems identified above. We furthermore believe that such progress would greatly ease the burden on the users of mechanized theorem proving tools applied in the service of hardware and software verification. Despite our focus on ACL2, many theorem proving systems could benefit. However, experience has taught us some lessons worth passing on here.

If you wish to pursue one of the research problems described in the preceding section, first, do an up-to-date literature search, especially an online search. Because these problems are of such importance in machine-assisted reasoning, there is active research. As noted early, our references tend to be of mainly historic or archival interest and should not be taken as representative of the current state of the art.

Second, choose any one of the "industrial strength" theorem proving tools available and learn to use it well. These tools have been well enough engineered so that many routine but complicated problems can be solved automatically. It is of little value for you to solve a problem that the moderately experienced user of a practical tool never encounters.

Then, choose a hard, commercially interesting proof project and solve it with the existing tool, to gain and demonstrate your mastery of the tool and to establish a benchmark against which to assess your contributions. Try to document where you spent your time in the proof effort.

Next, develop your ideas for solving your chosen problem. Keep in mind the actual needs you experienced while tackling the commercially interesting application with an existing proof engine. To develop your ideas you may find it necessary to do "toy" applications and to write a "toy" theorem prover. But do not "declare victory" until you have incorporated your heuristics and methods into an industrial strength proof engine.

When you finish your extension, use the tool you have developed to solve a harder variant of the original problem. Was it easier? Be prepared to admit that your new technique contributed less savings than your blossoming familiarity with the problems! Indeed, perhaps the second pass at a hard problem will expose some underlying patterns you can exploit to make the task still easier.

In summary,

- avoid doing theorem proving research in the abstract;

- show that your work can improve an industrial strength theorem prover by becoming an expert on at least one such system;
- do not content yourself with a demonstration on a toy problem (but we highly recommend the study of toy problems and the iteration of a problem solving technique on an increasingly sophisticated sequence of such toys);
- use the tool you build to prove something new and challenging; and
- document your effort.

These recommendations are not meant to be discouraging. But the research problems highlighted here are still problems – after many decades of research and engineering – because they are hard. The "industrial strength" theorem provers are as useful as they are because they provide logical features found *necessary* to cope with industrial sized problems and they have been well enough engineered to permit the features to be used. Some of these problems are simply hard in any setting at all. But many of them are hard because they must be solved in the context of logical richness and engineering excellence.

Acknowledgments

We thank the attendees of the weekly ACL2 seminar in Austin, Texas for feedback that was useful in the development of this paper.

References

[1] M. Aagaard, R. Jones, T. Melham, J. O'Leary, and C-J. Seger. A methodology for large-scale hardware verification. In W. Hunt and S. Johnson, editors, *Formal Methods in Computer-Aided Design (FM-CAD) 2000*, volume 1954 of *Lecture Notes in Computer Science*, pages 263–282. Springer-Verlag, November 2000.

[2] R. Aubin. Some generalization heuristics in proofs by induction. In G. Huet and G. Kahn, editors, *Actes du Colloque Construction: Amélioration et vérification de Programmes*. Institut de recherche d'informatique et d'automatique, 1975.

[3] T. Ball and S. K. Rajamani. The SLAM project: debugging system software via static analysis. *SIGPLAN Notices: Conference Record of POPL 2002*, 37(1):1–3, January 2002.

[4] W.R. Bevier, W.A. Hunt, J S. Moore, and W.D. Young. Special issue on system verification. *Journal of Automated Reasoning*, 5(4):409–530, 1989.

[5] R. S. Boyer and J S. Moore. Proving theorems about pure lisp functions. *JACM*, 22(1):129–144, 1975.

[6] R. S. Boyer and J S. Moore. *A Computational Logic*. Academic Press, New York, 1979.

[7] R. S. Boyer and J S. Moore. Metafunctions: Proving them correct and using them efficiently as new proof procedures. In *The Correctness Problem in Computer Science*. Academic Press, London, 1981.

[8] R. S. Boyer and J S. Moore. Integrating decision procedures into heuristic theorem provers: A case study of linear arithmetic. In *Machine Intelligence 11*, pages 83–124. Oxford University Press, 1988.

[9] R. S. Boyer and J S. Moore. *A Computational Logic Handbook, Second Edition*. Academic Press, New York, 1997.

[10] B. Brock, S. Cooper, and W. Pierce. Analogical reasoning and proof discovery. In E. Lusk and R. Overbeek, editors, *Proceedings of the 9th International Conference on Automated Deduction*, pages 454–468. Springer-Verlag, 1988.

[11] B. Brock and W. A. Hunt, Jr. Formal analysis of the motorola CAP DSP. In *Industrial-Strength Formal Methods*. Springer-Verlag, 1999.

[12] R. E. Bryant. Symbolic Boolean manipulation with ordered binary decision diagrams. *ACM Computing Surveys*, 1992.

[13] A. Bundy. The use of explicit plans to guide proofs. In E. Lusk and R. Overbeek, editors, *Proceedings of CADE-9*, volume 310 of *Lecture Notes in Computer Science*, pages 111–120. Springer-Verlag, 1988.

[14] E. M. Clarke and E. A. Emerson. The design and synthesis of synchronization skeletons using temporal logic. In *Proceedings of the Workshop on Logics of Programs, IBM Watson Research Center, Yorktown Heights, New York*, volume 131 of *Lecture Notes in Computer Science*, pages 52–71. Springer-Verlag, 1981.

[15] E. M. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, 1999.

[16] E.M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Progress on the state explosion problem in model checking. *Lecture Notes in Computer Science*, 2000:176–194, 2001.

[17] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Conference Record of the Fourth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 238–252, Los Angeles, California, 1977. ACM Press, New York, NY.

[18] G. Dowek, A. Felty, H. Herbelin, G. Huet, C. Paulin, and B. Werner. The Coq proof assistant user's guide, Version 5.6. Technical Report TR 134, INRIA, December 1991.

[19] E. A. Emerson. Branching time temporal logic and the design of correct concurrent programs. Phd thesis, Division of Applied Sciences, Harvard University, 1981.

[20] M. Gordon and T. Melham. *Introduction to HOL: A Theorem Proving Environment for Higher Order Logic*. Cambridge University Press, 1993.

[21] J. Harrison. A machine-checked theory of floating point arithmetic. In Yves Bertot, Gilles Dowek, André Hirschowitz, Christine Paulin, and Laurent Théry, editors, *12th International Conference on Theorem Proving in Higher Order Logics*, pages 113–130, Nice, France, 1999.

[22] J. Harrison. The HOL light manual (1.1). Technical Report http://www.cl.cam.ac.uk/users/jrh/hol-light/, University of Cambridge Computer Laboratory, New Museums Site, Pembroke Street, Cambridge CB2 3QG, England, April 2000.

[23] W. Hunt, R. Krug, and J S. Moore. The addition of non-linear arithmetic to ACL2. In D. Geist, editor, *Proceedings of CHARME 2003*, volume 2860 of *Lecture Notes in Computer Science*, pages 319–333. Springer Verlag, 2003.

[24] A. Ireland. Productive use of failure in inductive proof. *Journal of Automated Reasoning*, 16(1-2):79–111, 1996.

[25] D. Kapur and M. Subramaniam. Lemma discovery in automating induction. In M. McRobbie and J. Slaney, editors, *Proceedings of CADE-13*, volume 1104 of *Lecture Notes in Computer Science*, pages 538–552. Springer Verlag, 1996.

[26] D. Kapur and H. Zhang. An overview of rewrite rule laboratory (RRL). *J. of Computer and Mathematics with Applications*, 29(2):91–114, 1995.

[27] M. Kaufmann, P. Manolios, and J S. Moore. *Computer-Aided Reasoning: An Approach*. Kluwer Academic Press, Boston, MA., 2000.

[28] M. Kaufmann and J S. Moore. A precise description of the ACL2 logic. In `http://www.cs.utexas.edu/users/moore/publications/km97a.ps.gz`. Department of Computer Sciences, University of Texas at Austin, 1997.

[29] M. Kaufmann and J S. Moore. Structured theory development for a mechanized logic. *Journal of Automated Reasoning*, 26(2):161–203, 2001.

[30] M. Kaufmann and J S. Moore. The ACL2 home page. In `http://www.cs.utexas.edu/users/moore/acl2/`. Department of Computer Sciences, University of Texas at Austin, 2004.

[31] H. Liu and J S. Moore. Executable JVM model for analytical reasoning: A study. In *Workshop on Interpreters, Virtual Machines and Emulators 2003 (IVME '03)*, San Diego, CA, June 2003. ACM SIGPLAN.

[32] P. Manolios and J S. Moore. Partial functions in ACL2. *Journal of Automated Reasoning*, 31(2):107–127, 2003.

[33] J. McCarthy. Towards a mathematical science of computation. In *Proceedings of the Information Processing Cong. 62*, pages 21–28, Munich, West Germany, August 1962. North-Holland.

[34] J S. Moore, T. Lynch, and M. Kaufmann. A mechanically checked proof of the correctness of the kernel of the AMD5K86 floating point division algorithm. *IEEE Transactions on Computers*, 47(9):913–926, September 1998.

[35] J S. Moore and G. Porter. The Apprentice challenge. *ACM TOPLAS*, 24(3):1–24, May 2002.

[36] T. Nipkow and L. C. Paulson. Isabelle-91. In D. Kapur, editor, *Proceedings of the 11th International Conference on Automated Deduction*, pages 673–676, Heidelberg, 1992. Springer-Verlag LNAI 607. System abstract.

[37] S. Owre, J. Rushby, and N. Shankar. PVS: A prototype verification system. In D. Kapur, editor, *11th International Conference on Automated Deduction (CADE)*, pages 748–752, Heidelberg, June 1992. Lecture Notes in Artificial Intelligence, Vol 607, Springer-Verlag.

[38] D. Russinoff. A mechanically checked proof of IEEE compliance of a register-transfer-level specification of the AMD-K7 floating-point multiplication, division, and square root instructions. *London Mathematical Society Journal of Computation and Mathematics*, 1:148–200, December 1998. http://-www.onr.com/user/russ/david/k7-div-sqrt.html.

[39] D. M. Russinoff and A. Flatau. Rtl verification: A floating-point multiplier. In M. Kaufmann, P. Manolios, and J S. Moore, editors, *Computer-Aided Reasoning: ACL2 Case Studies*, pages 201–232, Boston, MA., 2000. Kluwer Academic Press.

[40] N. Shankar. Symbolic analysis of transition systems. In Yuri Gurevich, Phillip W. Kutter, Martin Odersky, and Lothar Thiele, editors, *Abstract State Machines: Theory and Applications (ASM 2000)*, number 1912 in Lecture Notes in Computer Science, pages 287–302, Monte Verità, Switzerland, mar 2000. Springer-Verlag.

[41] Standards Committee of the IEEE Computer Society. IEEE standard for binary floating-point arithmetic. Technical Report IEEE Std. 754-1985, IEEE, 345 East 47th Street, New York, NY 10017, 1985.

[42] G. L. Steele, Jr. *Common Lisp The Language, Second Edition*. Digital Press, 30 North Avenue, Burlington, MA. 01803, 1990.

[43] L. Wos and G. Piper. *A Fascinating Country in the World of Computing: Your Guide to Automated Reasoning*. World Scientific, 1999.

[44] L. Zhang and S. Malik. The quest for efficient boolean satisfiablity solvers. In Andrei Voronkov, editor, *Automated Deduction - CADE-18, 18th International Conference on Automated Deduction, Copenhagen, Denmark, July 27-30, 2002, Proceedings*, volume 2392 of *Lecture Notes in Computer Science*, pages 295–313. Springer, 2002.

Matt Kaufmann
Advanced Micro Devices, Inc.
5900 East Ben White Blvd.
Austin, TX 78741, USA

J Strother Moore
Department of Computer Sciences
University of Texas at Austin
Austin, TX 78712 USA
http://www.cs.utexas.edu/users/moore