An Industrial Strength Theorem Prover for a Logic Based on Common Lisp

Matt Kaufmann[*†] and J Strother Moore[‡]

Personal use of this material is permitted. However, permission to reprint/republish this material for advertising or promotional purposes or for creating new collective works for resale or redistribution to servers or lists, or to reuse any copyrighted component of this work in other works must be obtained from the IEEE.[1]

Abstract—ACL2 is a re-implemented extended version of Boyer and Moore's Nqthm and Kaufmann's Pc-Nqthm, intended for large scale verification projects. This paper deals primarily with how we scaled up Nqthm's logic to an "industrial strength" programming language — namely, a large applicative subset of Common Lisp — while preserving the use of total functions within the logic. This makes it possible to run formal models efficiently while keeping the logic simple. We enumerate many other important features of ACL2 and we briefly summarize two industrial applications: a model of the Motorola CAP digital signal processing chip and the proof of the correctness of the kernel of the floating point division algorithm on the AMD5$_K$86 microprocessor by Advanced Micro Devices, Inc.

Index terms—formal verification, automatic theorem proving, computational logic, partial functions, total functions, type checking, microcode verification, floating point division, digital signal processing

1. Introduction

FORMAL VERIFICATION is the use of mathematical techniques to verify properties of a system description. A particular style of formal verification that has shown considerable promise in recent years is the use of general-purpose automated reasoning systems to model systems and prove properties of them. Every such reasoning system requires considerable assistance from the user, which makes it important that the system provide convenient ways for the user to interact with it.

One state-of-the-art general-purpose automated reasoning system is ACL2: "A Computational Logic for Applicative Common Lisp." A number of automated reasoning systems now exist, as we discuss below (Subsection 1.1). In this paper we describe ACL2's offerings to the user for convenient "industrial-strength" use. We begin in Section 2 with a history of the ACL2 project. Next, Section 3 describes the logic supported by ACL2, which has been designed for convenient specification and verification. Section 4 discusses *guards*, which connect ACL2 to efficient execution in Common Lisp and provide a powerful specification capability. We illustration the role of guards in Section 5. In Section 6 we discuss other important features of ACL2. In Section 7 we present two industrial applications. We conclude with Section 8.

1.1. Brief Comparison with Other Theorem Provers

As we mentioned above, there are many other automated reasoning systems besides ACL2 and its ancestors. Although it is beyond the scope of this paper to survey the field or provide descriptions of other systems, we say a few words here in order to provide some context for our work.

Active research continues in automated reasoning in a number of areas. Here is an incomplete list. In each case we give one or two representative systems. Certainly the areas below contain considerable overlap.

- Provers providing strong support for specification of computing systems (see below)
- CTL model checkers [29, 11]
- Geometry provers [13]
- First-order provers [28]
- Classical Mathematics [21], [41]

---

[*] *The work reported here was performed while the authors were employed at Computational Logic, Inc.*

[†] *Matt Kaufmann is with Motorola @ Lakewood, P.O. Box 6000, MD F52, Austin, TX 78762. E-mail: Matt_Kaufmann@email.mot.com.*

[‡] *J Moore is with the Department of Computer Sciences, University of Texas at Austin, Austin, TX 78712. E-mail: Moore@cs.utexas.edu.*

[1] *Manuscript received Oct. 25, 1996; revised Mar. 31, 1997*
Recommended for acceptance by C. Heitmeyer and S.R. Faulk
For information on obtaining reprints of this article, please send e-mail to: transse@computer.org, and reference IEEECS Log Number 104892.0

- Constructive Mathematics [15, 16]
- Provers with symbolic computation engines [14]
- Meta-theoretic systems [34]

Provers in the first category are distinguished by the *convenience* they offer for specifying computing systems. Cases could be made that each prover in the first category has capabilities in most of the other categories; conversely, some provers in the other categories could be placed in this one.

The first category may be subdivided as follows.

- Higher-order tactic-based provers, e.g., HOL [20]
- Higher-order heavily-automated provers, e.g., PVS [18]
- First-order heavily-automated provers, e.g., ACL2 and Nqthm
- Provers integrated into program verification systems, e.g., Never/EVES [17]

Again, space does not permit detailed comparisons here. Bill Young's paper [42] in this Special Issue compares PVS and ACL2 on a particular example. ACL2's ancestral system, Nqthm, is compared to NuPRL in [3]. It is extremely difficult to compare two general-purpose theorem provers at least in part because experienced users can dramatically affect system behavior by proper formulation of the problems.

That said, other systems cited above support logics more powerful than that of ACL2. On the other hand, ACL2's theorem prover encourages more reliance by the user on the system's automatic aspects. For users happy with an essentially quantifier-free, first-order logic, we believe that ACL2 offers more overall *convenience* for the type of reasoning required to model and prove properties of digital computing systems. In addition to the sophisticated inference engine it provides, ACL2 provides extremely efficient evaluation, allowing formal models often to serve as simulators for the systems described. This, in turn, provides some immediate proof-independent payoff, e.g., requirements testing and code development. Additional reasons for ACL2's convenience can be broadly lumped into the "proof engineering" considerations discussed in Section 6.

See the URL http://www-formal.stanford.edu/clt/ARS/-ars-db.html for a data base of automated reasoning systems, including brief descriptions and links to the home pages of the systems mentioned and many more.

2. History

ACL2 is a direct descendant of the Boyer-Moore system, Nqthm [8, 9], and its interactive enhancement, Pc-Nqthm [23]. See [7] for an introduction to the two ancestral systems, including a reasonably large set of references for accomplishments using the systems. A few particular successes are described in [4, 5, 10, 22, 32, 26, 36, 38]. A tutorial introduction to the systems may be found in [24].

Like Nqthm, ACL2 supports a Lisp-like, first-order, quantifier-free mathematical logic based on recursively defined total functions. Experience with the earlier systems supports the claim that such a logic is sufficiently expressive to permit one to address deep mathematical problems and realistic verification projects. The fact that the Nqthm logic is executable is also an important asset when using it to model hardware and software systems: the models can be executed as a means of corroborating their accuracy. Consider for example [2] where an Nqthm model of the MC68020 is corroborated against a fabricated chip by running 30,000 test vectors through the Nqthm model.

Some of the largest formal verification projects done so far have been carried out with Nqthm. We cite explicitly the CLI short stack [4], the design and fabrication of the FM9001 microprocessor [22], and the verification of the Berkeley C string library on top of the MC68020 microprocessor [10]. The formal models in these projects are collectively several hundred pages long and involve many functions. Despite such successes, Nqthm was not designed for these kinds of large-scale projects and it has several inadequacies. The most important inadequacy of Nqthm is its lack of theorem proving power: if it would quickly settle every question put to it, one could proceed more efficiently. While we are always looking for better proof techniques (e.g., [33]), we do not know how to build a *significantly* more powerful and automatic theorem prover for Nqthm's logic.[2] Therefore, to "scale up" Nqthm we focused on engineering issues.

We decided that a good first step would be to adopt as a logic the applicative subset of a commonly used programming language, thereby gaining access to many efficient execution platforms for models written in the logic and many program development (i.e., modeling) environments. We chose Common Lisp because of its expressiveness, efficiency and familiarity. Properly formulated Common Lisp can execute at speeds comparable to C.

Three guiding tenets of the ACL2 project have been (1) to conform to all compliant Common Lisp implementations, (2) to add nothing to the logic that violates the understanding that the user's input can be submitted directly to a Common Lisp compiler and then executed (in an environment where suitable ACL2-specific macros and functions — the *ACL2 kernel* — are defined), and (3) to use ACL2 as the implementation language for the ACL2 system.

The third tenet is akin to recoding Nqthm in the Nqthm logic, a task that we believe would produce unacceptably slow performance. Programming the ACL2 system in ACL2

[2] We emphasize the word "significantly" here because ACL2's theorem prover is in fact more powerful than Nqthm in many ways. See Section 6.

repeatedly forced us to extend the subset so that we could write acceptably efficient code. Several iterations of the system were built. The current system consists of over 5 megabytes of applicative source code, including documentation.

The first version of the system was written in the summer and fall of 1989, by Boyer and Moore. As time went by, Boyer's involvement decreased and Kaufmann's increased. Eventually Boyer decided he should no longer be considered a coauthor. ACL2 has been used in modeling and verification projects within Computational Logic, Inc. (CLI), for several years. We released the first public version of ACL2 in September, 1995. See the URL http://www.cli.com.

3. The ACL2 Logic

The definition of Common Lisp used in our work has been [39, 40]. We have also closely studied [35].

The ACL2 logic is a first-order, quantifier-free logic of total recursive functions providing mathematical induction on the ordinals up to ϵ_0 and two extension principles: one for recursive definition and one for "encapsulation." We sketch the logic here.

3.1. Syntax

The syntax of ACL2 is that of Common Lisp. Formally, an ACL2 term is either a variable symbol, a quoted constant, or the application of an n-ary function symbol or lambda expression, f, to n terms, written `(`$f\ \ t_1..t_n$`)`. We illustrate the syntax for the primitive constants below. This formal syntax is extended by a facility for defining constant symbols and macros.

3.2. Rules of Inference

The rules of inference are those of Nqthm, namely propositional calculus with equality together with instantiation and mathematical induction up to ϵ_0. Two extension principles, recursive definition and encapsulation, are also provided; these are discussed in Subsection 3.6.

3.3. Axioms for Primitive Data Types

The following primitive data types are axiomatized.

- **ACL2 Numbers**. The numbers consist of the rationals and complex numbers with rational components. Examples of numeric constants are `-5`, `22/7`, and `#c(3 5)` (i.e., complex number $3 + 5i$).

- **Character Objects**. ACL2 supports 256 distinct characters, including Common Lisp's "standard characters" such as the character constants `#\A`, `#\a`, `#\,`, `#\Newline`, and `#\Space`.

- **Strings**. ACL2 supports strings of characters, e.g., the string constant `"Arithmetic Overflow"`.

- **Symbols**. Common Lisp provides a sophisticated class of objects called "symbol constants." Logically speaking, a symbol constant is an object containing two strings: a package and a name. The symbol constant with package `"MC68020"` and name `"EXEC"` is written `MC68020::EXEC`. By convention, one package is always selected as "current" and its name need not be written. Thus, if `"MC68020"` is the current package, the symbol above may be more simply written as `EXEC`. Packages may "import" symbols from other packages (although in ACL2 all importation must be done at the time a package is defined). If `MC68020::EXEC` is imported into the `"STRING-LIB"` package then `STRING-LIB::EXEC` is in fact the same symbol as `MC68020::EXEC`.

- **Lists**. ACL2 supports arbitrary ordered pairs of ACL2 objects, e.g., the list constant `(X MC68020::X ("Hello." (1 . 22/7)))`.

3.4. Axioms Defining Other Primitive Function Symbols

Essentially all of the Common Lisp functions on the above data types are axiomatized or defined as functions or macros in ACL2. By "Common Lisp functions" here we mean the programs specified in [39] or [40] that are (i) applicative, (ii) not dependent on state, implicit parameters, or data types other than those in ACL2, and (iii) completely specified, unambiguously, in a host-independent manner. Approximately 170 such functions are axiomatized.

Common Lisp functions are partial; they are not defined for all possible inputs. But ACL2 functions are total. Roughly speaking, the logical function of a given name in ACL2 is a completion of the Common Lisp function of the same name obtained by adding some arbitrary but "natural" values on arguments outside the "intended domain" of the Common Lisp function. ACL2 requires that every ACL2 function symbol have a *guard*, which may be thought of as a predicate on the formals of the function describing the intended domain. But guards are entirely extra-logical: they are not involved in the axioms defining functions. We discuss the role of guards when we explain the relation between ACL2 and Common Lisp.

3.5. Axioms for Additions to Common Lisp

To applicative Common Lisp we add four important new features by introducing some new function symbols and appropriate axioms.

- We add new multiple-valued function call and return primitives that are syntactically more restrictive but similar to the Common Lisp primitives `multiple-value-bind` and `values`. Our primitives require a function always to return the same number of values and to be called from contexts "expecting" the appropriate number of values. These restrictions allow our multiple-valued functions to be implemented more efficiently than Common Lisp's (at least in the case of Gnu Common Lisp). Logically speaking, a vector of multiple values returned by a function is just a list of the values; but the implementation is more efficient because the list is not actually constructed.

- We add an explicit notion of "state" to allow the ACL2 programmer to accept input and cause output. The input/output functions of Common Lisp are not in ACL2 because they are not applicative: they are dependent on an implicit notion of the current state. An ACL2 state is an n-tuple containing, among other things, the file system and open input/output "channels" to files. Primitive input/output functions are axiomatized to take a state as an explicit parameter and to return a new state as an explicit result (usually one of several results). Syntactic checks in the language ensure that the state is single-threaded, i.e., if a function takes state as an argument and calls a function that returns a new state, the new state (or more precisely, the final descendant of it) must be returned. This gives rise to a well-defined notion of the "current state" which is supplied to top-level calls of state-dependent ACL2 functions. The state returned by such calls becomes, by definition, the next current state. Because of these restrictions, the execution of a state-dependent function need not (and does not) actually construct new state n-tuples but literally modifies the underlying Common Lisp state.

- We add fast applicative arrays. These are implemented, behind the scenes, with Common Lisp arrays in a manner that always returns values in accordance with our axioms and operates efficiently provided certain programming disciplines are followed (namely, they are used in a single-threaded way so that only the most recently updated version of an array is used). There is no syntactic enforcement of the discipline; failure to follow it simply leads to inefficient (but correct) execution and warning messages.

- We add fast applicative property lists in a manner similar to that for arrays.

3.6. Extension Principles

Finally, ACL2 has two extension principles: definition and encapsulation. Both preserve the consistency of the extended logic [25]. Indeed, the standard model of numbers and lists can always be extended to include the newly introduced function symbols. (Inconsistency can thus be caused only if the user adds a new axiom directly rather than via an extension principle.)

The encapsulation principle allows the user to introduce new function symbols that are constrained by axioms to have certain properties. Consistency is ensured by requiring the user to exhibit witness functions satisfying the constraints. After a set of function symbols has been constrained, the witnesses used to establish consistency are irrelevant. The only axioms about the new functions are those stating the constraints. Theorems can then be proved about the constrained functions and these theorems can be instantiated in a higher-order way to derive analogous results about any functions satisfying the constraints. This is made possible by a derived rule of inference called functional instantiation [6].

The name "encapsulation" stems from the way the principle is implemented. An encapsulation command is essentially a "wrapper" around an admissible sequence of definitions and theorems. The wrapper allows one to mark certain of the definitions and theorems as "local." Local definitions and theorems are not "exported" from the wrapper; non-local ones are exported. Within the local context established by an encapsulation, the constrained functions are (locally) defined to be their witnesses and the constraints are (non-local) theorems about those functions. Outside the encapsulation, the function symbols are undefined and the theorems appear as (consistent) axioms. This implementation makes encapsulation very useful even when no new function symbols are introduced because it allows large proofs to be structured. See Section 6.

The definitional principle ensures consistency by requiring a proof that each defined function terminates. This is done, as in Nqthm, by the identification of some ordinal measure of the formals that decreases in recursion. In [8] we show (for Nqthm) that this ensures that one and only one set-theoretic function satisfies the recursive definition, and that proof carries over to the ACL2 case, with appropriate treatment of the non-uniqueness of any constrained functions used in the definition.

The form of an ACL2 function definition is as in Common Lisp,

(defun f ($x_1 ... x_n$) (declare ...) $body$)

ACL2 extends Common Lisp's `declare` so as to permit the specification of a guard expression, (g $x_1 ... x_n$), as well as to permit the optional specification of an ordinal measure and other hints. Some additional syntactic restrictions are put on $body$. These ensure that the Common Lisp version of f will execute efficiently and in accordance with claims we make below. Roughly speaking, it is here that we enforce the syntactic notion that the current state is single threaded (by restricting the use of the variable named `state`) and ensure that multiple values are used appropriately.

If the syntactic restrictions are met and the required termination theorems can be proved, then

Axiom .
(*f* x_1 ... x_n) = *body*

is added as a new axiom. Observe that the axiom added is independent of the guard.

4. The Relation Between ACL2 and Common Lisp

Guards have no role in the logic. However, they are crucial to the relation of the logic to Common Lisp.

The implicit guards of Common Lisp allow great efficiency. There are implementations of Common Lisp, for example, Gnu Common Lisp, in which the performance of the compiled code generated for arithmetic and list processing functions can be comparable to hand-coded C arithmetic and pointer manipulation. Exceptional execution efficiency on a wide variety of platforms, combined with clear applicative semantics when used properly, was one of the great attractions of basing the ACL2 logic on Common Lisp.

Consider for example the primitive function `car`. Page 411 of [40] says that the argument to `car` "must be" a cons or `nil`. On page 6 we learn "In places where it is stated that so-and-so 'must' or 'must not' or 'may not' be the case, then it 'is an error' if the stated requirement is not met." On page 5 we learn that 'it is an error' means that "No valid Common Lisp program should cause this situation to occur" but that "If this situation occurs, the effects and results are completely undefined" and "No Common Lisp implementation is required to detect such an error."

Thus, an implementation of the function `car` may assume its actual is a cons or `nil`. By a suitable representation of data, the implementation of `car` can simply fetch the contents of the memory location at which the actual is stored. No type checks are necessary. Of course, if `car` is applied to 7 the results are unpredictable, possibly damaging to the runtime image, and usually implementation dependent. These aspects of Lisp make it difficult to debug compiled Lisp code.

This also raises problems with the direct embedding of applicative Common Lisp into a logic. The situation is far worse than merely not knowing the value of `(car 7)`. We do not know that the value is an object in the logic: `(car 7)` might be π, for example. Worse still, we do not know that `car` is a function: the form `(equal (car 7) (car 7))`, which is an instance of the axiom `(equal x x)`, might sometimes evaluate to `nil` in some Common Lisps because the first `(car 7)` might return 0 and the second might return 1.

ACL2 solves this problem by axiomatizing `(car x)` to be a total function that returns `nil` outside the "intended domain" described by the guard `(or (consp x) (equal x nil))`. We claim our axioms describe Common Lisp's `car` when the argument to `car` satisfies our guard. Furthermore, ACL2 provides a general means of verifying that such a situation obtains in the evaluation of a given expression containing `car`.

While reading the rest of this section, the reader may wish to consider the possibility that our approach could be carried out for other programming languages.[3] Although programs are commercially available for mainstream languages such as C that check for certain kinds of errors, we believe that no utilities match the capability for making arbitrary semantic checks, statically, as we describe below for ACL2. Perhaps an analogous approach for C would present an opportunity for integration of formal verification into mainstream software development practice, lessening the need for dynamic error-checking.

4.1. Gold Function Symbols and Terms

To make precise the relation between ACL2 and Common Lisp, we define two inter-related notions: that of a function symbol being "gold"; and that of a term being "gold" under some hypothesis. (When no hypothesis is given, it is implicitly `t`, the true hypothesis.) Roughly speaking, a function symbol is gold if, when its guard is true, the guards of all subroutines encountered during evaluation are true of their arguments.

- All ACL2 logic primitive function symbols are *gold*.
- A defined function f with guard g and body b is *gold* if every function symbol mentioned in g is gold, the term g is gold, every function symbol besides f that is mentioned in b is gold, and the term b is gold under g.
- Variables and quoted constants are *gold* terms.
- The term (IF a b c) is *gold* under h if a is gold under h, b is gold under (AND h a) and c is gold under (AND h (NOT a)).
- The term (f a_1 ... a_n), where f is not IF and has guard (g v_1 ... v_n) (where the v_i are the formals of f) is gold under h provided each a_i is gold under

[3] We thank one of the referees for posing this question.

h and `(IMPLIES` h `(`g a_1 ... a_n`))` is a theorem. The formula that must be proved is called the *guard conjecture* for the subterm in question.

We sometimes say a function or term is *Common Lisp compliant* as a synonym for saying it is gold. We call the process of checking whether a function symbol or term is gold *guard checking* or *guard verification*.

4.2. The Story Relating the Logic to Common Lisp

We claim that if a function symbol of ACL2 is gold and a gold theorem has been proved about it, then every execution of that function in any compliant Common Lisp produces answers consistent with the theorem, provided the arguments to the function satisfy the guard and no resource errors (e.g., stack overflow) occur. Less precisely, gold ACL2 theorems describe the behavior of Common Lisp.

This claim can be made more precise as follows. We present the claim in a restrictive setting here for simplicity. Suppose f is a function symbol of one argument defined in some *certified book* (e.g., a file of admissible ACL2 definitions and theorems), that the guard of f is `t`, that f is gold, and that `(equal (`f x`) t)` is a (necessarily gold) theorem of ACL2 proved in that book. Consider any Common Lisp compliant to [40] into which the ACL2 kernel has been loaded. `Load` the book into that Lisp. Let x be a Common Lisp object that is also an object of ACL2. Then the application in that Lisp of f to x returns `t` or else causes a resource error (e.g., stack overflow or memory exhaustion).

The essence of the proof of this claim is to observe that `(`f x`)` evaluates to `t` in the logic (because of the soundness of the logic), and the computation will at no step exercise a function symbol outside of its guarded domain (because f is gold). Since the logic and Common Lisp agree inside the guarded domain, the Common Lisp computation of `(`f x`)` returns `t` also.

A less restrictive alternative formulation is that if thm is a gold theorem in some certified book then any ACL2 instance of thm evaluates to non-`nil` in any compliant Common Lisp into which the ACL2 kernel and the book have been loaded.

4.3. Guards and Efficiency

One obvious implication of the "Story" is that if one has a formal model that has been proved Common Lisp compliant and one wishes to evaluate gold applications of the model, one can ignore the ACL2 theorem prover altogether, load the model into a compliant Common Lisp (containing the ACL2 kernel), and directly execute the model to obtain results consistent with the axioms. For example, one might build a gold simulator of a microprocessor or high-level language and provide it to users via a stand-alone Common Lisp engine. The ACL2 theorem prover need not be present.

A less obvious use of our claim is made inside the ACL2 theorem prover. In the course of theorem proving it is not uncommon for ground subexpressions to arise, as by certain instantiations of lemmas, case splits on enumerated domains, the base cases of inductions, etc. Like Nqthm, ACL2 has an interpreter for evaluating such ground expressions. The ACL2 completions of the Common Lisp primitives are built in. Runtime type checks are done by this interpreter, e.g., to determine the value under the axioms of `car` applied to a constant, a, the interpreter determines whether a is a `consp` and then either uses Lisp's `car` or returns the default value `nil`. In this interpreter, calls of user-defined functions require recursive evaluation (and type-checking) of the body, repeatedly. But when ACL2 evaluates a call of a gold function symbol, it can use direct Common Lisp computation if the guard evaluates to t.

In both Nqthm and ACL2, the "interpreter" is implemented by defining and compiling auxiliary Common Lisp functions that do runtime type checking. Gold ACL2 functions are generally compiled by ACL2 and do no type-checking. Thus, both the interpreter approach and the gold short-circuit enjoy the benefits of compilation; the efficiency difference is ACL2's avoidance of runtime type checking for certain subexpressions. This can make a substantial difference in industrial-sized models.

In summary, one important incremental effect of proving that an ACL2 function symbol is gold is that subsequent applications of the function are more efficiently computed.

4.4. Guards as a Specification Device

Guards may also be used as type specifications. Gold functions are "well-typed." However, guards are much more expressive than conventional types, because they can be arbitrary terms in the logic. Of course, ACL2 "type checking" is not decidable for this same reason. For some related work, see [1].

If one attaches restrictive guards to one's functions and then proves the functions are gold, one obtains assurance that the functions are being exercised only on their intended domains. More precisely, one gains the knowledge that the computed value is provably equal to the function application in a weakened logical system in which the equality of each function application to its body is conditional on its guard being true. Nqthm provides no such assurance mechanism.

5. An Example

In this section we illustrate some of the points just made about guards.

5.1. Admitting a Definition

Consider the problem of concatenating two lists. We define the function app to do this, as follows.

```
(defun app (x y)
  (declare (xargs :measure (m x)
                  :guard (true-listp x)))
  (if (equal x nil)
      y
      (cons (car x) (app (cdr x) y))))
```

Ignore the declaration for the moment. Observe that the function terminates when x is nil and otherwise cdrs x in recursion. The intention is that x should always be a "true list," i.e., a cons tree whose right-most branch terminates in nil. If this function were applied to 7 and 8, in Common Lisp, the result would be unpredictable: 7≠nil, so we recur on (cdr 7) and 8, but (cdr 7) is undefined in Common Lisp. Replacing the (equal x nil) test with (atom x) is "more sensible" because then we know x is a cons when we cdr it. But Lisp programmers tend to use the test above because it is more efficient than a type check[4] and is equivalent provided x is a true list. The declaration of :guard (true-listp x) makes clear the intended domain.

Logically speaking, we must admit this function before we can reason about it. Logically speaking, the guard is irrelevant. We must show that a measure of the arguments decreases in the recursion, no matter what x or y is used. A suitable measure, (m x), is supplied by the user in the declaration; (m x) is defined (elsewhere) to be 0 if x is nil and otherwise to be one greater than the length of the right-most branch of x. It is easy to show that the measure decreases in the recursion, i.e., that (m (cdr x)) < (m x), when x≠nil. Intuitively, the recursion terminates because in ACL2 cdr has been completed to return nil on non-conses, so when the recursion hits the atom at the bottom of the right-most branch it stops if that atom is nil and otherwise it goes one more step, cdring the atom to produce a terminating nil.

After the function is admitted, the axiom

Axiom.

```
(app x y)
  =
(if (equal x nil)
    y
    (cons (car x) (app (cdr x) y)))
```

is added. Note that the axiom does not mention the guard.

[4] An equality test against a symbol can be done with a single address comparison and does not require a memory reference, as most type checks do.

5.2. Some Theorems

We can prove the surprising theorem

Theorem. surprising-app-call

```
(equal (app 7 8) (cons nil 8))
```

We will return to this surprising example later.

We can also prove the very useful and powerful unconditional equality stating that app is associative.

Theorem. associativity-of-app

```
(equal (app (app a b) c) (app a (app b c)))
```

The proof takes advantage of the fact that car and cdr return nil on non–cons arguments such as numbers.

If the guard for app infected the definition of app by limiting its applicability to true lists, then app would not be unconditionally associative. Identifying sufficient conditions can be difficult. Since a appears as the first argument of a call of app in the conjecture, a must be a true list in order to use the definition of app in that call. Similarly, b must be a true list. But (app a b) is also the first argument of a call of app; so to use the definition of app on that call the system must be able to establish — either from a third hypothesis or by proof from the other hypotheses — that (app a b) is a true list.

In short, if the guard for a function infects the definitional axiom, theorems inherit that complexity compositionally and are weakened by encumbering hypotheses. Such weakened theorems raise problems in three ways:

- they are harder for the user to state accurately;
- they are often harder to prove by induction because the induction hypothesis is weakened;
- they are harder to use subsequently because one must relieve their hypotheses.

Some of these same points are made in [27].

The decision that guards will not affect the definitional axioms thus has a far reaching effect. In fact, guards did play a logical role in earlier versions of ACL2, and we were driven to return to the Nqthm paradigm of total functions because of the complexity that guards introduced in some "industrial-strength" proof efforts, particularly the CAP project described later.

5.3. Compliance

Returning now to the app example, we next ask what is its relationship to Common Lisp? We can prove that app is gold, i.e., Common Lisp compliant: Every function used in its definition (except app itself) is gold (they are all primitive) and when its body is evaluated, every guard encountered is true if the guard for app is true initially. The latter condition can be expanded as follows. There are three subroutines in the body of app that have non-trivial guards: car, cdr, and the recursive call of app. For the definition to be gold, a theorem must be proved for each call of such a subroutine in the body of app. In particular, for each call we must prove that the guard of app, together with the tests leading to the call, imply the guard of the call. To prove app gold it therefore suffices to observe that if x is a non-nil true list, then x must be a cons and the cdr is itself a true list.

The extra-logical nature of guards is brought home by the observation that we could define another function, say xapp, that is analogous to app but contains no guard (i.e., has a guard of t). The two functions are provably equivalent. But app is Common Lisp compliant and xapp is not because its guard conjectures cannot be proved.

Because we know app is Common Lisp compliant, the "Story" tells us any call of app that satisfies the guard executes in accordance with the axioms of ACL2. Put another way, if we wish to determine the value under our axioms of (app a b), where a and b are constants, we can simply execute the expression in Common Lisp, provided a is a true list. Thus, the user wishing to execute a formal model on concrete data satisfying the guard can run the model in Common Lisp, provided the model has been proved compliant. Furthermore, because guards are gold terms, the suitability of the data can be determined by Common Lisp evaluation also.

5.4. Gold Theorems

What about our theorems about the gold function app? We have the surprising result that (equal (app 7 8) (cons nil 8)). Do we know that every compliant Common Lisp will evaluate this to true? No! The theorem is not gold. The guard for app is violated by 7. Nothing can be inferred about Common Lisp via our claim. Some Common Lisps may cause severe trouble if commanded to evaluate (app 7 8).

How about the associativity result for app? Can we expect app to be unconditionally associative in Common Lisp? Again, the answer is no because the theorem is not gold.

However, the following theorem is gold[5]

Theorem. gold-associativity-of-app
```
(implies (and (true-listp a)
              (true-listp b))
         (equal (app (app a b) c)
                (app a (app b c))))
```

[5] Here implies should be thought of as "lazy," i.e., (implies p q) should be read as (if p q t), to be given proper treatment by our definition of "gold."

Of course, we must prove this theorem, but its proof is trivial given the unconditional associativity result!

We must also verify that it is gold, i.e., that the guard of every call is satisfied by the arguments of the call, in the context of the call. The two true-listp hypotheses have true guards. The three calls of app in which the first argument is a variable symbol have guards requiring that the variable in question, a or b appropriately, is a true-listp. These guard conditions are trivial given the hypotheses. Finally, the guard condition for (app (app a b) c) generates the interesting guard condition that (app a b) is a true list when a and b are.

5.5. Separation of Concerns

Note how ACL2's treatment of guards separates concerns. For theorem proving simplicity in the Nqthm tradition ACL2 makes all functions total by completing the primitives with arbitrary but "natural" default values. Functions can be introduced into the logic without addressing the question of whether they are compliant with Common Lisp. Their properties can be proved without concerning oneself with questions of whether guards are satisfied. Often this allows the properties themselves to be more simply stated. This allows the data base of rules to be less restrictive, more powerful, and more easily applied. Nevertheless, non-gold functions are evaluated under the axioms with Nqthm efficiency.

Once a system of ACL2 functions has been defined and its logical properties proved, one can move on to the question of Common Lisp compliance, either to gain execution efficiency in an ACL2 setting or in a stand-alone Common Lisp, or to gain type assurance. Efficiency can be gained incrementally by doing guard verification on the core subroutines but not on the outlying checkers, preprocessors and postprocessors typically involved in a big system. Furthermore, having proved certain functions gold one can stop and settle for the corresponding efficiency or type assurance or one can prove that the key properties proved are also gold.

Recall for example that one can carry out the following sequence of steps:

- admit app as a function,
- prove it unconditionally associative,
- prove it gold, i.e., Common Lisp compliant or well-typed,
- trivially prove a restricted version of its associativity, and then
- prove that restricted version of associativity to be gold or Common Lisp compliant.

In versions of ACL2 predating Version 1.8, where guards were part of the definitional equations, these issues were often intertwined so that it was impossible to address them separately. While this makes little difference in a setting as simple as `app` and its associativity it makes a great deal of difference in models involving thousands of functions and properties.

6. Proof Engineering

We have argued that ACL2 is of "industrial strength." Up to now our main argument has been its improved efficiency over Nqthm by virtue of being executable as Common Lisp, with special consideration for efficient execution of operations involving arrays, property lists, and state. We have also indicated that a very expressive kind of type-correctness can be gained by "guard" verification, and yet this capability is separated from the logic proper so that proofs are not needlessly hindered. Below we consider some strengths of ACL2 besides efficiency as a programming language: robustness, general features, maintainability, and proof support.

6.1. Robustness

A notion of "industrial strength" is the robustness of the tool. We have put considerable effort into making the program bullet-proof, handling user errors graciously and with appropriate messages. The interface is consistent, providing the ability to submit definitions and theorems as well as the ability to execute applicative Lisp code efficiently.

6.2. Usability

Yet another notion of the "industrial strength" of a tool is its support for features that are crucial to get the job done. Here is a partial list of such features offered by ACL2.

- Extensible on-line documentation that may be read at the terminal, as well as through text and by way of hypertext (Emacs Info, HTML)
- Support for undoing back through a given command as well as "undoing an undo"
- A notion of "books" that allows independent development and inclusion of libraries of definitions and theorems:
  - books share the same underlying implementation as encapsulation in that forms within them can be marked "local" by the author of the book;
  - the reader of a book only sees the non-local definitions and theorems;
  - the possibly complex environment necessary to certify a book need not be exported to the reader's environment;
  - books are hierarchical and may include other books, locally or otherwise;
  - the reader may include many independently developed books to create his or her environment;
  - authors of books can install "theory invariants" to help the readers manage the environments created by multiple books;
  - authors of books can document the definitions and theorems in a book so that the book's inclusion updates the online documentation within ACL2;
  - consistency checks are done when books are included;
  - books carry certificates to help the community do version-control
- A "program" mode that allows definition and execution of functions without any proof burden being imposed, and without any risk of unsoundness being introduced (as the prover does not know about "program" mode functions)
- A "realistic" collection of data types that includes strings and (complex) rational numbers, with support for reasoning about such data (e.g., a fully integrated linear arithmetic decision procedure for the rationals)
- Extensive capabilities for controlling the prover (see below)
- Common Lisp macros for ease of programming and specification without cluttering up the collection of functions about which one needs to reason
- Many useful programming primitives, including efficient use of multiple values, arrays, property lists and file I/O
- Common Lisp *packages* that support distinct name spaces
- Mutually recursive definitions are supported

6.3. Maintainability

We also have found that the applicative style of programming is amenable to maintenance, both for fixing bugs and for implementing enhancements. Moreover, the subset of Common Lisp that ACL2 supports has been sufficient to code all but the very lowest levels of the system (which are needed to implement some of the primitives). We believe that a Common Lisp program that is applicative and perhaps even performs I/O is in fact an ACL2 program, or very nearly so.

6.4. Proof Support

As with Nqthm and Pc-Nqthm, proofs of significant theorems in ACL2 tend to require a serious effort on the part of the user to prove appropriate supporting lemmas, primarily stored and used as (conditional) rewrite rules. However, ACL2 offers many other ways by which the user can control the proof engine:

- As with Nqthm, a proof commentary in English that assists users in debugging failed proofs
- As with Pc-Nqthm, an interactive loop for proof discovery that is extensible through "macros" and has access to the full power of the theorem prover
- The capability to apply hints to individual subgoals
- "Proof tree" displays to show the evolving structure of the proof in real time, and which also make it convenient to inspect failed proofs
- Efficient handling of propositional logic, normally through a clause generator that is much more efficient than Nqthm's, but also through a facility that integrates ordered binary decision diagrams with rewriting
- A "functional instantiation" facility that gives ACL2 (like Nqthm) some of the convenience of a higher-order logic without sacrificing the simplicity of a first order logic
- A "break-rewrite" facility, more sophisticated than that in Nqthm (or any other prover as far as we know), that allows proof-time debugging of the rewrite stack
- A "theory" mechanism that makes it easy to manipulate sets of rules, especially turning them on and off but also checking desired invariants on sets of rules
- A "forcing" mechanism that gives the prover permission to defer checking of hypotheses of specified rules until the end of the "main" proof
- Support for a variety of types of rules (17), including types supported by Nqthm. These include:
  - Conditional rewrite rules, which may be used not only to replace equals by equals but may also work with respect to user-defined equivalence (congruence) relations
  - Linear arithmetic rules, together with a mechanism to create rules for certain simple orders other than the standard "less-than" order on the rationals
  - "Compound recognizer," "forward chaining," "type prescription," "equivalence," "congruence," and "built-in clause" rules for efficient automatic use of certain facts
  - An improved "meta lemma" facility that allows such lemmas to be conditional (i.e., have hypotheses)
  - Rules for use outside the simplifier/rewriter: elimination and generalization

Thus, we claim that ACL2 is industrial strength in its efficiency, its consistent and robust interface, its array of general features, its ease of maintenance, and the flexibility of its theorem prover.

7. Industrial Applications

Of course, ultimately the test for whether a tool is "industrial strength" must be whether it can be used to do jobs of interest to industry. The first two important applications of ACL2 support our claims that it is up to the task. These applications, summarized below, are discussed in more detail in [12] and [31], where we also detail the time and manpower resources spent on the component tasks.

7.1. Motorola CAP Digital Signal Processor

Bishop Brock of CLI, working in collaboration with Motorola, Inc., produced an executable formal ACL2 specification of the Motorola CAP [19], a digital signal processor designed by Motorola to execute a 1024 point complex FFT in 131 microseconds. Every well-defined behavior of the CAP is modeled, including the pipeline, I/O, interrupts, breakpoints and traps (but excluding the hard and soft reset sequences). The CAP is much more complex than other processors recently subjected to formal modeling, namely the FM90001 [22], MC68020 [10], and AAMP5 [30]. In principle, a CAP single instruction can simultaneously modify well over 100 registers. Brock's ACL2 model of the CAP is bit-accurate and cycle-accurate but runs faster than Motorola's SPW model. Furthermore, ACL2 can be used to reason about the CAP model. ACL2 can compute the symbolic effects of a complicated instruction in just a few seconds. With ACL2, Brock has proved that under suitable conditions his model of the CAP is equivalent to a simpler pipeline-free model.

Perhaps the most important aspect of the CAP work is that with ACL2 it is possible to prove the correctness of

CAP microcode programs. Because of the complexity of the instruction set, mechanical analysis of CAP microcode programs is perhaps the only way to assure that the programs have certain properties. Brock used ACL2 to verify the microcode produced by Motorola's assembler for several CAP application programs, including a FIR filter and a peak finding algorithm that uses the adder array as a chain of comparators. This work is discussed further in [12].

Following the same approach used with Nqthm for the MC68020 in [10], Brock configured ACL2 so as to make it easy to symbolically execute a CAP microcode program on symbolic data. He then specified and mechanically proved the total correctness of the microcode programs he considered. It is possible (and often less labor intensive) to use the CAP model and ACL2 to prove weaker properties of microcode, such as that no errors occur. We believe that once a microcode engine such as the CAP is specified in ACL2 and a few application programs are mechanically verified for it, subsequent microcode applications can be done "routinely." By "routinely" here we mean that the technical details of symbolic execution for that particular microcode can be managed by ACL2 and the effort of the verification task is dominated by consideration of the specification and behavior of the given program.

Another important aspect of the ACL2 CAP model is that it executes CAP programs faster than Motorola's SPW model. This makes it a convenient debugging tool. There are two reasons for its speed. One is that it is coded at a somewhat higher level than the SPW model, i.e., arithmetic in the model is arithmetic in Common Lisp, not (simulated) combinational logic. The other is that Brock used strict guards that ensure that the model is type correct: registers contain data of appropriate size, addresses are legal, objects used as CAP machine states have the appropriate form, etc. All the functions in the CAP model are proved gold. Thus, the ACL2 CAP model can be compiled and faithfully executed directly in Common Lisp.

7.2. AMD5$_K$86 Floating-Point Division

In another test of the industrial applicability of ACL2, we collaborated with Tom Lynch of Advanced Micro Devices, Inc., to formalize and mechanically prove the correctness of the microcode for the kernel of the floating point division operation used on the AMD5$_K$86 microprocessor, AMD's first Pentium-class processor. In particular, in [31] we prove that when p and d are double extended precision floating-point numbers ($d \neq 0$) and $mode$ is a rounding mode specifying a rounding style and target format of precision $n \leq 64$, then the result of the algorithm is p/d rounded according to $mode$.

As explained in detail in [31], the algorithm uses a table to obtain an 8-bit approximation to the reciprocal of d. Then two iterations of an efficiently computed variation of the Newton-Raphson iteration are used to refine this approximation so that the relative error is less than 2^{-28}. That approximation is then used to compute four floating-point numbers whose sum is sufficiently close to p/d that when it is rounded according to $mode$ the result is the same as rounding p/d instead.

This algorithm is implemented in microcode; all of its arithmetic computations use floating point operations with directed rounding. To prove that the algorithm works as specified we developed in ACL2 much floating-point "folklore." We also formalized the algorithm in ACL2 and then used ACL2 to check a fairly deep mathematical proof. A necessary step in our proof is to show that every intermediate result fits in the (floating-point) resources allocated to it by AMD.

How are the "industrial strength" aspects of ACL2 used in this proof? The fact that ACL2 is executable is important. Our proof that two applications of Newton-Raphson iteration produce a sufficiently accurate answer generalizes away from the particular table used by AMD. We define a predicate that recognizes when a table contains sufficiently accurate 8-bit approximations to d and prove that if the Newton-Raphson steps start from such a table, a correct answer is produced. The predicate is executable: given a concrete table, Common Lisp can determine by evaluation whether it satisfies the predicate. Thus, to apply the result to the actual AMD table, ACL2 merely executes the predicate on the AMD table.

Other aspects of ACL2 that were crucial to this proof were encapsulation, macros and books. By using encapsulation and macros we were able conveniently to configure ACL2 temporarily to derive the key steps in the proof, without having to impose the same proof strategy on each key step. By using books we were able to partition responsibility for various parts of the proof among the collaborators and then assemble the final results.

Subsequent to our proof of the floating-point division microcode, David Russinoff used ACL2 to prove the correctness of the AMD5$_K$86 's floating-point square root microcode [37].

8. Conclusion

ACL2 is a re-implemented extended version of Boyer and Moore's Nqthm and Kaufmann's Pc-Nqthm, intended for large scale verification projects. The ACL2 logic is an extension of a large applicative subset of Common Lisp, supporting a practical collection of data types, single threaded states, I/O, multiple-valued functions, arrays and property lists. By careful design of the notion of guards, ACL2 allows the elegant expression and proof of theorems in this logic without many of the encumbrances of "type-like" hypotheses, while at the same time allowing functions in the logic to be related to Common Lisp in a way that allows extremely efficient computation. Furthermore, the design allows for a clear separation of the problems of admission of logical definitions, proofs of simply stated properties, type-correctness, Common Lisp compliance and efficient execution. The design allows for incremental achievement of these goals via proof, allowing the user to focus effort on the important aspects of the project.

ACL2 provides a wide variety of features supportive of its industrial strength goals, including a rugged and extensively documented implementation, many convenient features for constructing models and developing and structuring proofs, and good maintainability. Finally, we have demonstrated that ACL2 can be used to tackle problems of importance to industry.

ACKNOWLEDGMENTS

Without Bob Boyer's contributions, especially in the initial stages of ACL2 development, it is quite possible that ACL2 would never have been created. We are also especially grateful to Bishop Brock for providing valuable feedback based on his heavy use of ACL2 for many months. Bill Young improved and contributed tutorial material to the documentation. Art Flatau, Noah Friedman, and Laura Lawless contributed to technical aspects of the documentation and provided useful feedback. We also acknowledge contributions from Bill Bevier, Alessandro Cimatti, Rich Cohen, John Cowles, Warren Hunt, Jun Sawada, Bill Schelter, and Mike Smith. Finally, we are grateful for the utility provided by Gnu Emacs; in particular, Richard Stallman provided a Texinfo patch to assist in building the documentation. We also thank the anonymous referees of this paper who provided many valuable suggestions.

This paper is an extended version the paper "ACL2: An Industrial Strength Version of Nqthm," which appeared in *Proceedings of the Eleventh Annual Conference on Computer Assurance* (COMPASS-96), IEEE Computer Society Press, pp. 23–34, June, 1996.

The work reported here was supported in part at Computational Logic, Inc., by the Advanced Research Projects Agency, ARPA Order D245, under Contract No. N66001-95-C-8634; ARPA Order 7406, and the Office of Naval Research, under Contract No. N00014-94-C-0193. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of Computational Logic, Inc. or the United States Government.

References

[1] R. L. Akers. Strong Static Type Checking for Functional Common Lisp. Ph.D. Thesis, University of Texas at Austin, 1993. Also available as Technical Report 96, Computational Logic, Inc., Suite 290, 1717 West Sixth Street, Austin, TX 78703. URL http://www.cli.com/.

[2] K. Albin. 68020 Model Validation Testing, CLI Note 280, August 1993.

[3] D. Basin and M. Kaufmann, The Boyer-Moore Prover and Nuprl: An Experimental Comparison. In: *Proceedings of the First Workshop on "Logical Frameworks"*, Antibes, France, May 1990.

[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] W. R. Bevier and W. D. Young. Machine Checked Proofs of the Design of a Fault-Tolerant Circuit, *Formal Aspects of Computing*, Vol. 4, pp. 755–775, 1992. Also available as Technical Report 62, Computational Logic, Inc., Suite 290, 1717 West Sixth Street, Austin, TX 78703, August, 1990 (URL http://www.cli.com/), and as NASA CR-182099, November, 1990.

[6] R. S. Boyer, D. Goldschlag, M. Kaufmann, and J S. Moore. Functional Instantiation in First Order Logic. In *Artificial Intelligence and Mathematical Theory of Computation: Papers in Honor of John McCarthy*, Academic Press, 1991, pp. 7-26.

[7] Robert S. Boyer, Matt Kaufmann, and J Strother Moore. The Boyer-Moore Theorem Prover and Its Interactive Enhancement. *Computers and Mathematics with Applications*, **29**(2), 1995, pp. 27-62.

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

[9] R. S. Boyer and J S. Moore. *A Computational Logic Handbook*, Academic Press: New York, 1988.

[10] R. S. Boyer and Y. Yu, Automated Correctness Proofs of Machine Code Programs for a Commercial Microprocessor. In D. Kapur, editor, *Automated Deduction – CADE-11, Lecture Notes in Computer Science 607*, Springer-Verlag, 416–430, 1992.

[11] R. K. Brayton, G. D. Hachtel, A. Sangiovanni-Vincentelli, F. Somenzi, A. Aziz, S.-T. Cheng, S. Edwards, S. Khatri, Y. Kukimoto, A. Pardo, S. Qadeer, R. K. Ranjan, S. Sarwary, T. R. Shiple, G. Swamy, T. Villa. VIS: A system for Verification and Synthesis. 8th Intl. Conf. on Computer-Aided Verification, July, 1996.

[12] B. Brock, M. Kaufmann and J S. Moore. ACL2 Theorems about Commercial Microprocessors. In M. Srivas and A. Camilleri, editors, *Formal Methods in Computer-Aided Design (FMCAD'96)*, Springer-Verlag, November 1996 (to appear).

[13] S.-C. Chou, Mechanical Geometry Theorem Proving, D. Reidel Publishing Co., Dordrecht, Holland, 1988.

[14] E. M. Clarke and X. Zhao. Analytica: A theorem prover for Mathematica. *The Journal of Mathematica*, 3(1), pp. 56–71,Winter 1993.

[15] R. L. Constable, *et al.*, *Implementing Mathematics with the Nuprl Proof Development System*, Prentice Hall, 1986.

[16] C. Cornes, J. Courant, J.-C. Filliatre, G. Huet, P. Manoury, et al. The Coq Proof Assistant, Reference Manual, Version 5.10. RT-0177, INRIA, B.P. 105, 78153 Le Chesnay Cedex, France.

[17] D. Craigen, S. Kromodimoeljo, I. Meisels, B. Pase, M. Saaltink. EVES: An Overview. Odyssey Research Center, ORA Conference Paper CP-91-5402-43, March, 1991.

[18] J. Crow, S. Owre, J. Rushby, N. Shankar, and M. Srivas. A Tutorial Introduction to PVS, presented at *Workshop on Industrial-Strength Formal Specification Techniques*, Boca Raton, FL, April 1995 (see http://www.csl.sri.com/pvs.html).

[19] S. Gilfeather, J. Gehman, and C. Harrison. Architecture of a Complex Arithmetic Processor for Communication Signal Processing. In *SPIE Proceedings, International Symposium on Optics, Imaging, and Instrumentation*, **2296** *Advanced Signal Processing: Algorithms, Architectures, and Implementations V*, 624–625, July, 1994.

[20] M. J. C. Gordon and T. F. Melham (editors). *Introduction to HOL: A Theorem Proving Environment for Higher-Order Logic.* Cambridge University Press, 1993.

[21] W. M. Farmer, J. D. Guttman, and F. J. Thayer. IMPS: An Interactive Mathematical Proof System. *Journal of Automated Reasoning*, **11**(2), 213–248, 1993.

[22] W. A. Hunt, Jr. and B. Brock. A Formal HDL and its use in the FM9001 Verification. *Proceedings of the Royal Society*, 1992.

[23] M. Kaufmann. A User's Manual for an Interactive Enhancement to the Boyer-Moore Theorem Prover, Technical report 19, Computational Logic, Inc., May, 1988. URL http://www.cli.com/.

[24] M. Kaufmann and P. Pecchiari. Interaction with the Boyer-Moore Theorem Prover: A Tutorial Study Using the Arithmetic-Geometric Mean Theorem. *Journal of Automated Reasoning* 16, no. 1-2 (1996) 181-222.

[25] M. Kaufmann and J S. Moore, High-Level Correctness of ACL2: A Story (DRAFT), URL ftp://ftp.cli.com/pub/acl2/v1-8/acl2-sources/-reports/story.txt, September, 1995. Revision in preparation.

[26] K. Kunen. A Ramsey Theorem in Boyer-Moore Logic, *Journal of Automated Reasoning*, **15**(2) October, 1995.

[27] L. Lamport, Types are Not Harmless, http://www.-research.digital.com/SRC/tla/tla.html, July, 1995.

[28] W. McCune. Otter 3.0 Reference Manual and Guide. Report ANl-94/6, Argonne National Laboratory (1994).

[29] Kenneth L. McMillan. *Symbolic Model Checking*. Kluwer Academic Publishers, 1993.

[30] S. P. Miller and M. Srivas. Formal Verification of the AAMP5 Microprocessor: A Case Study in the Industrial Use of Formal Methods, in *Proceedings of WIFT '95: Workshop on Industrial-Strength Formal Specification Techniques*, IEEECS, April, 1995, pp. 2–16.

[31] J S. Moore, T. Lynch, and M. Kaufmann. A Mechanically Checked Proof of the Correctness of the Kernel of the AMD5$_K$86 Floating Point Division Algorithm, March, 1996 (submitted). URL http://-devil.ece.utexas.edu:80/~lynch/divide/divide.html.

[32] J S. Moore. A Formal Model of Asynchronous Communication and Its Use in Mechanically Verifying a Biphase Mark Protocol, *Formal Aspects of Computing* **6**(1), 60–91, 1994.

[33] J S. Moore. Introduction to the OBDD Algorithm for the ATP Community, *Journal of Automated Reasoning* 12, 33–45, 1994.

[34] L. C. Paulson, *Isabelle: A Generic Theorem Prover*, Springer-Verlag LNCS 828, 1994.

[35] K. M. Pitman *et al. draft proposed American National Standard for Information Systems — Programming Language — Common Lisp; X3J13/93-102.* Global Engineering Documents, Inc., 1994.

[36] D. M. Russinoff. A Mechanical Proof of Quadratic Reciprocity. *Journal of Automated Reasoning*, **8**(1), 3–21, 1992.

[37] D. Russinoff, A Mechanically Checked Proof of Correctness of the AMD5$_K$86 Floating-Point Square Root Microcode, http://www.onr.com/user/-russ/david/fsqrt.html, February, 1997.

[38] N. Shankar. *Metamathematics, Machines, and Gödel's Proof*, Cambridge University press, 1994.

[39] G. L. Steele Jr. *Common LISP: The Language*, Digital Press: Bedford, MA, 1984.

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

[41] A. Trybulec. Built-in Concepts. Journal of Formalized Mathematics, Axiomatics, 1989.

[42] W. D. Young, Comparing Verification Systems: Interactive Consistency in ACL2, *IEEE Trans. Software Engineering*, [this issue].

Matt Kaufmann was trained as a mathematical logician at the University of Wisconsin, receiving his PhD in mathematics there in 1978 after receiving his bachelor's degree at MIT in 1973. His first career led to ~20 research papers in set-theoretic model theory, primarily while on the the mathematics faculty at Purdue University. He then migrated into computer science, first focusing on functional programming in a research position at Burroughs Corp. and then working for several years at Computational Logic, Inc. on general-purpose automated reasoning and applications. Dr. Kaufmann joined Motorola, Inc. in August, 1995, where he currently works on formal verification of hardware.

J Strother Moore received his PhD degree from the University of Edinburgh in 1973 and his BS degree from the Masachusetts Institute of Technology in 1970. He holds the Admiral B.R. Inman Centennial Chair in Computing Theory at the University of Texas at Austin. He is the author of many books and papers on automated theorem proving and mechanical verification of computing systems. His most recent book is *Piton: A Mechanically Verified Assembly-Level Language* (Kluwer, 1996). Along with Bob Boyer he is a co-author of the Boyer-Moore theorem prover and the Boyer-Moore fast string searching algorithm. He and Bob Boyer were awarded the 1991 Current Prize in Automatic Theorem Proving by the American Mathematical Society. Moore is a Fellow of the American Association for Artificial Intelligence.