Design Goals for ACL2

Matt Kaufmann and J Strother Moore

Technical Report 101 August, 1994

Computational Logic, Inc.
1717 West Sixth Street, Suite 290
Austin, Texas 78703-4776

TEL: +1 512 322 9951
FAX: +1 512 322 0656

EMAIL: kaufmann@cli.com and moore@cli.com

This work was supported in part at Computational Logic, Inc., by the Defense Advanced Research Projects Agency, ARPA Order 7406. The views and conclusions contained in this document are those of the author(s) and should not be interpreted as representing the official policies, either expressed or implied, of Computational Logic, Inc., the Defense Advanced Research Projects Agency or the U.S. Government.

Abstract

ACL2 is a theorem proving system under development at Computational Logic, Inc., by the authors of the Boyer-Moore system, Nqthm, and its interactive enhancement, Pc-Nqthm, based on our perceptions of some of the inadequacies of Nqthm when used in large-scale verification projects. Foremost among those inadequacies is the fact that Nqthm's logic is an inefficient programming language. We now recognize that the efficiency of the logic as a programming language is of great importance because the models of microprocessors, operating systems, and languages typically constructed in verification projects must be executed to corroborate them against the realities they model. Simulation of such large scale systems stresses the logic in ways not imagined when Nqthm was designed. In addition, Nqthm does not adequately support certain proof techniques, nor does it encourage the reuse of previously developed libraries or the collaboration of semi-autonomous workers on different parts of a verification project. Finally, Nqthm is implemented in an informally specified programming language (Common Lisp) and hence is not subject to mechanical verification. ACL2 is our response to these perceived inadequacies. While the logic of Nqthm is based on pure Lisp, the logic of ACL2 is based on the applicative subset of Common Lisp. By adding to the applicative subset of Common Lisp a single-threaded notion of state, fast applicative arrays and property lists, and efficiently implemented multiple values, an efficient and practical applicative programming language is produced. By axiomatizing the primitives and introducing appropriate rules of inference and extension principles, that language can be turned into a logic. A premise of the ACL2 Project is that the Nqthm proof heuristics allow the mechanization of the discovery of proofs in the ACL2 logic with the same degree of success that Nqthm has for its logic. The ACL2 system may be viewed as an extended re-implementation of Nqthm for extended applicative Common Lisp. ACL2 is written using the logic it supports. It provides all of Nqthm's proof techniques (except those for `V&C$`), as well as several that Nqthm does not provide, including forward chaining and congruence-based rewriting. An incremental data base extension facility, based on hierarchically structured "books" and several scoping mechanisms, is provided to address the problems of reusability and collaborative proof efforts. We discuss the inadequacies of Nqthm motivating the design of ACL2; we briefly describe the ACL2 logic, theorem prover, interface, implementation, and some applications; and we discuss some of our concerns and misgivings about the current design. Because ACL2 is not yet ready for public distribution, we make no claims as to its superiority to Nqthm or other theorem proving systems.

1 Disclaimer

ACL2 is being developed by Robert S. Boyer, Matt Kaufmann, and J Strother Moore, at Computational Logic, Inc. (CLI). It is the intention of the authors and CLI to release ACL2 for public use, without fee, when its reliability and documentation are up to our standards. Those levels have not yet been reached. To our colleagues who wish to try out ACL2 for themselves, we apologize and ask for their continued patience. Visitors to CLI are welcome to use it, but we do not want preliminary versions floating around the net.

If geographical exploration is taken as a metaphor for theorem proving research, then the present paper should be read merely as a scouting report of where we are headed and why. It should not be taken as a recommendation that anyone follow our trail, much less as an advertisement to buy parcels of the land we have surveyed.

Because of his role in the ongoing development of ACL2, it would have been entirely appropriate for Boyer to be a co-author of this paper. He declined for fear of seeming hypocritical after his recent exhortations to the ATP community not to publish papers about theorem proving systems unavailable for public scrutiny. We support his position when the paper in question claims that the described system has been found empirically to be superior to existing systems. We make no such claims about ACL2 in this paper.

2 Mathematical Modeling of Digital Systems

The mathematics of computation was identified in the 1930's by the collective work of Church, Kleene and Turing. They established that recursive functions can be used to model digital computation. By operating within a logical framework, deductions about computational models can be carried out formally. By mechanizing the formal logic, one can assist the human user in the proof discovery process as well as eliminate logical errors from conjectures and proofs.

Following the way mathematics is generally used in engineering, system verification proceeds in three steps. First, a formal model of the desired digital system is constructed. Second, the model is corroborated, usually by executing it on concrete test data, to confirm that it exhibits the desired behavior on some finite set of tests. Often these first two steps are iterated until the model is deemed a suitably accurate specification of the requirements. Finally, theorems are proved about the model to establish some of the interesting properties of the modeled system. Since the state space of models of digital systems is often exceedingly large or even infinite (in the case of some higher-level specifications), proof is often the only practical means of confirming properties of a model.

In our work, we use a logic based on recursive functions. In the first step, above, we exploit the fact that recursive functions can model any computation. In the second we exploit the fact that recursive functions can be executed. In the third we exploit the fact that recursive functions can be embedded in a logical framework so as to provide formal (and hence machine-checkable) notions of deduction and proof.

Our models usually take the form of *abstract machines* defined as recursive functions in the formal logic. Generally speaking, these functions take two inputs: a "state" and some "signals" that impinge upon the machine over "time." Such an abstract machine returns the final state of the machine after processing all of the signals. It does this by "stepping" through a sequence of states, each successive state being obtained by applying the machine's "step function" to the current state and the signal (if any) that arrives at that "time."

Such machines are commonly used as formalizations of programming languages. However, they have also been used to model other aspects of the digital systems problem, including hardware description languages, operating systems, concurrent programs, physical systems interacting with digital ones, and requirements modeling.

3 Nqthm: The Prototype of ACL2

To define abstract machines formally and reason about them, one must have a mathematical logic that provides inductively constructed objects such as numbers, sequences, and trees, and principles of recursive definition and inductive proof. The "Nqthm" system, developed by Boyer and Moore, provides such a logic and a mechanized theorem prover for it [8, 11]. Nqthm is widely used in the formal modeling of digital systems. Nqthm is well known for its robustness and the extensive body of verification work done with it.

However, for the past five years the two authors and Robert S. Boyer have been developing a new logic and theorem prover, called "ACL2." ACL2 has adopted (and often attempted to improve) almost all of the ideas behind Nqthm. ACL2 was designed to correct the flaws uncovered by two decades of use of Nqthm. Those flaws primarily concern the scale of the projects to be undertaken with the system.

Because Nqthm is the prototype of ACL2, we begin our discussion of ACL2 by briefly reviewing Nqthm and some of its applications and flaws.

3.1 The Nqthm System

The Nqthm logic is a first order, quantifier-free logic resembling pure Lisp. The logic provides for the schematic introduction of new inductively defined data types, mathematical induction on the ordinals up to ϵ_0, the definition of total recursive functions, and the witnessed constraint of new function symbols coupled with a derived rule of functional instantiation giving the logic some of the features of a higher-order logic. In addition, the logic provides an axiomatization of a nonconstructive function, `V&C$`, which is an interpreter for the logic and allows the introduction of any partial recursive function. See [11, 6] for details.

The mechanization of the Nqthm logic is a system of Common Lisp programs allowing the user to define functions in the logic, execute them on concrete data, and state and prove theorems about such functions. The user interface to Nqthm is the Common Lisp read-eval-print loop: Common Lisp forms are typed to define functions in the logic, invoke the theorem prover, etc. A special environment is provided in which forms in the logic may be executed. The Nqthm theorem prover uses a variety of proof techniques, e.g., simplification and induction. These techniques are sensitive to rules in a data base. Hundreds of heuristics orchestrate the application these rules. An interactive proof checker for Nqthm, called "Pc-Nqthm" has also been developed [20, 21].

Important to Nqthm's success has been the fact that when a new user-supplied theorem is proved, rules are derived from it and stored in its data base; these rules change the way the system behaves. By stating an appropriate collection of lemmas the user can effectively program Nqthm to prove theorems in a given domain. A well chosen sequence of lemmas can lead Nqthm to the proofs of very deep theorems. Target theorems can often be changed and "re-proved" automatically because proof scripts tend to describe powerful and general-purpose rules for manipulating the concepts rather than particular proofs. That is, the user programs Nqthm to deal with the concepts and expects Nqthm to fill in the gaps between application-specific lemmas describing the proof at a high level. This makes it easy to "maintain" an evolving system of definitions and theorems if the system was initially verified with Nqthm. That, in turn, has allowed Nqthm to accumulate a vast quantity of benchmark theorems.

The latest version of Nqthm, named "Nqthm-1992," was released in 1994. A companion "Pc-Nqthm-1992" was also released. To obtain Nqthm-1992, connect to Internet site ftp.cli.com by anonymous ftp, giving your email address as the password, 'get' the file `/pub/nqthm/nqthm-1992/README`, and follow the instructions therein. (Analogous instruction apply to Pc-Nqthm-1992.) Nqthm is documented in two books [8, 11], and Pc-Nqthm is documented in [20, 21, 22]. Both systems and many of their applications are briefly described in [7]. A detailed tutorial introduction to Nqthm and Pc-Nqthm may be found in [24]. The recent Nqthm-1992 release includes 1.3 megabytes of updated documentation consisting of new versions of the five most important chapters in [11]. In addition, the releases include more than 17 megabytes of example input for Nqthm and Pc-Nqthm, including most of the important benchmarks listed below.

3.2 Some of Nqthm's Applications

Space does not allow even a brief but exhaustive summary of theorems proved with Nqthm; we therefore only describe a few. The theorems alluded to below were selected to illustrate the expressivity, flexibility, and power of Nqthm to deal with digital systems and the related mathematics. Why do we dwell on Nqthm's successes when we are here interested in its flaws? The relative weakness of Nqthm's quantifier-free, first order logic could be regarded as a flaw to be corrected but we cite these examples to establish the fact that the expressivity of the logic is not a serious bottleneck.

- **Mathematics for Computation** Among the well-known Nqthm benchmarks are Gauss' law of quadratic reciprocity [31] and Gödel's incompleteness theorem for Shoenfield's first order logic extended with Cohen's axioms for hereditarily finite set theory, Z2, [34]. The first illustrates Nqthm's use in deep reasoning about integers, perhaps the most commonly used mathematical construct in computing. The second illustrates Nqthm's use in modeling other formal systems, a capability that has often been exploited to use Nqthm to do proofs "in" other computational logics: Nqthm has proved the soundness and completeness of a propositional calculus decision procedure [8], the Turing completeness of pure Lisp [10], the Church-Rosser theorem for lambda calculus [33], and the soundness of the proof rules of Misra and Chandy's Unity logic [14, 16].

- **System Verification** Perhaps most representative of digital systems verification is the now classic example, the "CLI short stack," which combines both hardware and software verification. The short stack consists of a chain of abstract machines starting with a microprocessor (the FM8502) described at the gate-level and ending with an operational semantics for a simple high-level programming language (Micro-Gypsy). Between each pair of machines in this chain is an Nqthm function implementing the higher machine on the lower one. At the highest level the implementing function is a "(cross-)compiler;" as one descends the implementing function is called an "assembler," a "linker" and eventually a "downloader." Nqthm has done the proofs necessary to establish that each higher-level machine is correctly implemented on the next lower-level machine. Because of the constructive nature of the Nqthm logic, the statement of each such theorem involves the definition of a "clock function" which calculates the number of low-level steps necessary to carry out a given number of high-level steps. Nqthm has also done the proofs necessary to "glue" these results together. Thus, assuming the soundness of the Nqthm theorem prover and the correctness of our models, it is known with mathematical certainty that the results of any given Micro-Gypsy program on any given data can be computed by running the microprocessor on the result of compiling, assembling, linking, and downloading the initial high-level program and data. Furthermore, the clock functions can be composed so that it is possible to say how many microcycles are necessary to do a given high-level computation. This work consumed several man years and is described in [4].

- **Hardware Verification and System Maintenance** Since the publication of the short stack work, CLI has designed, verified, and fabricated another microprocessor, called the FM9001 [19]. This involved formalizing the NDL hardware description language [13] of LSI Logic Inc., and verifying that a certain collection of NDL hardware modules implemented a formally described machine language. In what we regard as a significant demonstration of Nqthm's ability to support the maintenance of verified systems, the CLI short stack was ported to the FM9001 in less than one man-week by retargetting the assembler and linker from the FM8502 to the FM9001 and then (re-)verifying them. We say "(re-)verifying" because the theorems proved had not actually been proved before but were merely analogous to those proved in the FM8502 work. This required relatively minor modification of the proof scripts developed for the FM8502 and relied upon Nqthm's theorem prover to fill in the gaps.

- **Software** A substantial subset of the MC68020, a widely used microprocessor built by Motorola, has been formalized within Nqthm. Roughly 80% of the user available instructions were formalized. (Most of the remaining instructions have undefined effects on the machine state.) All eighteen MC68020 addressing modes were formalized. Using this formal description of the machine it is possible to analyze formally the behavior of given MC68020 object code programs. Consider, for example, the C program

```
isqrt(int i)
{
  int j;
  j = (i / 2);
  while ((i / j) < j)
    j = (j + (i / j)) / 2;
  return (j);
}
```

  which computes the integer square root of a given nonnegative integer. By compiling this program with the Gnu C compiler, `gcc -O`, one may obtain MC68020 object code. One can then state and prove the theorem that if a suitably configured MC68020 is executed a certain number of steps starting from an invocation of the `isqrt` object code on suitable input, i, the result left in `D0` is the greatest integer whose square is less than or equal to i. In addition, one can prove that execution proceeds without error and that the final machine state is suitable for continued execution (e.g., `A6`, used by `LINK`, is unchanged).

  The correctness of MC68020 object code programs for binary search, Hoare's Quick Sort, and some other well known algorithms have been mechanically checked with Nqthm. The object code for these examples was generated using the Gnu C, the Verdix Ada, and the AKCL Common Lisp compilers. Perhaps most importantly, 21 of the 22 programs in the Berkeley Unix C string library were mechanically verified. See [12, 39].

In related work by Yuan Yu at DEC's Systems Research Center in Palo Alto, California, Nqthm is being used to specify part of the DEC Alpha architecture.

- **Fault Tolerance** A model of asynchronous communication was developed in Nqthm and used to prove the reliability of the biphase mark communications protocol [27]. The model transduces the waveform written by one processor into that read by an independently-clocked processor, as a function of the phases and rates of the two clocks and the communications delay. The correcntess of a gate-level design of a device implementing the biphase mark protocol has been proved [32]. The correctness of a gate-level design of a device implementing an 8-bit parallel io Byzantine agreement processor has been proved [26]. In addition, it was proved that the algorithm implemented by the Byzantine agreement processor correctly solves the "oral messages" problem [5]. Finally, the correctness of the interactive convergence clock synchronization algorithm was proved [38].

- **Scheduling, Concurrency, and Distributed Computing** Nqthm has been used to prove that an operating system implemented in machine code on a uniprocessor correctly provides multitasking and task isolation and communication [3]. An Nqthm formalization of Misra and Chandy's Unity language [14] is described in [16, 17] along with the proofs of several Unity programs. An Nqthm-checked proof that an earliest-deadline-first (EDF) scheduler is optimal is proved in [37]. The final theorem is the classic theorem about EDF schedulers.

3.3 The Role of Executability

The executability of Nqthm's logic played a significant role in the development and use of the computational models described above. We reconsider briefly several of the applications.

3.3.1 The CLI Short Stack

Consider the Piton task in the CLI short stack project [4]. The task was to design an assembly-level language for the FM8502 suitable as the target language for high-level language compilers, implement the language via an assembler and linker, and verify the correctness of the implementation mechanically. A stack-based abstract machine was chosen as the model for the resulting language, which was named "Piton." The model was formalized in Nqthm as an interpreter. In the present context this formal model may be best thought of as a 60 page system of Lisp definitions.

Having formalized the language semantics, the Piton interpreter was used to run several Piton programs on test data. The purpose of this exercise was two-fold. By hand-compiling some high level language programs and testing them we found that some useful features had been omitted from the original Piton prototype. In addition, we discovered bugs in the formal model, i.e., oversights or other errors in the formal definitions that caused the model to describe a different language than the one intended. Such testing eventually led us to conclude that Piton was specified as intended and required.

The next phase was to implement Piton on FM8502 by defining an assembler and linker. This required about 30 pages of Lisp. We then tested the implementation by using it to run the previously generated Piton programs, both on the Piton model and the FM8502 model (which, recall, was another system of executable Nqthm definitions). In some cases we ran Piton programs for thousands of FM8502 steps. We found several gross bugs in the implementation this way.

Eventually, such testing convinced us the implementation was "probably correct" and we were prepared to invest the effort in proving it. (In fact, the implementation was not correct but the bugs were hard to find, such as errors arising only if "maliciously chosen" systems of names or labels were used.) Formulating the correctness theorem required defining various mapping functions between the high level Piton state and the low level FM8502 state. Again, testing was done to verify the correspondence claimed between the states. Testing was also done to check that the statement of the main theorem held for particular Piton programs and test data.

The proof involved breaking the implementation down into steps and defining several machines intermediate between Piton and FM8502, such as a symbolic FM8502 with a program space separate from its data space. The design of these machines, the mapping functions between them, and the formal statements of the lemmas relating them were all supported by additional testing.

3.3.2 FM9001

The FM9001 project required the formalization of a model of digital circuits. The model is based on the NDL language of LSI Logic, Inc., and is essentially a formalization of a commercial NDL simulator with which one of the FM9001 designers was very familiar. The Nqthm model is a recursive function named `dual-eval` — so named because it computes either the output lines of the circuit or the final states of the various state-holding devices within it. `Dual-eval` takes among its inputs a "netlist," an Nqthm constant describing a tree of hardware modules and their interconnections via named input/output lines. The leaf nodes of these trees are primitive logical gates.

`Dual-eval` can be thought of as a logic simulator (without, however, the graphic and debugging facilities of commercial simulators). It is about 50 pages of Lisp code. Running `dual-eval` on a concrete netlist and data involves simulating in the proper sequence the input/output behavior of every logical gate in the design and is a computationally intensive activity.

The implementation of FM9001 was a concrete netlist. It was proved correct in the sense that its `dual-eval` semantics was shown to correspond appropriately to the formal model of the FM9001 as a machine code interpreter. `Dual-eval` was executed on the FM9001 netlist to help debug the netlist, the mapping functions, and the statement of the correctness theorem.

But extensive testing played a more important role later in the project, after the FM9001 had been fabricated by LSI Logic, Inc., from the verified netlist. Upon delivery of the fabricated devices, CLI tested them for conformance to the design. One reason post-fabrication testing is important is to check the reliability of the unverified layout tools involved in the lower levels of the hardware design process. Another motivation of such testing is to check for production flaws such as malfunctioning gates caused, perhaps, by material imperfections. The FM9001 was tested against its `dual-eval` model (rather than the higher level machine code model) because the `dual-eval` model contained output pins designed explicitly for testing — outputs that were not part of the machine code model.

Note that the post-fabrication testing of verified devices changes the role of model execution. Heretofore execution was merely an efficient way to avoid "premature" proof attempts. An instantaneous proof oracle would obviate the need for the kind of model execution done in the Piton project because that kind of execution was done merely in answer to a purely mathematical question: "is this conjecture always true?" or, equivalently, "do these two mathematical expressions give the same answers?" But the post-fabrication testing of a verified device is not a purely mathematical question. On the one hand one has a physical object. On the other one has a mathematical expression. The question is whether the behavior of the object is accurately predicted by the mathematical expression. The behavior of the object can only be manifested by giving it concrete data and observing its concrete output. Thus, one is forced to give concrete data to the mathematical expression and derive its concrete output. That is, one must compute with the formal model and compare the results with those produced physically. Execution of the mathematical model is inherent in post-fabrication testing. Often, at least for devices for which verification is needed, the required computations are so large and the test cases so numerous that we expect the efficiency with which the model can be executed becomes an important issue.

3.3.3 The MC68020

We conclude this discussion of executability by considering the programmer level model of the Motorola MC68020. The model is about 80 pages of Lisp and was written using the MC68020 programmer's manual [29] as the primary source. While the MC68020 can be regarded as a formal artifact in as much as, ideally, its behavior is exactly as specified by its gate-level design, that design was unavailable to the author of the Nqthm model. In effect, the Nqthm model was "reverse engineered" from the artifact.

It was necessary therefore to corroborate the model. To do so, CLI tested it against an MC68020 chip in the form of a Sun 3 workstation. Over 30,000 test vectors were executed both in the Nqthm model of the MC68020 and on an actual chip [2]. Again, the importance of the executability of the Nqthm model is obvious.

This might be considered a sorry state of affairs given the fact that the circuit design of the MC68020 can be regarded as a mathematical object (as FM9001's netlist was) and its correspondence to the machine code model is therefore subject to formal proof. But this state of affairs is common and the reasons go beyond the mere technical. Even after it becomes practical to verify such large designs, a variety of economic, legal, and other obstacles stand in the way of the publication or distribution of the "proprietary mathematical models" of commercial products by their manufacturers. The eventual promulgation of formal standards (e.g., for VHDL or Ada) can mitigate this problem for some products. But after verification has becomes more practical and cost-effective, we speculate that there will be commercial trade in "reverse engineered" formal models and those models will have been corroborated against the modeled products exactly as though those products had no other mathematical models.

3.4 Some of Nqthm's Flaws

Despite our success with Nqthm we are aware of many shortcomings. Most of these shortcomings relate to the scale of the project to be undertaken. When Nqthm was designed (primarily in the 1970's) the most impressive theorem proved by it was the uniqueness of prime factorizations, which required about 100 lines to state starting from Peano's axioms. By 1985, Nqthm was being used to prove theorems requiring 1,000 lines to state (Gödel's theorem and the correctness of the FM8501 microprocessor [18]). By 1990 it was being used to prove theorems requiring almost 8,000 lines to state (FM9001). This is almost two orders of magnitude larger than the "inspirational" theorems of Nqthm's design stage.

Below we list those flaws of Nqthm that we believed could and should be fixed by a new design.[1]

- **Prototyping Formal Models** Admitting functions to the logic requires theorem proving. But users prefer to prototype their formal models first, testing them on concrete data until convinced that the model is accurate enough to warrant the investment in proof.

- **Execution Speed** Testing a formal model on concrete data presupposes the logic is executable, i.e., that the value of a variable-free term or formula

[1] Of course, the most troubling "flaw" of Nqthm is the weakness and slowness of its theorem prover. If the theorem prover would instantly recognize any Nqthm formula that was a theorem and recognize no non-theorems, the other flaws would cause little trouble!

can be computed. Nqthm's logic is executable (unless undefined function symbols are present). But Nqthm's logic executes less efficiently than we would like. This must be understood in the context of the large models and numerous test cases discussed above. Perhaps the main reason Nqthm's logic executes slowly is the need for runtime type checking. Another is the presence of user-defined shells and other abstract objects (e.g., `(TRUE)` and `(FALSE)`) in the logic for which no corresponding data types exist in the underlying execution engine. If some Common Lisp conses must be set aside for the representation of the logic's shells, then those conses are not available for the representation of their natural counterparts in the logic. The result is that the implementations of all of the logic's constructors and accessors are complicated by runtime type checking and the provision for various "escape" mechanisms. Furthermore, because of these complications, calls to the logic's accessor functions are compiled into procedure calls rather than handled more efficiently in-line.

- **Useful Proof Techniques** There are many useful proof techniques not supported by Nqthm. Consider for example the notion of "quotient structure." Traditionally this notion refers to a set-theoretic construction in which a new structure is formed from the equivalence classes of an existing structure, for a particular equivalence relation. A typical use of such structures in our work is the partitioning of the set of states of some formalized machine into equivalence classes by a projection that ignores hidden resources. In suitable circumstances — formally described by a congruence rule — one can regard two nonidentical states as equivalent *vis-a-vis* the behavior of some higher level machine. The first-order, non-set-theoretic setting of Nqthm makes this sort of construction inconvenient for us. The identity relation (Nqthm's `EQUAL`) is the only equivalence relation with which Nqthm's heuristics will do replacement or substitution during simplification. To get Nqthm to replace a term by a nonidentical but suitably equivalent term is quite awkward, requiring one to arrange for all of the corresponding congruence and transitivity rules to be explicitly used by the simplifier during backchaining. Other proof techniques that Nqthm users have from time to time requested include forward chaining, the use of alternative definitions of a concept, and "forcing" a hypothesis to be true in back chaining by assuming its truth temporarily and bringing the full resources of the system to bear on it later, when the proof is otherwise complete.

- **User Control of the Theorem Prover** Nqthm is guided in its search for a proof by the enabled rules of its data base. Few "local" scoping mechanisms are available to the Nqthm user. The user must thus manipulate the global state of the data base in a sequential way to configure it for each theorem. This flat structure produces complicated proof scripts that are highly dependent upon an implicit notion of the current status of all rules. It also complicates the task of combining two independently developed proof scripts. Furthermore, many heuristics are beyond the user's control and arcane tricks are sometimes necessary to provoke the desired behavior. Finally, the hints available to the Nqthm user for directing the system's search explicitly are quite limited.

- **The Command Language** Nqthm's command language is Common Lisp. A strength of the arrangement is that Lisp is a powerful command language; by the suitable definition of Common Lisp macros users can tailor their command environments to their tastes. But typical commands consist of a mix of Common Lisp and Nqthm terms and formulas. This can be quite confusing, especially when dealing with function symbols such as `CAR` and `APPEND` which are defined slightly differently in the two languages. Furthermore, because the configuration of the host Lisp is not part of Nqthm, proof scripts developed by one user are not always readable by another.

- **State Saving, Reusability, and Collaboration** Nqthm allows the user to save the data base into a file, called a "library" file, and thus start a subsequent session in the same state. Nqthm libraries for many domains have been developed, e.g., natural numbers, list processing, MC68020 object code programs. But it is not possible to combine two libraries because each is a complete snapshot of the Common Lisp image. Nevertheless, such combination is exactly what one would want to do if one needed a number theoretic result while constructing the proof of an MC68020 program. This makes it quite difficult to build on the work of others or one's own past work, except by having Nqthm reprocess the old script. This raises the cost of verification by encouraging the repeated redevelopment of foundational work and tends to linearize the development of a verified system into a monolithic project by a single person who has complete knowledge of the state. Because of the growing size of verification projects, we believe that verification systems should encourage the collaborative efforts of many people working semi-autonomously on different parts of the system.

- **Practicing What We Preach** It has always struck us as hypocritical that we would, on the one hand, advocate the use of formally defined programming languages and machine-checked formal proof as a means of assuring correctness while, on the other hand, programming our own system, Nqthm, in an informally specified language and relying entirely on informal arguments that we had done our job correctly. Would it not be more convincing if we programmed the system in a formal language and could, at least, state its correctness? Should we not aspire someday to prove the correctness of the system and to somehow check that proof mechanically?

4 ACL2

ACL2 is an extended, reimplemented analogue of Nqthm that supports an extension of the applicative subset of Common Lisp as its logic. "ACL2" stands for "A Computational Logic for an Applicative Core Language." By "core language" we mean a formalism that can be used — as Nqthm was — to model many different computing systems. Common Lisp is such a language. By formalizing a logic around applicative Common Lisp we can take advantage of the exceptionally good optimizing compilers for Common Lisp to get, in many cases, execution speeds comparable to C. Two guiding tenets of the ACL2 project have been to conform to all compliant Common Lisp implementations and 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 are defined).

The definition of Common Lisp used in our work has been [35, 36]. We comment on the draft proposed ANSI standard for Common Lisp [30] in the conclusion.

Just as Nqthm is more than a verification tool for pure Lisp — in particular, it has been used as a modeling tool for a wide variety of digital systems — ACL2 is intended to be more than a verification tool for Common Lisp. Its current applications range from hardware verification to models of high level programming languages such as Ada.

4.1 Logic

The ACL2 logic is a first-order, quantifier-free logic of recursive functions providing mathematical induction on the ordinals up to ϵ_0 and two extension principles: one for recursive definition and one for constrained introduction of new function symbols, here called encapsulation.

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 `(`$f\ \ t_1 ... t_n$`)`. This formal syntax is extended by a facility for defining constants and macros. We discuss macros later.

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.

The following primitive data types are axiomatized.

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

- **Character Objects**. ACL2 supports Common Lisp's "standard characters" including `#\A`, `#\a`, `#\ `, and `#\Newline`, as well as three of Common Lisp's "nonstandard characters," `#\Page`, `#\Tab` and `#\Rubout`.

- **Strings**. ACL2 supports strings of standard characters, e.g., `"Arithmetic Overflow"`.
- **Symbols**. ACL2 supports Common Lisp's symbols. In general, symbols are objects consisting of two parts, a package and a name. The symbol `EXEC` in the package `"MC68020"` is written `MC68020::EXEC`. 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 as `MC68020::EXEC`.
- **Lists**. ACL2 supports arbitrary ordered pairs of ACL2 objects, e.g., `(X MC68020::X ("Hello." (1 . 22/7)))`.

It is our intention that 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 [35] or [36] 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 150 such symbols are axiomatized.

Common Lisp functions are partial; they are not defined for all possible inputs. Consider for example the primitive function `car`. Page 411 of [36] 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."

This raises some 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 evaluate to `nil` in some Common Lisps because the first `(car 7)` might return `t` and the second might return `nil`. The "story" relating our logic to Common Lisp is complicated and we explain it after completing the description of the logic.

In support of the "story" we formalize in ACL2 the notion of "guards." Each ACL2 function symbol has a *guard*, which is a term that specifies the domain of the function. The guard of `car` is `(or (consp x) (equal x nil))`. Applications of a function outside its guarded domain produce unspecified results. We have identified a guard for each of the Common Lisp primitives in ACL2 and made sure that our axioms do not constrain the primitives outside of their guarded domains.

To applicative Common Lisp we add four important new features.

- We add a notion of "state," containing, among other things, the file system and input/output "channels" to files. Syntactic checks in the language insure that the state is single-threaded, thus giving rise to a well-defined notion of the "current 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).

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

- We add new multiply-valued function call and return primitives that are syntactically more restrictive than those of Common Lisp (requiring a function always to return the same number of values and always to be called in the appropriate manner) but which can admit a faster implementation than Common Lisp's.

Finally, ACL2 has two extension principles: definition and encapsulation. Both preserve the consistency of the extended logic. 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 definitional principle insures 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 insures 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 nonuniqueness of the constrained functions used in the definition. The encapsulation principle preserves consistency by requiring the exhibition of witness functions that have the alleged properties.

The form of a 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." If the required termination theorems can be proved,

Axiom.
$(g\ x_1 ... x_n) \rightarrow (f\ x_1 ... x_n) = body$

is added as a new axiom. Observe that the value of the function outside of its guarded domain is unspecified. Logically, our guards are just terms that appear as hypotheses in many axioms.

Encapsulation allows the introduction of new function symbols satisfying arbitrary constraints provided one can exhibit definitions of those symbols that make those constraints theorems. This allows for abstractions to be introduced conservatively. An encapsulation command takes the form of an arbitrary sequence of commands, e.g., definitions and theorems, some of which are labeled "local." When an encapsulation command is verified for admissibility, all of the subcommands are executed and each must be successful. But the effects of the command are obtained by executing only the non-local subcommands. To constrain a new function symbol f of one argument so that it always returns a rational number, it suffices to define `(`f `x)` locally to be `22/7`, say, and then to prove and "export" the theorem `(rationalp (`f `x))`. The local definition of f is merely a witness to the consistency of the constraint. "Outside" the encapsulation, `(`f `x)` is known only to be rational.

A derived rule of inference, called "functional instantiation," [6], gives ACL2 some of the features of a higher order logic by allowing one to instantiate the function symbols of a previously proved theorem, replacing them with other function symbols or lambda-expressions, provided one can prove that the replacements satisfy the constraints on the old symbols. For example, any theorem proved about the rational f above could later be used to obtain the analogous theorem about any rational function or expression.

ACL2's logic is strictly weaker than Nqthm's because ACL2's logic does not contain an analogue of Nqthm's nonconstructive `V&C$`. The logical tendrils of `V&C$` are pervasive and the heuristics for dealing with it are very complicated (and hence invite implementation errors). Whether it is a comment on the utility of `V&C$` or (more likely) on the weakness of our heuristics for handling it, it is a fact that Nqthm users avoid `V&C$`. Little substantial use of it appears in Nqthm's benchmarks, and since the introduction of functional instantiation, some of those uses have been replaced by the use of constrained functions and functional instantiation, as illustrated in [15] where the two methods are used to prove the termination of Knuth's generalized 91-function [25]. In any case, we decided not to burden ACL2 or its users with an analogue of `V&C$`. We are optimistic that this does not seriously lessen the applicability of ACL2 to practical verification problems.

4.2 Metatheoretic Considerations

Lisp and ACL2 exploit the fact that there is a straightforward mapping from terms in the logic to the objects of the logic. For example, the term $(f\ t_1 ... t_n)$

can be identified with the list structure '(f $t_1 ... t_n$). The latter is said to be the *quotation* of the former. To make this possible we insist that if f is a function symbol in the syntax, then 'f is a symbol object. The mapping is so straightforward that one often forgets it and thinks of terms as being objects in the logic, though this is technically a categorical mistake.

One way this mapping is exploited is in the macro facility. Macros allow the syntax of the logic to be extended. Macros are functions. If f is a macro and x_1, ..., x_n are arbitrary objects, then (f $x_1 ... x_n$) may be used as a term and denotes the term (whose quotation is) obtained by applying f to the x_i. For example, by the appropriate macro definition of `case` one might arrange for `(case x (1 "a") (2 "b") (otherwise "c"))` to "macroexpand" to `(if (equal x 1) "a" (if (equal x 2) "b" "c"))`. Thus, the syntactic denotation of an expression is determined by computation and the power of recursive functions can be used (for better or worse) to introduce essentially arbitrary notation. By introducing well-designed application-specific notation one can make specifications more succinct and easy to grasp.

Like Nqthm, the ACL2 theorem prover also exploits the identification of terms and objects. It is possible to define "term transforming" functions (which actually operate on the quotations of terms). If one then proves that such a transformer preserves semantic identity (i.e., that the object denoted by the input term is the same as the object denoted by the output term), the transformation can be incorporated soundly into the simplification routines of the theorem prover. See [9].

The link between terms and objects means that some system design and user interface issues impact the choice of axioms. For example, execution efficiency dictates that the quotation of a function symbol be an object in the logic that can be concretely and uniquely represented by that symbol. Thus, while theoretical considerations permit one to encode the quotations of function symbols as integers, efficiency suggests including in the logic some suitable objects.

Nowhere is the impact of the user interface on the choice of axioms more apparent than in the provision of packages for symbol objects. Why are our symbol objects complicated by the notion of a package component? Nqthm's symbols (the `LITATOM`s) were not so complicated. The answer is that we care less that the symbol objects of the logic have packages than we care that their "dequotations" as function symbols have packages. We want to make it easy to combine theories developed by different users. Packages allow independent users to create disjoint systems of function definitions. For example, each of two users, Smith and Jones, can define the function `step`. Provided the two users work in differently named current packages, say `"SMITH"` and `"JONES"`, their two systems can be combined without logical conflict and each can continue to reference his or her own `step` by that name while referencing the other by prefixing it with the appropriate package, e.g., `JONES::step`. But because function symbols are identified with the symbol objects, `'JONES::step` must be a symbol object and the axioms must make explicit the package component of a symbol and the relationships between packages.

4.3 Execution Efficiency, Guard Verification and Colors

The implicit guards of Common Lisp allow great efficiency. The 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. Similarly, `append`, whose guard requires that the first argument be a list ending in `nil`, can recur down the `cdr` of that argument until it encounters `nil`, without type checking. Of course, if `car` or `append` is applied to `7` the results are unpredictable. There are implementations of Common Lisp, for example, Gnu Common Lisp (which used to be called Austin Kyoto Common Lisp), in which the performance of the compiled code generated for arithmetic and list processing functions is 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.

Because guards are explicit in ACL2, two choices are possible when considering how to evaluate a function call. ACL2 could check at runtime that the actuals satisfy the guard or it could glibly execute the Common Lisp function, taking a chance that anything might happen if the guard were not actually satisfied. Because we are in a logical setting, however, we could arrange also for ACL2 to do the latter only if it had proved, from the context of the function call, that the guard must be true. For example, one might show that if the guard to a given defined function is true of the formals, then every guard that will be encountered in the execution of the body of that function will also be satisfied. If one marked such functions then their safe evaluation would be fast: check the guard upon the first entry from an "unsafe" context and then execute the body with no checking (and no risk). If an entire system were so marked, the only runtime check would be of the initial inputs.

We codify this marking scheme with what we call "colors." Colors, however, are also connected to another aspect of ACL2 function definition, namely the issue of prototyping and testing before admission to the logic.

At any moment, every ACL2 function has one of four colors. The colors and their significance are described below. Colors are not part of the logic but merely a feature of our implementation of it. Facilities are provided for changing the color of a function. Functions of all colors can be compiled.

- **Red** A function symbol has the color red if the symbol has been defined for computational purposes but not admitted to the logic. When such a function is defined, syntax checks are done (to insure that the definition is in the ACL2 language) but no termination proof is done and no axiom is added. Thus, the logic remains consistent. Calls of the function can be evaluated at the command level and thus tested. The function can be freely redefined because no nontrivial results about it could have been proved or entered into the data base. Finally, the guard of a red function is ignored at runtime, so execution can cause Common Lisp errors but optimal performance is obtained. Red functions are useful for prototyping a formal model for both behavior and performance before any proof burden is incurred; red functions are also useful as utility functions (e.g., for use in macros or for data base query).

- **Pink** Pink functions are like red ones except that the guards are checked at runtime. After prototyping a system of red definitions one might convert it to pink to confirm that, on the given tests, the functions are being used in accordance with their guards. Since pink functions can be freely redefined (along with their guards), guards can be prototyped without proof burdens.

- **Blue** A function symbol is blue if the symbol has been defined for computational purposes and also admitted to the logic. To make a symbol blue, a termination proof must be done. Theorems can be proved about blue functions. Blue functions can be executed on concrete data, but guards are checked at runtime. After a function has been prototyped in a "hot" color (red or pink), its conversion to blue at the cost of termination proofs adds an axiom and thus enables one to undertake proofs of correctness and other properties.

- **Gold** A function symbol is gold if the symbol has been defined computationally and admitted to the logic (i.e., was blue) and, in addition, every subfunction called in the body is gold and theorems have been proved establishing that when the guard of the function is true the guards of every subfunction are also true on their actuals. This is called "guard verification." When a gold function is called (from outside a gold function), its guard is checked at runtime but then the body is executed without any guard checking. Thus, gold functions run as fast as red ones but with no risk of runtime error (except, possibly, resource exhaustion). After a blue formal model has been proved correct, one could undertake to make it gold by proving all of the guard conjectures. When that is done, the model will run efficiently.

Because guards are arbitrary expressions, guard verification is, in general, undecidable. But if guards are primitive type expressions on the formals, guard verification is usually a trivial theorem proving problem and can be done by special syntactic means. Some work is this direction has already been done [1]. Admissibility does not require guard verification: termination of the recursion can be proved without necessarily showing that all subfunctions are well-defined. Indeed, it is often necessary to admit a function to the logic in color blue, prove theorems about its value, and then convert it to gold.

4.4 The Story Relating the Logic to Common Lisp

We make the following claim about ACL2. Suppose f is a function symbol of one argument defined in some certified book (as described below), that the guard of f is `t`, that f is gold, and that `(equal (`f x`) t)` is a theorem of ACL2 proved in that book. Consider any Common Lisp compliant to [36] 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 ACL2 object (i.e., an ACL2 number, ACL2 character object, an ACL2 string, a symbol in some ACL2 package, or the `cons` of two ACL2 objects). 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).

Given the lack of details of the logic in this paper, it is impossible to argue the truth of this claim here. Nor have we written down a rigorous argument elsewhere at this point, though we can explain the idea here as follows. `(`f x`)` must evaluate to `t` (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), where the logic and Common Lisp agree.

This claim can be generalized considerably. The most useful generalization introduces the notion of a "gold formula" which is, roughly put, a formula in which the guards of all function symbols are true in the context in which they occur and claims, roughly, that any ACL2 instance of a gold theorem evaluates to non-`nil` in any compliant Common Lisp.

4.5 Theorem Prover

The ACL2 theorem prover is a reimplementation of the Nqthm theorem prover for the ACL2 logic. Most of the proof techniques of Nqthm have been implemented in ACL2. Many have been extended significantly.

One of the driving forces behind our design of ACL2 is that its architecture should be open so users can configure it in different ways. ACL2's proof techniques are sensitive to a hierarchically structured data base of rules derived initially from previously certified "books." Furthermore, an evolving "theory," which is computationally determined by the user as a function of the current data base and goal, specifies "views" of the data base. Books and theories are discussed later.

4.5.1 Proof Techniques

Most of the proof techniques are allowed to "force" hypotheses, by assuming them true. The theorem prover is organized so that if the main goal is proved but hypotheses are forced, the forced hypotheses are addressed in appropriate contexts in a later "round" of proof. Typically, guards are forced when they are not "obviously" true, since otherwise we cannot use the functions's definitions. This means that it is possible to submit simple (but false) conjectures about

ACL2 functions to the system and be informed, explicitly by the system, that the conjecture holds *provided* some previously unrelieved guard (or, "type") checking can be done.

While guards are forced by default, the user may actually specify which hypotheses are to be forced. By delaying the consideration of forced hypotheses ACL2 can combine them to reduce the total proof effort. For example, while a particular function call might be opened many times in different contexts, the validity of its guard might be addressed just once in a suitably general context.

Here now is a description of the ACL2 theorem prover. Readers familiar with Nqthm will recognize that system's structure and techniques.

The user's conjecture is translated via macro expansion into a formal term and converted to a set of clauses, each of which must be proved. The clauses are added to a pool of goals and extracted one at a time for further consideration by a succession of proof techniques. If an extracted goal is proved, the size of the pool shrinks by one. When the pool is empty, the user's conjecture has been proved. If a proof technique reduces the given goal to several subgoals, it puts the subgoals into the pool. Otherwise, the proof technique passes the extracted goal to the next proof technique.

The proof techniques, in order of application are:

- **preprocessing**: This process expands some propositional functions and uses `IF`-normalization, tautology checking, recognition of common cases, and equivalence closure. An ordered binary decision diagram (OBDD) algorithm has been coded for ACL2 [28] and we hope to integrate it into the preprocessor.

- **simplification**: By far the most complicated proof technique, simplification combines primitive type checking, forward chaining, backward chaining, forcing, congruence based rewriting under arbitrary equivalence relations and their refinements, generalized alternative recursive definitions, verified conditional metatheoretic simplifiers, tautology checking, congruence closure, and generalized linear arithmetic.

  Because of its importance we illustrate briefly the role of equivalence relations here. The rewriter has a "goal equivalence relation" which specifies the required relation between its input term and its output. For example, at the top-level the rewriter maintains propositional equivalence, i.e., the rewritten term must be propositionally equivalent to the input term. As the rewriter descends through the structure of the term being rewritten, it changes the goal equivalence relation according to congruence rules previously proved. Suppose for example that a `member` expression is being rewritten, and that the data base contains the rule establishing that propositionally equivalent `member` expressions are produced when `set-equal` lists are substituted into the second argument of `member`. Then when rewriting the second argument of the target `member` expression the system will allow itself to maintain `set-equal`. This means that a rule such as `(set-equal (append a b) (append b a))` can be used as a replacement rule to commute the arguments of `append`, even though `append` is not in general commutative. The user can prove new relations to be equivalence relations and prove appropriate congruence rules. Thus, even when nonunique concrete representations are used for abstract entities, the user can enjoy the simplicity of "substitution of equals for equals" at the expense of setting up the appropriate equivalence and congruence rules.

- **destructor elimination**: This process trades "bad" terms for "good" ones by a "change of variables" technique driven from the data base. For example, under suitable conditions, `(- i j)` might be reduced to `k` by replacing `i` everywhere by `(+ k j)`.
- **cross-fertilization**: Equivalence hypotheses are used and possibly discarded. Such hypotheses are used by replacing certain occurrences (depending on available congruence rules) of one side of the equivalence by the other.
- **generalization**: This process selects certain occurrences of terms involved on both sides of equivalence relations or in the hypothesis and conclusion of the goal and then replaces them with new variable symbols. Restrictive hypotheses about the new variables may be added.
- **elimination of irrelevance**: Irrelevant hypotheses are thrown out, based on variable isolation and deduced type information.
- **mathematical induction**: This process attempts to find an induction scheme appropriate for the conjecture, based on the terms in the conjecture. The analysis involves the arbitrary well-founded relations used to justify recursive functions, user supplied rules linking function symbols to additional schemes, various techniques for merging and otherwise combining schemes, and selection heuristics.

4.5.2 The Data Base, Books, Rules and Theories

A *book* is a file of definitions and theorems and references to other books whose contents are recursively included. ACL2's data base may be extended repeatedly by the "inclusion" of books. "Views" of the data base are specified by "theories," as described below.

The book mechanism is related to the encapsulation mechanism; a book can hide the details of its proofs (locally) while exporting powerful collections of rules. That is, the contents of a book appears different to its author than to its readers. All events are visible to the author, or more precisely, to the agent "certifying" the book, as discussed below. But to the reader, only "non-local" events are visible. Thus, the author might include in a book many "intermediate" theorems whose only purpose is to lead ACL2 to the proofs of the "main" theorems in the book. By marking the intermediate theorems as "local" the author can hide them from readers. Hiding is important in collaborative work since the tactical choices made by one user are often counterproductive in the context of the tactical choices of another.

Books can also define packages and declare a given package "current" for the purposes of the book, thus providing the namespace protection of packages. Books can also define theories and theory manipulation functions, as discussed below. When two books are loaded together, syntactic checks are done to insure their logical compatibility.

Books can be *certified*, which involves processing them and their subbooks so as to determine that every definition and encapsulation is admissible and every alleged theorem is provable. Certification looks at every event in a book, both the local and the non-local ones. Certificates containing details of the certification, including the checksums of the relevant books, are generated by the certification process. When a book is later included in a session only its non-local events are seen and their proofs are not reconstructed but just assumed, provided the certification data is consistent. Inconsistent certificates cause informative warnings to the user. Attention to directory structures allows books to be moved between directories on the host system without requiring recertification. The checksum computation is insensitive to comments in the file so that certified books can be documented or reformatted without requiring recertification. The certification process provides a means of version control that is integrated into the proof system. This is particularly important if multiple users are developing a system. (Strictly speaking, logical soundness requires that only one ACL2 process have write permission for the duration of a proof on all of the books involved and that certification of all books be performed by that process at the beginning of the proof. Given the fundamental insecurity of most host file systems, it was thought that checksums provide an acceptable level of assurance to detect accidental corruption by cooperative colleagues. Final formal assurance is obtained by a root-and-branch stand-alone certification at the end of the project.)

As with Nqthm, proved theorems generate rules that are added to the data base. Unlike Nqthm, ACL2 does not require the syntactic form of a theorem to specify how it will be used as a rule. Instead, every rule generated from a theorem is derived from a "corollary" formula, optionally specified by the user (and defaulting to the theorem itself), which is implied by the theorem and whose syntax describes the rule.

Including a book into an ACL2 session adds all the rules derived from the non-local events in the book. Each rule in ACL2 has a unique name. A rule can be used only if it is "enabled." Whenever ACL2 goes to the data base to obtain information an appropriate rule name is found, checked for being enabled, and tracked in a "tag tree," a structure that follows the evolving proof construction and records relevant information. Pervasive tracking and use of enabled rule names provides the ACL2 user with much finer grained control of ACL2 as well as more information, if desired, about the evolving proof.

To be "enabled" a name must be in the current theory, where a *theory* is a list of rule names. A theory thus gives a view of the data base in the sense that a rule is seen only if its name is in the current theory.

Theories are just objects in the logic, namely lists. Since a theory is just a list of ACL2 objects, theories can be computed and manipulated by ACL2 functions. Utility functions for manipulating theories include functions for unioning together theories (enabling all the rules in each), taking the set difference of two theories (disabling all the rules in the latter theory), and computing the current theory as of some previous proof. The user may define theory manipulation functions. Daemons can be installed on names to insure that incompatible rules do not coexist within a derived theory. Books may provide alternative sets of rules and make them conveniently available via theories and theory functions provided in the books. The author's advice about how rules are used in concert may be codified into daemons.

4.5.3 Proof Trees and Commentary

The ACL2 theorem prover prints a running commentary on its evolving proof attempt, explaining each transformation, the derivation of forced hypotheses, the use of hints, etc. In addition, if Emacs is available, ACL2 sketches the evolving proof tree as it goes, pruning branches that are proved. The proof tree facility is linked to the evolving commentary so that the commentary associated with any point in the tree can be conveniently obtained. This allows the user to ignore ACL2's scrolling commentary (indeed, many users simply do not display that buffer) and simply jump directly to the trouble spots (the "checkpoints" of [11] and of the tool described in [23]).

4.5.4 Documentation

ACL2 is documented via an online documentation facility that is part of ACL2. The ACL2 documentation is maintained in a hypertext-like structure which may be browsed via ACL2 documentation commands. In addition, it may be browsed via Emacs' Info mode and via Mosaic. Roughly .9 megabytes of online documentation is available about ACL2, including tutorials. Instructional materials are being prepared as part of this documentation.

The ACL2 user may wish to document his or her formal models and browse that documentation with the facilities provided. ACL2 supports this use of its documentation facilities. In particular, Common Lisp allows documentation strings to be included in the definitions of constants, macros and functions and ACL2 extends that to the commands that introduce other names, such as theorems and theories. ACL2 recognizes documentation strings that are formatted in a certain way and links them into its browser. Facilities are provided to convert ACL2's graph into files suitable for browsing with Info mode and Mosaic. Thus, when a book is included in a session — possibly introducing many new function symbols and rules — the author's documentation of those new names also becomes available.

4.5.5 The User Interface

ACL2 presents itself to the user as a read-eval-print loop in which ACL2 forms are read and evaluated. Upon reflection, this is a dizzying statement, because it implies that the command to define a function, for example, is an ACL2 form. We discuss this in the next section.

We here focus on three interesting aspects of this interface. First, as with traditional Common Lisp interfaces, the user may freely mix the definition of functions with their evaluation, typing only Common Lisp. Indeed, the only language one must know to use ACL2 is ACL2 (applicative Common Lisp). We find this unity pleasing. Indeed, the aspects of ACL2 relating to logic and proof are transparent as long as one is merely prototyping and testing a red system. A window based interface could be engineered from this base with about the same difficulty as one could be engineered for a traditional Common Lisp.

Second, because the command language is ACL2, the user can define macros to tailor the command environment. Here are three illustrative command-level macros.

- A macro form might automatically "disable" all the rules generated by a theorem embedded in the form.
- A macro form might generate schematically a collection of theorems from some data structure provided in the form. Such a macro might implement some specialized methodology for proving certain kinds of properties about the object model.
- A macro form might submit a given theorem for proof and compute automatically a certain style hint known by the user to be necessary when theorems of the given kind are proved.

In Nqthm, such macros were defined in Common Lisp. But proof scripts using those macros were unreadable by unmodified Nqthm images. In ACL2, such macros may be included in the proof scripts in which they are used (perhaps by the recursive inclusion of a customization book). Checks made when books are loaded insure that the macros used by a collection of books are compatible. It is our intention that the provision of macros at the command level will give ACL2 an advantage over Nqthm when using the system as a "shell" in which to construct verification environments for specialized domains precisely because it allows one to codify, formally, the methodology required.

Third, ACL2 can be used to query the data base. Sophisticated users of Nqthm frequently use this aspect of Nqthm, as to collect all the names defined since a given name was introduced, etc. But in Nqthm the use of Common Lisp can potentially render the system unsound, as would happen if the user redefined the Common Lisp function `prove` or destructively modified the property lists while inspecting them. But in ACL2, such utility functions can be defined without having to program in a new language and without risking damage to the system, because of ACL2's applicative nature and the checks made by its definitional principle. Indeed, users can define useful utilities and exchange them via books, without rendering the system unsound.

4.6 Practicing What We Preach

ACL2 is coded in ACL2. That is, the ACL2 system is a collection of books containing definitions of ACL2 functions. The type mechanism, the rewriter, the linear arithmetic procedure, the induction heuristic, the error checkers, the error handlers, the mechanism for reading and processing books, the top-level read-eval-print loop — all are ACL2 functions. To boot the system, the compiled code is loaded into any Common Lisp and then the system reads and processes its own source files, first in the color red and then gold. (At the moment, only part of the system, about a tenth of it, has been processed in gold. Among our goals is to process the entire system in gold.) The system currently stands at 4.9 megabytes (or about 109,000 lines of code), making it one of the largest applicative programs in the world.

The discipline of using ACL2 as its own implementation language has had a remarkable impact on the language, the theorem prover, and the system support tools. Simply expressing ACL2 in itself stretched the applicative language from the rather confining natural subset of Common Lisp to a practical applicative programming language. Our concern for the efficiency of our software drove us to make ACL2 efficient. Array access and change are essentially constant-time operations, that operate at about half the speed of the array access and change in C. Appropriately `declare`d simple arithmetic expressions execute at C speeds. However, unlike other practical applicative languages, ACL2 includes an axiomatization and a mechanical theorem proving environment.

The discipline of processing our source code with the definitional principle has yielded important improvements to the heuristics originally taken from Nqthm. For example, one 700 line definition in our source code expanded under Nqthm's definition normalization routines to 25 megabytes of formal code. ACL2 does not normalize definitions, which on the one hand forces its theorem prover to do so but on the other allows context-sensitive normalization to reduce the number of cases. Whether ACL2 can manage to prove things about definitions of practical size (e.g., its own source code) remains to be seen; but the experiment has at least established that the heuristics of Nqthm could not.

By "practicing what we preach" we have been forced to confront in ACL2 the same problems of scale, e.g., efficiency, problem size, documentation, version control, distributed development, etc., that Nqthm users were beginning to confront. Furthermore, solving these problems has been on our "critical path."

Some of the other side-effects of our choice of ACL2 as implementation language are noted briefly below.

- Because it is written in that subset of applicative Common Lisp that is host-independent, ACL2 is portable to any compliant Common Lisp platform.
- The ACL2 state, and in particular the data base, is an ACL2 object that may be inspected by ACL2 functions and the user. It is possible to save versions of the data base for later retrieval and otherwise take advantage of the "first class" nature of the data base. Theory manipulation functions get the data base as an argument. Metafunctions could in principle be sensitive to the data base.
- Metafunctions can be coded more efficiently.
- It is possible for ACL2 to reason about its own behavior since its source code is among the axioms. So far, only trivial use of this has been made, namely, to verify the guards of hundreds of ACL2 system functions. (It should be noted that until ACL2 constructs independently checked formal proof objects — the direction in which our tag tree mechanism is headed — a proof by ACL2 about its own source code has to be regarded with the same skepticism one is inclined toward when someone says "I would never lie to you.")
- It is possible to state in ACL2 that the system is sound.
- It is in principle possible to prove that ACL2 is sound. This is among our long-term goals.

5 The Associativity of Append

In this section we exhibit a Common Lisp definition of list concatenation and prove that it is associative. We avoid using the built-in function `append` because it is defined as a macro that takes two or more arguments.

The function `app` below concatenates two lists. However, the first argument must be a `true-listp`, which is to say, it must end in `nil`. Note the declared `:guard` below. This allows the definition of `app` to terminate with a `null` check, which is more efficient than the type check `atom`.

```
(defun app (x y)
  (declare (xargs :guard (true-listp x)))
```

```
  (cond ((null x) y)
        (t (cons (car x) (app (cdr x) y)))))
```

This function is associative provided its guard is satisfied. Here is a statement of the theorem:

```
(implies (and (true-listp a)
              (true-listp b))
         (equal (app (app a b) c) (app a (app b c)))).
```

The proof of this theorem, constructed by the current version of ACL2 (Version 1.6), is shown below. The Common Lisp reader generally converts lower case input to upper case (except in strings and in certain delimited symbols) and thus the formulas printed below are in upper case.

Of special interest is the "forcing round." The main proof proceeds by an induction on `A`. The term `(APP (APP A B) C)` in the induction conclusion expands, using the definition of `APP` on the innermost `APP` term, to `(APP (CONS (CAR A) (APP (CDR A) B)) C)`. This expansion is permitted because the guard for that term, `(TRUE-LISTP A)`, is known to be true by hypothesis. But to expand the outermost `APP` in `(APP (CONS (CAR A) (APP (CDR A) B)) C)` we must know that its first argument is a `TRUE-LISTP` and this is problematic because it involves an inductive argument about the innermost `APP`. But the proof proceeds by "forcing" this guard (in `Subgoal *1/3'` below). At the successful conclusion of the main proof, ACL2 undertakes a "Forcing Round" to show, by induction, that under the given hypotheses the innermost `APP` term returns a `TRUE-LISTP`.

```
Name the formula above *1.

Perhaps we can prove *1 by induction.  Five induction schemes are suggested
by this conjecture.  Subsumption reduces that number to four.  These
merge into two derived induction schemes.  However, one of these is
flawed and so we are left with one viable candidate.

We will induct according to a scheme suggested by (APP A B).  If we
let  (:P A B C) denote *1 above then the induction scheme we'll use
is
(AND (IMPLIES (NOT (TRUE-LISTP A))
              (:P A B C))
     (IMPLIES (AND (TRUE-LISTP A)
                   (NOT (NULL A))
                   (:P (CDR A) B C))
              (:P A B C))
     (IMPLIES (AND (TRUE-LISTP A) (NULL A))
              (:P A B C))).
This induction is justified by the same argument used to admit APP,
namely, the measure (ACL2-COUNT A) is decreasing according to the relation
E0-ORD-< (which is known to be well-founded on the domain recognized
by E0-ORDINALP).  When applied to the goal at hand the above induction
scheme produces the following three nontautological subgoals.
```

```
Subgoal *1/3
(IMPLIES (AND (TRUE-LISTP A)
              (NOT (NULL A))
              (EQUAL (APP (APP (CDR A) B) C)
                     (APP (CDR A) (APP B C)))
              (TRUE-LISTP B))
         (EQUAL (APP (APP A B) C)
                (APP A (APP B C)))).

By the simple :definition of NULL we reduce the conjecture to

Subgoal *1/3'
(IMPLIES (AND (TRUE-LISTP A)
              (NOT (EQUAL A NIL))
              (EQUAL (APP (APP (CDR A) B) C)
                     (APP (CDR A) (APP B C)))
              (TRUE-LISTP B))
         (EQUAL (APP (APP A B) C)
                (APP A (APP B C)))).

But forced simplification reduces this to T, using primitive type reasoning,
the :rewrite rules CDR-CONS and CAR-CONS and the :definitions of TRUE-
LISTP and APP (forced).

Subgoal *1/2
(IMPLIES (AND (TRUE-LISTP A)
              (NOT (NULL A))
              (NOT (TRUE-LISTP (CDR A)))
              (TRUE-LISTP B))
         (EQUAL (APP (APP A B) C)
                (APP A (APP B C)))).

But we reduce the conjecture to T, by primitive type reasoning.

Subgoal *1/1
(IMPLIES (AND (TRUE-LISTP A)
              (NULL A)
              (TRUE-LISTP B))
         (EQUAL (APP (APP A B) C)
                (APP A (APP B C)))).

By the simple :definition of NULL we reduce the conjecture to

Subgoal *1/1'
(IMPLIES (AND (TRUE-LISTP A)
              (NOT A)
              (TRUE-LISTP B))
         (EQUAL (APP (APP A B) C)
                (APP A (APP B C)))).

But simplification reduces this to T, using primitive type reasoning,
the :definition of APP and the :executable-counterparts of TRUE-LISTP,
NOT and EQUAL.
```

```
That completes the proof of *1.

q.e.d. (given one forced hypothesis)

Modulo the following forced goal, that completes the proof of the input
Goal.  See :DOC forcing-round.

[1]Goal, below, will focus on
(TRUE-LISTP (APP (CDR A) B)),
which was forced in
 Subgoal *1/3', above,
  by applying (:DEFINITION APP) to
  (APP (CONS (CAR A) (APP (CDR A) B)) C).

We now undertake Forcing Round 1.

[1]Goal
(IMPLIES (AND (CONSP A)
              (TRUE-LISTP (CDR A))
              (TRUE-LISTP B))
         (TRUE-LISTP (APP (CDR A) B))).

The destructor terms (CAR A) and (CDR A) can be eliminated by using
CAR-CDR-ELIM to replace A by (CONS A1 A2), generalizing (CAR A) to
A1 and (CDR A) to A2.  This produces the following goal.

[1]Goal'
(IMPLIES (AND (CONSP (CONS A1 A2))
              (TRUE-LISTP A2)
              (TRUE-LISTP B))
         (TRUE-LISTP (APP A2 B))).

This simplifies, using primitive type reasoning, to

[1]Goal''
(IMPLIES (AND (TRUE-LISTP A2) (TRUE-LISTP B))
         (TRUE-LISTP (APP A2 B))).

Name the formula above [1]*1.

Perhaps we can prove [1]*1 by induction.  Three induction schemes are
suggested by this conjecture.  These merge into two derived induction
schemes.  However, one of these is flawed and so we are left with one
viable candidate.

We will induct according to a scheme suggested by (APP A2 B).  If we
let  (:P A2 B) denote [1]*1 above then the induction scheme we'll use
is
(AND (IMPLIES (NOT (TRUE-LISTP A2))
              (:P A2 B))
     (IMPLIES (AND (TRUE-LISTP A2)
                   (NOT (NULL A2))
                   (:P (CDR A2) B))
              (:P A2 B))
     (IMPLIES (AND (TRUE-LISTP A2) (NULL A2))
```

```
              (:P A2 B))).
This induction is justified by the same argument used to admit APP,
namely, the measure (ACL2-COUNT A2) is decreasing according to the
relation E0-ORD-< (which is known to be well-founded on the domain
recognized by E0-ORDINALP).  When applied to the goal at hand the above
induction scheme produces the following three nontautological subgoals.

[1]Subgoal *1/3
(IMPLIES (AND (TRUE-LISTP A2)
              (NOT (NULL A2))
              (TRUE-LISTP (APP (CDR A2) B))
              (TRUE-LISTP B))
         (TRUE-LISTP (APP A2 B))).

By the simple :definition of NULL we reduce the conjecture to

[1]Subgoal *1/3'
(IMPLIES (AND (TRUE-LISTP A2)
              (NOT (EQUAL A2 NIL))
              (TRUE-LISTP (APP (CDR A2) B))
              (TRUE-LISTP B))
         (TRUE-LISTP (APP A2 B))).

But simplification reduces this to T, using the :definitions of TRUE-
LISTP and APP and primitive type reasoning.

[1]Subgoal *1/2
(IMPLIES (AND (TRUE-LISTP A2)
              (NOT (NULL A2))
              (NOT (TRUE-LISTP (CDR A2)))
              (TRUE-LISTP B))
         (TRUE-LISTP (APP A2 B))).

But we reduce the conjecture to T, by primitive type reasoning.

[1]Subgoal *1/1
(IMPLIES (AND (TRUE-LISTP A2)
              (NULL A2)
              (TRUE-LISTP B))
         (TRUE-LISTP (APP A2 B))).

By the simple :definition of NULL we reduce the conjecture to

[1]Subgoal *1/1'
(IMPLIES (AND (TRUE-LISTP A2)
              (NOT A2)
              (TRUE-LISTP B))
         (TRUE-LISTP (APP A2 B))).

But simplification reduces this to T, using the :definition of APP
and the :executable-counterparts of TRUE-LISTP, NOT and EQUAL.

That completes the proof of [1]*1.

Q.E.D.
```

```
Summary
Form:  ( THM ...)
Rules: ((:DEFINITION IMPLIES)
        (:REWRITE CDR-CONS)
        (:REWRITE CAR-CONS)
        (:ELIM CAR-CDR-ELIM)
        (:DEFINITION TRUE-LISTP)
        (:FAKE-RUNE-FOR-TYPE-SET NIL)
        (:DEFINITION NULL)
        (:DEFINITION NOT)
        (:EXECUTABLE-COUNTERPART TRUE-LISTP)
        (:EXECUTABLE-COUNTERPART NOT)
        (:EXECUTABLE-COUNTERPART EQUAL)
        (:DEFINITION APP))
Warnings:  None
Time:  1.60 seconds (prove: 0.80, print: 0.43, proof tree: 0.28, other: 0.08)
```

6 Applications of ACL2

We list briefly some of the ongoing applications of ACL2. Most of these projects are still in the formalization stage and have not yet gotten to the most significant proofs. However, this collection of ACL2 projects already includes over 1,500 theorems proved, not counting those proofs performed during the admission of the blue and gold definitions in the systems described below.

- The ACL2 system is being built in ACL2. The construction involves termination proofs and guard verification of ACL2 functions. The source code now stands at 4.9 megabytes of ACL2.

- The Nqthm package is being developed. Roughly speaking, this is a book that embeds the Nqthm logic into the ACL2 logic and causes ACL2's theorem prover to emulate Nqthm's. With the exception of Nqthm's `V&C$` and its superiors, all primitive Nqthm functions are now defined within the `"NQTHM"` package and the relevant Nqthm axioms about them have been proved as ACL2 theorems by ACL2. For example after appropriately defining such symbols as `NQTHM::CAR`, the formula `(IMPLIES (NOT (LISTP X)) (EQUAL (CAR X) 0))` is an ACL2 theorem, provided it is read while in the current package `"NQTHM"`. This is one of the axioms of Nqthm. The Nqthm definitional principle and shell principle have been implemented. A book of rules causing the ACL2 theorem prover to emulate the Nqthm theorem prover is being developed. It is hoped that with this book most of the 11 megabytes of Nqthm benchmarks will "replay" automatically in ACL2. This project is still under way but results so far are promising. We have processed roughly 75% of the file `/nqthm-1992/examples/basic/proveall.events`, which includes such theorems as the correctness of a simple optimizing expression compiler, Euclid's theorem, and the soundness of a tautology checker. We expect the Acl2 emulation of Nqthm to result in an overall degradation of performance (compared to both Nqthm's proof engine and its execution environment). However, the Nqthm package will provide a migration pathway for Nqthm users to ACL2. Furthermore, the Nqthm package will provide an important collection of benchmark theorems on which we can tune Acl2 performance.

- Many low-level books are being developed with the expectation that they will find widespread use. Among them are books for natural, integer, modulo, rational and complex arithmetic, groups, hardware specifications, and metatheoretic reasoning.

- The Nqthm work related to the Motorola MC68020 is being recast into ACL2. In particular, an ACL2 definition of the object code interpreter is being developed from the Nqthm model and the Nqthm library used in the program correctness proofs will be developed as an ACL2 book. Initially, our aim is to reproduce the Nqthm proofs with Acl2 — a goal that could perhaps be achieved more easily via the Nqthm package. But ultimately we hope to tune to ACL2 model so that the simulation of object code programs is faster than via the Nqthm model or its emulation in the Nqthm package.

- The semantics of a subset of Ada is being coded in ACL2 in the form of an Ada interpreter. Some simple Ada programs have been verified with respect to this semantics. A book of useful Ada rules is being developed. The intent of these and other "high-level" books about a given subject formalism (in this case Ada) is to configure ACL2 so that the proofs about objects in the subject formalism (here, Ada programs) are straightforward when certain paradigms are followed.

- A top-level specification of a proprietary Motorola digital signal processing (DSP) microprocessor is being developed. When complete, the specification will be used to prove the correctness of some DSP algorithms and to develop a high-level book about the processor. This work is analogous to the Nqthm work on the MC68020 and the C string library.

- The semantics of VHDL is being coded in ACL2 in the form of a VHDL interpreter or simulator. When completed it will be, in principle, possible to prove theorems about the behavior of VHDL systems. Because of the size and complexity of VHDL, such proofs will be an interesting challenge to ACL2 and its users.

7 Conclusions and Criticisms

The ACL2 project is now five years old. The system has not been released because we are still finding bugs in it and we are not happy yet with its documentation. Within CLI, ACL2 is used more than Nqthm, although this is probably more due to social reasons than technical ones (one hears more talk about ACL2 than Nqthm around the coffee pot). ACL2 now has about a dozen users, all of whom have experience with Nqthm.

We have many concerns about ACL2's viability.

7.1 Guards

Perhaps the most pervasive concern is the feeling that guards are not yet adequately handled. Guard checking is slow because so many common cases are handled by full-blown theorem proving rather than fast syntactic checkers. More problematic is that guards complicate the statement of theorems. Nqthm's logic gets incredible mileage out of the notion that functions — especially arithmetic functions — default "unexpected" input to reasonable values so that many theorems are stated without hypotheses. In ACL2 one must be careful to restrict every variable appropriately so that the guards of all functions are satisfied. (Macros can be written to supply restrictive hypotheses based on variable names, so the syntactic burden is not the issue.) For example, Common Lisp makes no guarantee that `(+ i j)` is `(+ j i)` unless both `i` and `j` are numbers. This means not only that theorems are more cumbersome to state but rules — especially rewrite rules and definitions — are more restrictive, cause more backchaining and fail to apply more often. A consideration is that sometimes rules fail to apply even though the hypotheses are true, because the system is too weak to establish their truth without additional help.

In the original version of ACL2 we made no special provisions for guards and found that many rules, especially definitions, could not be applied under Nqthm's heuristics because hypotheses (namely guards) could not be relieved at the time they were needed. This is illustrated by the proof above of the associativity of `APP`: the hypothesis required induction to prove.

Our first attempt to handle guards specially was suggested by Nqthm's handling of guard-like hypotheses in its linear arithmetic decision procedure, where a new case split is introduced whenever the procedures "needs" a hypothesis it cannot establish. With ACL2 this generated proofs that were hard to follow because of the numerous and apparently spontaneous case splits. While this may seem like mere carping, the effect was quite deleterious on the user's ability to "debug" a proof attempt and guide the system to successful proofs.

In addition, it is important to realize that most proof attempts fail because the goal conjecture is not a theorem. This happens (in Nqthm, in ACL2, and, we suspect, in most mechanized theorem proving systems used for similar tasks) because the user is still grappling with the modeling and formalization problems.

The guard-generated case splits, even those eventually dispatched successfully, merely delayed the user's discovery of the "real" reason the proof attempt failed.

The introduction of "forcing rounds" was our second attempt to try to mitigate the problem of guards. Unlike the approach in our first attempt, forcing rounds delay the consideration of the "type-like" guard conjectures until after the "gist" of the proof has been successfully done. Forcing rounds were regarded by the users as an improvement over the earlier scheme. But users still remember fondly the Nqthm days when such details simply did not arise. Of course, guards allow execution efficiency. Therefore, the trade-off is whether the added complexity in proofs is worth the speedup in the simulation speeds of large formal models. The results are not yet in because we have not completed suitably large scale experiments yet.

When we are discouraged about the ability of the system to handle guards, we are sometimes tempted to change slightly the story relating ACL2 to Common Lisp. Rather than maintain that ACL2 functions are undefined outside their guarded domains we could define them explicitly as is done in Nqthm's logic. The result would be a particular implementation of a Common Lisp, which we sometimes refer to as "completed Common Lisp." In completed Common Lisp functions would coerce unexpected arguments to natural values. Thus, for example, arithmetic functions would coerce non-numeric arguments to `0`. This would simplify many axioms and theorems. Definitions could be used unconditionally. Some unusual theorems would hold, e.g., `(equal (car 7) nil)`, that would surprise some Common Lisp programmers. But the story relating the completed Common Lisp logic to Common Lisp would be the same as it is now: gold theorems are true in all compliant Common Lisps. We are reluctant to go this route just now, choosing instead to proceed first by evaluating ACL2 on big examples.

Perhaps a natural question at this point is: "Why don't the Acl2 implementors do away with this notion of guards and instead take a more standard approach to the same issue, i.e., using some kind of decidable type checking?" Our most important reason: we have already decided to stay compliant with the Common Lisp language, and we are not aware of any simple way of inventing a decidable type system that lets us do this in a reasonable fashion. Also, we are encouraged by the progress we have made in the handling of guards (specifically, in the success of the "forcing round" technique), and users have begun considering their use in specifications. If guards are found to be useful this way, they will enjoy a decided advantage over decidable type systems that we have seen, because they are as expressive as the Common Lisp language itself: any predicate may be used as a guard. Most decidable type systems do not even allow types other than conjunctions of calls of unary predicates! At any rate, even if we ultimately feel that we must give up the present notion of guards, the "completed Common Lisp" idea discussed above provides a route for eliminating much of the proof burden introduced by guards, without eliminating the connection to Common Lisp or the ability to use guards as a specification device.

7.2 Draft Proposed ANSI Standard

We are aware of one area in which ACL2 is at odds with the draft proposed ANSI standard for Common Lisp [30] and that is in connection with the "generalized Boolean" functions of the proposed standard. In ACL2, `equal`, for example, is a Boolean valued function, meaning that it returns either `nil` or `t`. But in [30] `equal` is a "generalized Boolean" valued function, meaning that it may return any non-`nil` value to indicate truth. Many functions that return Booleans in traditional Lisp implementations return generalized Booleans in [30], including `equal`, `<`, `symbolp`, and `subsetp`. Apparently programs are not to use such functions in contexts other than those in which their values are tested as propositions.

As ACL2 now stands there are gold theorems that are not true of the language described in [30]. One example is `(equal (equal nil nil) (equal nil nil))`. In ACL2, both of the interior `equal` terms return `t` and so the theorem follows from the axiom `(equal x x)`. But in [30] the first `(equal nil nil)` might return `t` and the second might return `23` and the two are not `equal`. This formula violates the spirit of [30]: generalized Booleans are to be used only in propositional tests. To conform to [30] we might introduce a new attribute of function symbols, namely whether they have generalized Boolean values, and narrow the class of gold formulas to exclude those where generalized Booleans are "misused."

7.3 The Complexity of ACL2

The proliferation of ACL2 rule classes may be a mistake. For example, the current ACL2 supports rewrite rules, linear rules, linear alias rules, well-founded relation rules, built-in clause rules, compound recognizer rules, destructor elimination rules, generalization rules, meta rules, forward chaining rules, equivalence rules, refinement rules, congruence rules, type-prescription rules, alternative definition rules, induction rules, and type-set inverter rules. This is an aspect of ACL2's open architecture: it is easy to program ACL2's theorem prover. But it is easy to program it very inefficiently. Moreover, the plethora of rule classes can be intimidating to new users (though we expect that improved, "layered" documentation may help in this regard).

Programming disciplines (perhaps augmented by some additional heuristics) need to be developed so that users can create rule sets that are effective and efficient. We may find the plethora of rule classes simply too complicated for the development of adequate disciplines and be forced to abandon some of these. More likely, we may develop disciplines that essentially recommend that only "experts" employ all but the few well-understood rule classes.

More generally, we are concerned about the intellectual complexity of the ACL2 proof engine and environment. We find its applicative definition wonderfully clear, but 4.9 megabytes of source code — even very clear source code — is difficult to keep in mind. An ACL2 image is about 30 megabytes. We estimate that about 20 of that is devoted to the initial property list world which contains the results of processing the 4.9 megabytes of ACL2 source code. For example, one of the definitional axioms of ACL2 is the formula that equates `(prove term pspv hints wrld ctx state)` to its body, i.e., the initial axioms include the definition of our heuristic theorem prover. Along these same lines, it is possible to inspect the initial data base and see the definition of the rewriter with its 16 formal parameters and its 30 mutually recursive entries. There is a lot of complexity here.

While soundness is, of course, an issue, it is not the main obstacle. Ultimately, soundness could be be insured, we hope, by having the system generate proof objects which are checked. Our concern is that if the system is too complex it will be impossible for us to coordinate its various parts so that they operate harmoniously, i.e., so that it discovers any proofs at all. It would be very reassuring to us to see ACL2 reproduce the proofs in Nqthm's benchmark files, simply as evidence that ACL2's abundance of proof techniques interoperate harmoniously "enough."

7.4 Performance

Finally, ACL2 feels sluggish. One must, of course, ask "Compared to what?" Since all of its users are former Nqthm users, the answer is that ACL2 feels sluggish compared to Nqthm.

One possible explanation is that ACL2's performance is degraded by its applicative implementation. We believe this conjecture is false. Experiments with isolated code fragments indicate that the applicative expression of Nqthm's algorithms generally execute as fast or faster than their Nqthm implementation. This is not surprising: Nqthm's coding style is heavily influenced by its previous expression in Interlisp where the performance trade-offs between global variables, special variables, and local variables were different than in Common Lisp. We expect that if we were to recode Nqthm applicatively it would speed up by perhaps as much as 10%.

But ACL2 is not such a recoding of Nqthm: its logic is much more complicated and its heuristics are sometimes different. The differences in the logic make it difficult even to present the two systems with the "same problem." To do so one must embed the relevant fragment of one logic in the other and prove the rules necessary to cause one system to emulate the other. But even when that is done, the proofs generated for identical problems sometimes diverge because of the heuristic differences in the two systems.

These considerations make it very difficult to produce a meaningful quantitative comparison of the two systems. Rather than pursue comparisons then it is our intention to try to speed up ACL2. We know that its applicative printing functions are quite slow compared to the Common Lisp primitives; speeding up these functions would make ACL2 feel faster. But the performance of the theorem prover can also be improved, both by reconsidering some heuristics and by recoding some poor implementations of individual functions. This kind of tuning, of which Nqthm has had a great deal, comes only after a very large collection of benchmark theorems is available for test purposes. Therefore, we are inclined at this stage simply to use ACL2 (and to ignore, for the time being, our qualms about its performance). The Nqthm package, which will make Nqthm's benchmark files available to ACL2, is especially important for performance tuning, as it will allow us to compare various "tunings" of ACL2 on thousands of theorems.

References

[1] R. L. Akers. Strong Static Type Checking for Functional Common Lisp. Ph.D. Thesis, University of Texas at Austin, 1993. Also available through Computational Logic, Inc., Suite 290, 1717 West Sixth Street, Austin, TX 78703.

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

[3] W. R. Bevier. A Verified Operating System Kernel. Ph.D. Thesis, University of Texas at Austin, 1987. Also available through Computational Logic, Inc., Suite 290, 1717 West Sixth Street, Austin, TX 78703.

[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, 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." (Submitted.)

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

[9] R. S. Boyer and J S. Moore. Metafunctions: Proving Them Correct and Using Them Efficiently as New Proof Procedures. In R. S. Boyer and J S. Moore, editors, *The Correctness Problem in Computer Science*, Academic Press, 1981.

[10] R. S. Boyer and J S. Moore. A Mechanical Proof of the Turing Completeness of Pure Lisp. In W. W. Bledsoe and D. W. Loveland, editors, *Automated Theorem Proving: After 25 Years*, American Mathematical Society, Providence, R.I., 1984, pp. 133-167.

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

[12] 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.

[13] B. C. Brock, W. A. Hunt, Jr. and W. D. Young. Introduction to a Formally Defined Hardware Description Language. In *Theorem Provers in Circuit Designs*, Number A10 in IFIP Transactions. North Holland, 1992.

[14] K. M. Chandy and J. Misra. *Parallel Program Design: A Foundation.* Addison-Wesley: Massachusetts, 1988.

[15] J. Cowles. Meeting a Challenge of Knuth. Internal Note 286, Computational Logic, Inc., Austin, Texas, September, 1993.

[16] D. M. Goldschlag. Mechanizing Unity. In M. Broy and C. B. Jones, editors, *Programming Concepts and Methods*, North Holland, Amsterdam, 1990.

[17] D. M. Goldschlag. Mechanically Verifying Concurrent Programs with the Boyer-Moore Prover. *IEEE Transactions on Software Engineering*, 16, September, 1990.

[18] W. A. Hunt, Jr. FM8501: A Verified Microprocessor, Ph.D. Thesis, The University of Texas at Austin, December, 1985. Also available through Computational Logic, Inc., as Technical Report ICSCA-CMP-47, Institute for Computing Science and Computer Applications, University of Texas at Austin, December, 1985.

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

[20] M. Kaufmann. A User's Manual for an Interactive Enhancement to the Boyer-Moore Theorem Prover, Technical report 19, Computational Logic, Inc., May, 1988.

[21] M. Kaufmann. Addition of Free Variables to an Interactive Enchancement of the Boyer-Moore Theorem Prover, Technical Report 42, Computational Logic, Inc., 1990.

[22] M. Kaufmann. An extension of the Boyer-Moore theorem prover to support first-order quantification. *Journal of Automated Reasoning*, 9(3):355–372, December 1992.

[23] M. Kaufmann. An Assistant for Reading Nqthm Proof Output. Technical Report 85, Computational Logic, Inc., November, 1992.

[24] M. Kaufmann and P. Pecchiari. Interaction with the Boyer-Moore Theorem Prover: A Tutorial Study Using the Arithmetic-Geometric Mean Theorem, Technical Report 102, Computational Logic, Inc., 1994.

[25] D. E. Knuth. Textbook examples of recursion. In V. Lifschitz, editor, *Artificial Intelligence and Mathematical Theory of Computation: Papers in Honor of John McCarthy*, Academic Press, San Diego, CA, 207–229, 1991.

[26] J S. Moore. Verified Hardware Implementing an 8-Bit Parallel I\O Byzantine Agreement Processor. Technical Report 69, Computational Logic, Inc., Austin, Texas, August, 1991.

[27] 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.

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

[29] Motorola, Inc. *MC68020 32-bit Microprocessor User's Manual.* Prentice Hall, New Jersey, 1989.

[30] 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.

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

[32] D. M. Russinoff. Specification and Verification of Gate-Level VHDL Models of Synchronous and Asynchronous Circuits. Technical Report 99, Computational Logic, Inc., Austin, Texas, May, 1994.

[33] N. Shankar. A Mechanical Proof of the Church-Rosser Theorem. *JACM* 35(3), 475–522, 1988.

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

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

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

[37] M. Wilding. An optimal real-time scheduler based on a simple model of computation. Internal Note 276, Computational Logic, Inc., July 1993.

[38] W.D. Young. Verifying the Interactive Convergence Clock Synchronization Algorithm using the Boyer-Moore Theorem Prover. Contractor Report 189649, NASA, April 1992.

[39] Y. Yu. Automated Proofs of Object Code for a Widely Used Microprocessor. Ph.D. Thesis, The University of Texas at Austin, 1992. Also available through Systems Research Center, Digital Equipment Corporation, 130 Lytton Avenue, Palo Alto, California, 94301.