The Role of Automated Reasoning in Integrated System Verification Environments

Donald I. Good
Matt Kaufmann
J Strother Moore

Technical Report 73 January, 1992

Computational Logic Inc.
1717 W. 6th St. Suite 290
Austin, Texas 78703
(512) 322-9951

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

This paper focuses on ‘‘system verification,’’ the activity of mechanically proving properties of computer systems. Distinguishing characteristics of systems include their large size, their multiple layers of abstraction, and their frequent concern with multiple programming languages and, more generally, multiple models of computation. System verification *systems* require supreme integration of their component parts. To reason formally about systems one must be able to reason about the relations between its disparate modules, layers of abstraction, and various models of computation. Facilitating system verification is, we believe, the underlying theme of this ‘‘Workshop on the Effective Use of Automated Reasoning Technology in System Development.’’ We address that theme explicitly in this paper. We make an important, often overlooked, but essentially trivial point: to reason formally about the relations between various ideas, those ideas must be formalized. However, the popular approach to imposing formal semantics on computational models, namely the creation of ‘‘formula generators,’’ fails to formalize the very ideas allegedly being studied. In our view, this explains why so much work on system verification is thwarted by ‘‘integration problems.’’ These ‘‘integration problems’’ simply disappear when the ideas being studied are formalized and available for logical manipulation by the automated reasoning system. Our observations are supported by our experience in verifying several of the largest systems studied to date.

Preface

The introductory discussion in the call for papers for this workshop mentions *system* specification and verification repeatedly. For example, it says ‘‘The purpose of this workshop is to explore issues in the application of automated reasoning technology to *systems* with critical requirements’’ [emphasis added]. System verification, as opposed to the more limited area of program verification, is a topic of particular concern to those of us at Computational Logic, Inc. (CLI). We find ourselves compelled to respond to all six of the announced ‘‘Areas of Interest.’’ These may be summarized as follows:

- Applying Different Tools to the Same Problem
- Exchanging Data
- Exchanging Code
- Interface Logics
- User Interfaces and Interaction Styles
- Significant System Verification Applications

We will speak to each of these issues in turn, all within the context of *system* verification. In our companion submission to this workshop, [1], we discuss the possible use of interface logics in program verification.

However, a general theme running throughout this paper is what we will call the

Fundamental Admonition:
To reason formally about an idea you must formalize the idea.

Put another way, if you wish to study an idea formally, you must cast the idea into some formal mathematical logic.

Unfortunately, a popular approach to verification is the ‘‘formula generator’’ approach. A ‘‘formula generator’’ is a computer program that models some aspect of computation such as programming language semantics, security, performance, concurrency, etc. We call these ‘‘computational models.’’[1] For expository reasons, let us fix upon the most common kind of formula generator, a ‘‘verification condition generator’’ or ‘‘vcg’’ program. Such a program is a procedure which implicitly represents some of the semantics of some particular programming language. Let us suppose we have a vcg for the language L. Applied to a particular program in L and its specification, the vcg will formalize the problem ‘‘does this program satisfy its specification?’’ by converting it to the problem ‘‘are these formulas theorems?’’ If one wishes to study particular programs of L, then vcgs are adequate formalizing devices. This explains their historical relevance to verification.

Now suppose one is interested in verifying a system in which L is one of the computational models provided or involved. If all we can do with the vcg for L is apply it to particular programs, then it is not possible to answer questions about L, such as ‘‘is L implemented by this system?’’, ‘‘does L permit these bad transitions?’’, ‘‘how fast does L execute under this model?’’ But these are the kinds of questions asked in a systems context.

In general, formula generators procedurally encode the semantics of some computational model, but they

---

[1]By ‘‘computational model’’ we mean to include universal models of computation such as the Turing machine model and lambda-calculus, but also to include less sweeping models of computation such as security models, performance models, or even models of local relevance only such as the integrity model of some particular data base or spread sheet system.

do not formalize that model (unless the formula generator is, itself, written in a language which has been formalized). Thus, in a systems context, they do not offer a workable response to the fundamental admonition: formalize the ideas you want to reason about.

Perhaps because systems are so complicated, this simple fact is often overlooked. Rather than formalizing the various layers of abstraction and the computational models of local interest to the system, researchers often attack system verification problems by building additional formula generators (e.g., that produce formulas that establish that a particular program doesn't violate any of the integrity rules) and other *ad hoc* ''specification processors.'' These tools are built in an attempt to deal with an unavoidable problem, namely the complexity of the system, by introducing abstractions. This need for abstraction is an inherent property of systems and will arise even if the system being studied is not an operating system or universal computational paradigm but is just a data base or spread sheet or word processor. Abstraction is necessary. But formula generators are an unworkable response because they do not formalize the abstractions and thus do not permit one to reason about relationships between them. The result is the feeling that one has many formula generators that deal with aspects of the system but they are not ''integrated.'' But the problem is not one of integration so much as it is one of formalization.

In our opinion, the fundamental admonition, when taken in a systems context, requires the following responses, in roughly the order listed:

1. choose a mathematical logic,
2. formalize the relevant computational models in that logic,
3. choose or develop some automated reasoning tools to support the chosen logic,
4. explicate each computational model, i.e., develop within the logic a ''reusable theory'' about each model so that high-level proofs can be constructed by the reasoning tools, and then
5. apply the tools and the theories to reason about the system.

Today, the first three steps are often interwoven. System developers who do not wish to work on automated reasoning tools should choose a logic already supported by such tools. As time goes on and more computational models are formalized and explicated within mechanically supported logics, system verification gets easier; it may ultimately become merely a matter of acquiring the appropriate formal models and their reusable theories and then reasoning about the relationships relevant to the particular system being studied. However, because system development so often involves the creation of new computational paradigms, we suspect there will always be some need to formalize and explicate new models.

The five-step program outlined above has been followed repeatedly by numerous researchers and has led to excellent results. The largest systems verified to date have been verified by this approach. Where this methodology is practiced, systems verification is alive and well and fully integrated tool sets proliferate. But the tools are logical entities such as formal models of computations, reusable theories, proof strategies, etc., not formula generator programs.

The structure of this paper is parallel to the Call for Papers for the workshop. We display in italics each ''area of interest,'' taken verbatim from the Call for Papers. We add a final unsolicited item of interest, namely the view of verification as an engineering activity.

1. Applying Different Tools to the Same Problem

1. Applications of different automated reasoning tools to the same problem or specification.

1.1 An Integrated System Verification Environment

At CLI we have a wide variety of automated reasoning tools. One tool allows us to analyze hardware, both for functional correctness and for such ‘‘intensional’’ properties as gate delay and the adequacy of the loads and drives through a netlist. Another tool supports formal reasoning about a machine language. A third supports formal reasoning about an assembly language. A fourth supports formal reasoning about a high-level Pascal-like language. These four tools are completely integrated. We can freely mix the analysis of functional correctness, timing, and other intensional properties at all levels. We can build and verify systems at any one of these levels of abstraction. More interestingly, we can build and verify systems that span many levels of abstraction and involve hundreds of modules at each level. We can smoothly move from the highest levels of a system (e.g., the properties of application programs) to the lowest (e.g., the gate-delay through a given hardware module).

1.2 The FM8502 Short Stack

This integration was achieved as follows. During the period 1985-1988, we and our colleagues at CLI, working concurrently and only loosely coordinated, used the Nqthm (‘‘Boyer-Moore’’) logic [2] to formalize the semantics of a series of computational engines consisting of a gate-level register transfer machine, a machine code (‘‘FM8502’’), an assembly language (‘‘Piton’’), and a high-level language (‘‘Micro-Gypsy’’). Each model was formalized operationally in the Nqthm logic (though the use of operational semantics is irrelevant to our argument); however, the models differ significantly in style because they were formalized by different people. In addition, ‘‘transducers’’ were defined that mapped between the different models: a ‘‘compiler’’ mapped from the high-level language to the assembly code, an assembler and linker mapped down to the binary machine code, and a loader mapped down to the initialization of the gate-level machine. These transducers were defined as functions in the Nqthm logic. Each transducer was proved to preserve, in a certain sense, the semantics of the appropriate computational models. These proofs were constructed with Nqthm and its interactive proof-checker, Pc-Nqthm [3]. These theorems were then combined to obtain a proof of a formula that states, roughly, that if a non-erroneous high-level program is compiled, assembled, linked, loaded, and run on the register-transfer machine, then the answer produced by that machine agrees with that specified by the high-level language. This assemblage is called the ‘‘FM8502 short stack.’’ See [4] for an introduction to our approach and a discussion of each of the components.

1.3 Nqthm: A Verification Shell?

As a side-effect of this work, we developed reusable libraries of lemmas in the Nqthm logic that convert the general-purpose Nqthm theorem prover into a special-purpose machine for reasoning about the particular language concerned. For example, when operating under the ‘‘Piton library,’’ Nqthm becomes an engine for reasoning about Piton. But when the Micro-Gypsy library is loaded, Nqthm is an engine for reasoning about Micro-Gypsy. Thus, in some real and practical sense, we have built and mixed verification environments for a hardware design language, a machine code, an assembly language, and Micro-Gypsy. Indeed, we have used these ‘‘verification environments’’ independently of the short stack to verify properties of programs written in the individual languages. See for example [5].

We have also derived performance bounds on high-level programs in the sense that the proofs give us a constructive characterization of the number of microcode cycles that must be executed to carry out a high-level computation. These ‘‘clock functions’’ are, of course, hierarchically developed in the sense that we measure the performance of Micro-Gypsy programs in terms of assembly language instructions, assembly code in terms of machine instructions, and machine instructions in terms of microcycles. Thus, we have verification environments that let us do performance analysis on programs in several different languages, these languages are connected by verified transducers, and the verification environments are totally integrated.

1.4 The FM9001 Short Stack

Since the FM8502 short stack was reported in 1989, Warren Hunt and Bishop Brock, of CLI, have fabricated (a successor to) the register-transfer layer in silicon. This required formalizing a significant fragment of a commercial hardware description language (the NDL language of LSI Logic, Inc.) and capturing the semantic properties of the conventional ‘‘off-the-shelf’’ hardware modules used in LSI Logic designs. In the new language we also address certain other inadequacies of our earlier approach to hardware; in particular, it is now possible to analyze formally such design aspects as the fanout, gate-delay, and the adequacy of the electrical loads and drives through the netlist. We then re-expressed our chip design in the new language. This slightly changed the machine code implemented by our device and so the semantic definition of the machine code was altered. Because fabrication was intended, we added scan chains and other considerations to allow for post-fabrication testing. The hardware was mechanically verified with Nqthm to implement the new machine code. The new device, called the FM9001, has now been fabricated by LSI Logic, Inc., and is being tested at CLI.

Thus, we added to our tool kit an integrated verification environment for a subset of a commercial hardware description language, including features for analyzing fanout, gate delay, loads and drives. The reusable library for this language has since been used to verify the design of a Byzantine agreement chip [6].

Because we formalized LSI Logic’s NDL we can also use commercial tools to analyze our hardware designs. We have used Nqthm to verify the logical and some performance properties of chip designs and then used LSI Logic tools to do electrical analysis, layout, schematic liberation, and fabrication. This is another example of the application of different tools to the same problem. See [6] for an example.

Upon the fabrication of the FM9001, we ‘‘ported’’ the old stack proof to the new machine using Nqthm. This required modifying the linker so as to generate the binary appropriate for the new machine. Other changes were made in anticipation of actually running the device (e.g., we arranged to lay out the binary in a designated portion of memory so that memory-mapped io could be used). The new assembler/linker was re-verified with Nqthm from a modified version of the old proof script. Because the upper level specification of the assembler did not change, it was logically unnecessary to re-verify the higher levels of the stack. The entire port took less than one man-week.

1.5 Integrating Verification Tools

In summary, at CLI we have a wide variety of integrated verification environments in which we can analyze hardware designs and several different programming languages. We can mix correctness, performance and other analyses. We can address ourselves to properties of particular programs and to properties of particular programming languages. We can smoothly move from the highest levels of a system to the lowest.

This was not achieved by the adoption of an interface logic, disparate formula generators and the cobbling together of a bunch of random tools. We do not believe that the integrated verification systems of the future will be built that way either.

We propose a different and far more flexible and powerful role for logic and automated reasoning: describe your problem—*your whole problem*—in a suitable logic and then reason about it with your theorem prover. In this approach, the first step is to choose a logic suitable for expressing the computational models of interest. This logic will be the ‘‘implementation language’’ of the system verification system in the sense that the computatational models of interest must be expressed in the logic. The chosen logic must be supported by a powerful general-purpose theorem prover. If one is unavailable for the chosen logic, one must be built; this task is hard. System developers wishing to avoid it should choose a logic that is mechanically supported. The automated reasoning system is essentially a system verification system shell; both the development and the application of the verification system proceeds by interacting with the automated reasoning system. In particular, the verification system is built by:

- acquiring or developing formal models of the computational paradigms involved; and then
- acquiring or developing reusable theories, tactics, etc., that explicate those models so that high-level proofs are convenient.

A mathematical analogy may be in order: logic can be used to reason about numbers and to reason about sets. The activity of reasoning is the same in both cases; it is only the data (i.e., the axioms) that lead us to call one activity ‘‘number theory’’ and the other ‘‘set theory.’’ This is the power of logic. To gain access to this power you must cast your problem into logic.

The activity of reasoning is implemented by mechanized reasoning systems. When operating with the axioms of number theory, the theorem prover is doing number theory. When operating with the axioms of set theory, the theorem prover is doing set theory.

Specialized verification environments for two different computational paradigms can be built from a single theorem prover and logic. *Nqthm operating with two distinct data bases is as surely two distinct verifiers as would be obtained by implementing them independently in a common programming language.*

From a systems perspective, however, there is a crucial difference between our approach and the alternative of independently implementing two theorem provers: if our approach is followed it is immediately possible to reason about the union of the two models, i.e., about their interactions and relationships.

Nqthm has been used (or is being used) to formalize and explicate

- a commercial netlist language [6],
- a gate-level description of a microprocessor [7],
- a model of asynchrony suitable to proving the reliability of communication protocols for independently-clocked processors [8],
- Turing machines [9],
- Lambda calculus [10],
- a simple but usable machine code [11],
- a large part of the machine code for the MC68020 [12],
- a stack-based assembly language [5],
- several high level languages including Micro-Gypsy [13], Middle-Gypsy [14], the Nqthm logic itself [2], and a small subset of Ada [15],
- a home-grown separation kernel (implementing multi-processing on a uniprocessor) [16],

- a requirements model for the Mach micro-kernel, and
- the Unity system (a model of the Misra-Chandy language for describing nondeterministic, distributed, concurrent programs) [17].

All of these computational models are integrated in the sense that we can formally reason about their interactions. (See however the footnote about ‘‘Nthm’’ on page 9.)

This is not to say that Nqthm’s logic or its proof engine are ideal; far from it. But their clumsiness and inadequacy are greatly offset by the leverage obtained by casting problems formally and using logic the way it was intended: to reason about formal objects. Nqthm’s success can perhaps be attributed to the fact that its users have ‘‘bitten the bullet’’ and truly formalized their problems.

1.6 Does the Approach Scale?

A standard misconception is that this approach is merely an elegant ‘‘toy’’ that will not ‘‘scale.’’ In this section, by ‘‘scale’’ we mean ‘‘remain manageable as applications approach the complexity of practical systems.’’ Later, in Section 7, we discuss a second sense of the word ‘‘scale’’ where we mean ‘‘remain viable in an industrial setting.’’

The suggestion that the fundamental admonition is an ideal that only works on ‘‘toy’’ examples is simply inconsistent with the fact that *systems verified by this method already approach realistic complexity*. To support this claim we will briefly indicate the complexity of some Nqthm system applications. Because we don’t know how to quantify complexity, we will simply describe the verified systems, emphasizing quantitative aspects and peeling back the first layer of abstraction.

1.6-A The Complexity of the FM9001 Short Stack

The short stack involves four main models of computation: a hardware description language, the FM9001 machine code, the Piton assembly language, and the Micro-Gypsy programming language. Each successive layer was proved to have been correctly implemented on top of the previous one.

The hardware description language permits the description of modules composed of connected primitives and other modules. Approximately 60 primitives are supported. The particular primitives chosen are among those supported by LSI Logic Inc’s language NDL. The model permits reasoning about the values delivered by modules as well as gate-delay, fanout, and the loads and drives requirements. The model provides both two-valued (true and false) and a four-valued (true, false, undefined, and indeterminate) semantics. An extensive reusable library makes it convenient to reason about modules. About one-half page of Lisp code is sufficient to translate our formal hardware description language into the syntax of LSI Logic’s NDL and hence use commercial layout and fabrication tools on the same designs. This direct translation to NDL (it is almost a lexeme-by-lexeme transliteration from parse-tree form to external syntax) is an informal demonstration of the relevance of our formal model.

Using the formal hardware description language, Brock and Hunt designed the FM9001, a simple 32-bit microprocessor. The user-visible state of the FM9001 includes a general-purpose 16-element register file, four arithmetic flags, and the external memory. Register 15 is overloaded to contain the program counter address, but in all other respects operates like the other registers. The implementation allows any of the 16 registers to be used as the program counter.

Each FM9001 instruction fetches operand A, then fetches operand B if necessary, computes a result, and stores the result in the location referenced by operand B. There are two instruction formats (two-address format and immediate datum format). Except for the immediate datum mode there are four addressing modes: register direct, register indirect, register indirect with pre-increment, and register indirect with post-increment. Thus, the FM9001 has five addressing modes for operand A and four addressing modes for operand B. Every addressing mode works with all instructions. There are 15 different arithmetic instructions. Each instruction contains four bits that allow the arithmetic flags, carry, overflow, zero, and negative, to be selectively updated. Also, each instruction contains four bits that specify whether the result computed by an instruction is stored, based on the values of the arithmetic flags at the beginning of an instruction. Conditional branch and jump instructions are just arithmetic instructions that operate on the program counter.

The FM9001 is implemented in a netlist that is composed of 85 modules that collectively reference 1800 primitives of 48 different types. Roughly 28,000 transistors are required in the design. The FM9001 has 95 I/O pins, of which 32 are bi-directional. The NDL translation of the verified netlist contains over 91,000 characters in 2,215 lines. This NDL specification is used by LSI Logic as a wiring guide for an actual implementation.

To facilitate post-fabrication testing, the 248 internal single-bit latches are connected in a scan chain. Using the test enable and input pins, it is possible to load or read every bit in the chain.

The stack-based assembly language, Piton, is implemented on top of FM9001. The Piton language provides provides execute-only programs, recursive subroutine call and return, stack based parameter passing, local variables, global variables and arrays, a user-visible stack for intermediate results, the usual instructions for control and resource monitoring, and instructions for handling seven abstract data types: integers, natural numbers, bit vectors, Booleans, data addresses, program addresses (labels) and subroutine names. There are 65 different Piton instruction opcodes. Piton is implemented by an Nqthm function which transforms Piton programs and data into binary FM9001 machine code.

The Micro-Gypsy language is a high-level language based on Gypsy (which was largely influenced by Pascal). Micro-Gypsy provides four data types: Boolean, integer, character, and arrays of these. The Micro-Gypsy statement types are **`IF`** and **`LOOP`**, together with **`BEGIN-WHEN`**, and **`SIGNAL`** for condition handling, and also procedure definition, call and return. Fourteen predefined utilities are provided. Others can be added at the incremental cost of verifying their Piton implementations. Micro-Gypsy is implemented by an Nqthm function that transforms Micro-Gypsy programs and data into Piton.

The entire short stack effort, i.e., specification and proof of the correctness of Micro-Gypsy on top of the hardware description language, requires about 2,500 Nqthm definitions and the proof of approximately 5,500 lemmas. We estimate the size of the formal proof which Nqthm verified to be roughly 15 million inference steps.[2]

1.6-B The Complexity of the Motorola MC68020

While the stack is relatively deep, the computing engine at the bottom, the FM9001, is simple by the standards of commercial microprocessors.

Yuan Yu, a graduate student at the University of Texas at Austin, has used the Nqthm system to model about 80% of the user available instruction set of the Motorola MC68020 microprocessor. The user

---

[2]By ‘‘inference step’’ we mean the application of such rules of inference as instantiation, modus ponens, substitution of equals for equals, and tautology checking. Because Nqthm does not actually construct a formal proof but (allegedly) verifies that one exists, the size of the verified formal proof is merely estimated here. The estimate is based on figures obtained from an augmented version of Nqthm that counted the number of such steps (in successful branches of the proof search) during a complete run of the FM9001 portion of the stack. The number for FM9001 alone is slightly over 6 million.

visible state consists of 8 data registers, 8 address registers, a 32-bit program counter, an 8-bit condition code register (which includes condition codes for carry, overflow, zero, negative, and extend), and memory. Semantics are formalized for approximately 80 instruction opcodes. Breaking the instructions down into the ten categories of the MC68020 user's manual [18], Yu formalizes all 9 of the data movement instructions, 27 of the 28 integer arithmetic instructions (**CMP2** is excluded), all 9 of the logical operations, all 9 of the shift and rotate instructions, all 4 of the bit manipulation instructions, all 8 of the bit field instructions, none of the binary coded decimal instructions, 11 of the 13 program control instructions (**CALLM** and **RTM** are excluded), 5 of the 21 system control instructions, and none of the multiprocessor instructions. The formal specification is an operational model involving about 700 Nqthm function definitions. It occupies about 80 pages.

Yu has also developed a reusable library of rules about these definitions so that Nqthm can behave as a proof engine for MC68020 machine code programs. The MC68020 library contains approximately 900 rules. The library deals with nonnegative integer arithmetic and modulo arithmetic, alternative interpretations of bit vectors (e.g., the arithmetic meaning of some logical operations), and symbolic memory management (i.e., what is obtained by reading from a symbolic memory address in a symbolic machine state).

To demonstrate the viability of the approach Yu has produced mechanically checked proofs of many interesting machine code programs. These programs are obtained from high-level language programs written in C or Ada and then compiled with commercially available compilers targetting the MC68020. The Gnu C compiler and the Verdix Ada compiler were used. Among the programs so verified are Hoare's *in situ* Quick Sort, binary search, and several C programs from the widely used Berkeley string processing library. See [12] for details.

To the best of our knowledge, this is the first example of the *post hoc* verification of commericially produced software for a commercially produced microprocessor. We do not recommend *post hoc* verification, but it is sometimes possible.

1.7 Conclusion: Logic Comes First

It is no wonder that trouble ensues when we try to impose logic *post facto* via a ''formula generator.'' In such an approach, the computational engine of interest is literally described *informally* in the sense that its semantics is not expressed within the logic and is thus not subject to formal reasoning. Attempts to repair this by building other formula generators, formalizing other models of the engine, introduce the ''integration'' problem by inviting the suggestion that we ought to be able to reason about the relationships between the two models. This of course is impossible as long as the two models themselves remain informal.

Logic comes first. Given a logic, one should ''code'' the semantics of the various computational engines in the chosen logic. ''Formula generation'' is merely an artifact of proof and is mechanically realized by whatever theorem prover supports your chosen logic.

Formula generation programs are just (sometimes elaborate) derived rules of inference about the formal semantics. They allow you to prove certain kinds of theorems about *concretely* given programs. They are useful as such derived rules. We consider it a worthy task to formalize a vcg for one of our computational models and verify its correctness, simply to offer the user additional avenues of proof. But formula generators fit into an integrated whole only when their true role is understood. And in the context of system verification, integration is paramount.

2. Exchanging Data

2. Methods for exchanging useful data among automated theorem provers, software engineering tools, and other analysis tools.

First, we would like to note that to the extent that the various incarnations of Nqthm above are different software engineering tools, we have a perfectly formal method for exchanging data: everything is expressed in the Nqthm logic and is readily available to the different tools.[3]

Second, as noted above, we do not believe that useful integration will be achieved via the pasting together of different logics and theorem provers. Roughly speaking, we think every integrated tool set will be based on a single logic and theorem prover.[4] But this is not to say that the world needs only one logic or theorem prover, much less that we have yet identified even one empirically adequate system. The nature of the logic and its mechanized support depend upon the kinds of computational models needed. This is the subject of today's research.

Such research requires exchanging data between the advocates of the various logics and theorem provers. *Note that this is different from exchanging data between different parts of an implemented system!* We are advocating the need for what might be called *informal data exchange*. In our opinion, such research-level exchange is most easily facilitated by

- having a clear, self-contained, human-readable description of your logic including *all* abbreviation conventions and parsing information,
- self-contained, human-readable papers in which the formalizations of computational models are presented and explicated in the traditional, journal-level informal style, and
- self-contained, machine-readable files in which correspond to the published papers.

Our logic is presented in complete detail in pages 93-141 of [2]. We currently make available approximately ten megabytes of formal definitions and theorems. Each problem set is documented by a published book, journal article, PhD dissertation, technical report, or extensive comments in the file. This discipline succeeds in fostering valuable collaboration.

Our problem sets are used world-wide both to benchmark our system and challenge others. Our bit-vector, natural, and integer libraries, for example, are currently being studied by Gerard Huet's group at INRIA (France) with the aim of converting them to useful libraries for their Coq theorem prover for the calculus of constructions. That group has also taken the Nqthm proof script for the Church-Rosser

---

[3]The newest and as yet unreleased version of Nqthm, called Nthm, is in fact a radical departure from Nqthm in that it provides a great deal of support for combining independently generated reusable libraries. Indeed, in Nthm we have replaced the term ''library'' by the term ''book'' to suggest more directly the idea that a given development effort may rely on the results in many different books. Two problems typically encountered in trying to combine two books or ''explicated models'' are name clashes and disagreement over strategies encoded in rules. These problems are addressed explicitly in Nthm by scoping features and ''theories.''

[4]This sentence is unnecessarily strong but intended provocatively. We actually imagine families of very closely linked logics and various theorem provers that support individual logics within the families. At CLI the Nqthm logic is used both with and without a nonconstructive ''patch'' that adds the power of first-order quantification; we use both the Nqthm theorem prover and its interactive enhancement Pc-Nqthm. But these systems are so closely linked that for all practical purposes (i.e., except when feeling particularly picky about constructivity) we could just always use the extended Nqthm logic and Pc-Nqthm. We also believe that ultimately several different logical systems will be unified by a single formal metalogic. We have already seen this in [19], where propositional calculus, Shoenfield's first-order logic, Church's lambda calculus, and Cohen's hereditarily finite set theory Z2 are all formalized with Nqthm's logic as the metalogic. Properties of these formal systems are then derived, including such interesting ones as that every tautology has a proof, the Church-Rosser property of lambda calculus, and Goedel's incompleteness theorem. We believe that eventually such formal, mechanized metalogical embeddings will permit automated reasoning systems to make the kind of intralogical jumps so common in everyday mathematics.

theorem [10] and massaged it so that Coq can reproduce the proof. Similarly, Hantoa Zhang and Xin Hua, of the University of Iowa, have massaged our Nqthm proof script for the finite version of Ramsey's theorem [20] into a form suitable for successful processing by the RRL [21]. These are significant examples of data exchange at the research level.

In addition, it is easy for us to ‘‘farm out’’ the non-inductive parts of our proofs out to a predicate-calculus proof engine. We can easily generate clause sets capturing such logical problems and have shipped hundreds of such clause sets to Argonne over the years so they could experiment with our problems.

We have also been on the receiving end of such scientific exchanges. For example, Bill Young of CLI has taken the EHDM proof of the interactive convergence clock synchronization algorithm by Rushby and von Henke [22] and converted it to a successfully processed Nqthm proof script [23]. This script contains 200 definitions and theorems (not counting those in our standard rational arithmetic library) and represents another significant example of data exchange.

3. Exchanging Code

3. Methods for promoting the interchangeability of component parts of automated reasoning tools.

There are three major obstacles in the way of interchanging component parts of theorem provers: different implementation languages, different logics, and the so-called ‘‘open black box’’ problem discussed below.

The world's major automated reasoning systems are written in a wide variety of programming languages. Nqthm and RRL are written in Common Lisp, HOL and Nuprl are written in ML, Larch is written in CLU, OTTER is written in C, and Oyster/Clam is written in Prolog. Even if all these systems supported the same logic, it would not be easy or natural to plug in component modules. Unless we wish to standardize on a single implementation language, we believe that interchangeability of components is best achieved via clearly written descriptions of the algorithms and representations.

Such methods actually work. Our system is documented so thoroughly in the book [24] that it has been reprogrammed from scratch in such remote sites as Amherst, Massachusettes and Bejing, China. Our algorithms are at the heart of Alan Bundy's Oyster/Clam project (Edinburgh, Scotland) where they are used to guide the construction of proofs in a Martin Lof-style type theory. Bundy reports that our heuristic algorithms search the space and generate detailed proof plans about ten times faster than the formal proof checker can check them. Our algorithms or methods contributed to the design of such diverse systems as RRL [21], NEVER [25], and INRIA's Coq. *Clear documentation essentially allows the exchange of components even in the face of vast differences in the underlying logical systems and implementation languages.*

We also have much experience with adapting components written by others. It has led us to understand what we called the ‘‘open black box’’ problem. In [26] we describe a four-year effort to encorporate linear arithmetic decision procedures into Nqthm. We studied and implemented procedures by Hodes [27], Bledsoe [28], Shostak [29], and Nelson and Oppen [30]. The primary benefit promised by the inclusion of a decision procedure in a general-purpose setting is to relieve the general-purpose device of the need to explore the search space attributable to the decidable fragment. Our study indicates that in order to realize this benefit it is necessary to ‘‘open’’ the black-box decision procedure so that it can be used in all the ways the general-purpose device would have used the axioms and theorems to be eliminated. Not only may the general-purpose device invoke the decision procedure in many different ways, but it may be necessary for the decision procedure to communicate with the general-purpose device to cause it to pursue goals required by the procedure. Furthermore, once so integrated, we found that the performance of the whole is not much affected by the speed of the decision procedure! The dominant consideration is the speed with which information flows back and forth between the two components. A component that gains internal efficiency by using intricate representations of the data is often less efficient (because of the cost of the transformation at the interface) in an integrated system than a component that sacrifices speed to work directly on the input representations. Thus, despite our conviction that decision procedures are extremely important to general-purpose theorem proving, much research devoted to the run-time efficiency of stand-alone decision procedures is not just wasted effort but counterproductive to the goals of general-purpose theorem proving.

More recently, we have studied the integration of a binary decision diagram (BDD) algorithm [31] into Nqthm. While our work on this topic is still young, the opinions expressed above have been validated by our experiments so far.

In conclusion, while interchangeability of parts is clearly desirable, it will not be enabled by the adoption of a common programming language and/or a common logical basis. Observe that in our linear arithmetic example above, we never discussed the fact that the formalizations of arithmetic employed by the various decision procedures were different from each other and different from that in Nqthm. These differences are often minor (when the decision procedure is of interest at all). Practical interchangeability of parts must await a better understanding of how theorem provers are structured, how their data is structured and stored, how contextual information is maintained, etc. The field is too young for us to set standards on such issues now.

4. Interface Logics

4. Establishing a formal basis for exchanging information among automated reasoning tools and between formal specification processing tools and automated theorem provers.

The ‘‘formal basis’’ mentioned above has otherwise become known as an ‘‘interface logic.’’ In [32], Josh Guttman informally calls an interface logic a *lingua franca* for verification (page 2 of [32]). However in the introduction of the paper Guttman summarizes the most attractive benefits of an interface logic:

> The goal of this report is to advocate the idea of an *interface logic* for verification environments. By this we mean a logic with a syntax that is simple for machines to generate and parse, and that has a standard semantics and a sufficient degree of expressiveness. It is intended to codify logical presuppositions that are common to a considerable number of efforts in specification and verification.
>
> Our hope is that it will become possible largely to separate the work of developing formula-generators, such as verification-condition generators and specification-language processors, from the effort of developing theorem-proving software.

That hope is well-founded. It is clearly possible to separate the development of formula generators from the development of theorem provers. To the extent that formula generators are an attempt to define the semantics of a computational engine, such work has nothing to do with automatic theorem proving. However, it has everything to do with logic.

It is possible to develop a formula generator without selecting a theorem prover. However, it is not possible to develop a formula generator without selecting a logic. The process of generating verification conditions (say) requires mapping from programming constructs (such as the ‘‘**+**’’ symbol in the programming language) to logical symbols (such as the ‘‘integer addition modulo 2^{32}’’ function symbol). These logical symbols must be defined by axioms. Generally speaking, the particular axioms are intimately connected to the programming language. Modulo addition is pretty common and might be expected to be ‘‘predefined’’ in some available logical theory. But other functions, such as ‘‘compute the effective address,'' are unlikely to be predefined. The creation of the axioms is an integral part of the development of the formula generator. Thus, one cannot undertake the creation of a formula generator without having *some* logic in mind.

Having chosen a logic, the formula generator developers create a formula generator and a large pot of axioms in their chosen logic. They then enter the theorem proving market and shop for a system that supports their logic. Logic being what it is, they must find an *exact* match or else the whole point is lost. But unless they considered the available theorem provers before they started, they probably won't find a suitable one because of their tendency to ''roll their own'' logics.

Mike Gordon [email message to **`deftpi`** mailing list on 14 October, 1991] proposed what we believe is the best way to ease the pain of the formula generator people: publish a catalog of all the logics that are supported by automated reasoning systems. Gordon's proposal was:

> One way to address this need is to provide a catalog of available theorem proving tools. This should include, for example, how to get the tool, (availability, costs etc), details of the logic supported (formal syntax and semantics), infrastructure required (e.g. X-windows and Common Lisp), ease of use, cost and availability of training, example case studies, characterization of which kinds of problems the tool is good for, lists of existing users and other general remarks of interest to potential users. Those of us in the field may know this information anyway, but such a catalog might help others.
>
> I think the only way to get such a catalog would be to pay someone to compile it.
>
> My current feeling is that compiling such a catalog should be prior to any attempt to define an interface logic -- and after it is compiled maybe we won't need the interface logic after all!

We would add that the catalog should begin with a dire warning that once a formula generator project has selected a logic they must refrain from ''modifying'' it unless they want to write their own theorem prover.

An alternative approach is the adoption of an interface logic. We discuss some perils of that approach in more detail in [1].

We do not believe there is much merit in adopting an ''interface logic.'' Suppose there is an interface logic and all theorem proving groups undertake to translate from a subset of it to their particular logic. That is, given a problem in the interface logic IL suppose one can mechanically determine whether it has a rendering in the logic PL of some prover and what that rendering is. (Here we are assuming that if the rendering is a theorem in PL then the original formula is a theorem of IL, and we simply ignore the necessary consideration of the axioms and their renderings.) We suspect that most PLs will accommodate only a fragment of the IL, and that fragment will not be as well managed by the host theorem prover as is necessary to handle large problems. Unless some theorem prover is designed to support the entire logic IL, the formula generator people are likely to end up in the disastrous position of not having the means to prove their formulas. They will nevertheless have to discipline themselves to understand and live within the interface logic. And finally, even when a suitable theorem prover is available, they will have to suffer the inefficiency of seeing two levels of transducers between their problem (some program or specification) and the machinations of the proof engine: their own formula generator followed by the translation from IL to PL. In our experience, such transducers significantly complicate and obscure the problem.

Given that the formula generator people must educate themselves on a particular logic and discipline themselves to use it, why not simply provide the catalog so they have a choice?

Turning from Guttman's proposal to the ''**`deftpi`**'' online discussion and the purposes of this workshop, it seems to us that the whole interface logic idea has become a sort of coat tree upon which a host of less considered objectives have been hung (considered in [1]). In particular, the whole connection between interface logics and system verification is spurious. Interface logics were an attempt to solve the problem faced by formula generators. System verification is an activity that requires supreme integration of verification components across a spectrum of computational paradigms and layers of abstraction. The juxtaposition of those two ideas (as in the inclusion of this Item 4 in the Call for Papers) suggests that interface logics may let us accomplish that integration. Wrong. If two formula generators are made to output formulas in the same logical language then it *is* possible to process their output with a single theorem prover. But that is not ‘‘integration.’’ In particular, it does not allow us to relate the models implemented by the formula generators. It does not permit system verification.

As we have already argued, the formula generation approach to semantics is antithetical to system verification because it splits the semantics into formal and informal parts. No amount of ‘‘harmonization’’ of the formalized part will put the two back together to create a formal whole that can be integrated with other levels.

5. User Interfaces and Interaction Styles

5. The effectiveness of alternative user interfaces and interaction styles for automated reasoning tools.

One factor in Nqthm’s success, as already mentioned, is the tradition among Nqthm users to follow the fundamental admonition and formalize their problems. But a number of *engineering decisions* significantly increase the user’s ability to get the job done with Nqthm. We discuss these here.

5.1 The Balance of Expressivity and Proof Power

Logical power seems to come at the cost of effective proof techniques. When we are formalizing models, we want great expressive power; the urge is to design a rich, elaborate logic. When we are automating (or even just mechanizing) effective proof techniques, the urge is toward simple, even decidable, logics. The designers of automated reasoning systems must understand this tension in order to build effective reasoning tools. Too much expressivity frustrates the serious user because it is so hard to prove even simple utterances. Too little expressivity frustrates the serious user because little of interest can be formalized.

Throughout Nqthm’s 20 year evolution it has constantly been applied, both by its developers and others. We believe this ‘‘applications driven’’ methodology is one of the keys to Nqthm’s success and we recommend it to the developers of other mechanized logics. For example, Goldschlag’s work on Unity [17] was an important driving force in recent major extensions to Nqthm: the provision of two new derived rules of inference permitting ‘‘definition by constraint’’ and ‘‘functional instantiation’’ (an apparently higher order act) [33] and the nonconstructive allowance for full first-order quantification in function definitions [34].

In the case of Nqthm, the tension has created a rather weak logic which is balanced somewhat by the power of its proof techniques.

The weakness of the logic works in Nqthm’s favor for people who stay the course. Work devoted to formalizing a model pays off when Nqthm’s heuristics manipulate the model ‘‘naturally.’’ Of course, it is not ‘‘natural,’’ it is just that the system extracts a great deal of strategic information from the user’s constructive, recursive definitions. This has two good effects. It ‘‘front loads’’ the cost of using Nqthm into the formalization phase, giving users better control over the management of projects that are expected to produce models *and* proofs. Secondly, it allows the user to provide strategic guidance while engaged in the creative, high-level activity of modeling.

5.2 User Guidance: Strategy or Tactics?

Consider the spectrum from proof checker to fully automatic theorem prover. While many people think of Nqthm as a fully automatic tool it is in fact far from it. Nqthm has many interactive features. But while Nqthm can be guided by the user, the guidance is not generally at the tactical level of individual proof steps but rather at the strategic level. The system's behavior is largely determined by the *theorems* the user has caused it to prove. Thus, the experienced user's attention is largely focused on mathematics: the formal explication of the concepts in the model and their relationships. The system is responsible for applying these theorems and slogging through the enormous detail involved.

For example, consider our port of the short stack proof from the FM8502 to the fabricated FM9001. Suppose the proof had been done with a proof checker and stored as a series of explicit proof steps. Converting that script to ‘‘replay’’ with the FM9001 would have been difficult because the new functionality at the bottom introduced new cases, new argument positions, and new hypotheses into over 2,000 theorems or their proofs. Detailed proof steps would have been rendered obsolete. But the original proof was encoded as a series of rules that explicated a strategy for Nqthm to follow. The strategy worked for the new machine and so the port was nearly painless.

It should be noted that the explicated strategy worked partly because of the experience of the user who designed it. Less experienced users might have described a less powerful strategy and made the port more painful. ‘‘Tactics’’ as provided, for example, in HOL [35] offer many of the same advantages and disadvantages.

The Nqthm proof checker, Pc-Nqthm, provides users direct control over the construction of a proof of an Nqthm theorem. Among the commands is the invocation of Nqthm itself, so that within the proof checker one can direct some proofs at a low level or with tactics, and let the system handle others with the rules it has been ‘‘taught.’’ A common use of the proof checker is to explore the search space of a problem and find useful strategies for proving theorems. These strategies are then codified as rules for Nqthm's automatic use. Sometimes, as in the Piton proof, scripts are produced that do not appeal to the proof checker at all, even though its use was helpful in the formalization and discovery processes.

5.3 Fully Supported Syntax

The syntax of Nqthm's logic is fully supported by many text editors, window systems, and all Common Lisp programming environments. Furthermore, the formulas seen when you look inside the system (as with various lemma trace and break packages, to see what the system is doing) is suitable for input to the system; indeed, except for some abbreviations commonly used when formulas are typed, the internal and external representation is identical. The lack of barriers between the user and system contributes much to the ease of use.

While Nqthm formulas often look large, they are usually created with a few keystrokes from text that is on the screen. A typical action by the user is to realize (by observing a failing proof) that Nqthm doesn't ‘‘know’’ a certain fact. That fact is then expressed with a series of keystrokes that grab things from the screen and deposit them into the emerging formula: ‘‘This (*click*) **AND** this (*click*) **IMPLY** that this (*click*) is **EQUAL** to this (*click*).’’ The size of such a formula is independent of the time it takes to create it, thanks largely to the support *other systems* provide for ‘‘our’’ syntax.

5.4 Powerful Command Environment

Nqthm's command language is Common Lisp. Thus, many users have developed their own personal commands. The verification environment for our formal hardware description language includes a Lisp command that means ''generate the standard recognizer definition and decomposition lemmas for this module.'' The Piton environment includes the command ''disable all rules except function definitions and the primitives.''

5.5 Effective Interaction Style

We have developed a style of interacting with Nqthm that has been explained at length in [2]. The first rule is that you should have a good proof sketch in mind before you present a formula to Nqthm. This forces you to develop the basic lemma decomposition and proof structure. The second rule is that, tactically speaking, you should ''let the theorem prover have its head but keep it on a short leash.'' That is, you should not try overly hard to direct the system. If you begin to fight the system, by constantly giving it tactical advice, you probably have not adequately explicated a strategy for handling the concepts. Once you begin to fight it, you must fight it forever; *its ability to tolerate detail and keep slogging works against you* because it keeps throwing out problems for you to solve. It is thus better to read its proof attempts as it goes, stop it as soon as it has gotten off the ''right'' path, and explain a strategy by proving rules. This harnesses its tolerance for detail to work in your favor. In the end, input proof scripts are collaborative efforts between the user and the system.

6. Significant System Verification Applications

6. Significant applications of automated reasoning technology to system development.

A characteristic difference between ''system verification'' on the one hand and ''hardware verification'' or ''program verification'' on the other is one of scale: systems, in our usage, are composed of many component hardware and/or software modules. Another characteristic of systems is that their analysis involves multiple layers of abstraction. These layers often include general purpose computing machines or languages. In this section we simply name some significant applications of Nqthm in system development.

Our first example is an unusual one: We assert that the formal metamathematical work done by Shankar in [19] is an excellent example of system verification. The work involves mind-boggling levels of abstraction and several equivalent computational models. In the work, Shankar uses Nqthm's logic as a metalanguage to formalize propositional calculus, Shoenfield's first-order logic, Church's lambda calculus, and Cohen's hereditarily finite set theory Z2. He then proceeds to construct mechanically checked proofs of such important properties of these systems as the metatheorem that every tautology has a proof, the Church-Rosser property of lambda calculus, and Goedel's incompleteness theorem.

We have already offered the two CLI short stacks as significant applications. But even the *components* of the stacks qualify as interesting *systems*:

- the verification of the gate level design of FM9001 involves 85 modules whose composite behavior implements a computing machine best described via four successive layers of abstraction (user, two-valued logic, four-valued logic, and netlist);
- the Piton assembler/linker involves hundreds of functions implementing a transducer described in four successive layers (user, resource model, relocatable symbolic machine code, and absolute binary).

We offer Yu's MC68020 work [12] as a significant example.

We offer the work William R. Bevier, of CLI, in which a ‘‘separation kernel’’ implemented in binary machine code on a uniprocessor was shown to provide multitasking and interprocess communication via message buffers [4].

Finally, we offer the work of David M. Goldschalg, formerly of CLI, who used Nqthm to formalize Misra and Chandy's Unity system for concurrent programming [17]. He then developed a reusable library of proof rules, reproducing those in Misra and Chandy's book [36], causing Nqthm to ‘‘become’’ a verification engine for Unity. He has demonstrated the utility of this proof environment by proving the correctness of several Unity programs, including solutions to the mutual exclusion problem, the minimum node value in a tree, dining philosophers, and a n-bit delay insensitve FIFO queue.

7. Verification as an Engineering Exercise

It is our opinion that the great logicians, e.g., Goedel and von Neumann, did a ‘‘good enough’’ job in inventing logic and foundations for mathematics (set theory, recursive functions). This received logic and foundations is suitable for stating and checking, mechanically, any part of mathematics at a cost not more that an order of magnitude greater than the time it takes to write proofs down at the detail of an undergraduate math textbook. (This experience is confirmed by numerous other proof checking systems, not just Nqthm.) Computing is just a rather tiny and relatively easy part of mathematics.

We believe that those who have for the last 30 years been ‘‘improving’’ the logic or the foundations of computing are fooling themselves if they think that they are easing the task of mechanically checking the correctness of programs. Yet it seems to us that a disproportionately large segment of the community is focusing on this misguided enterprise. We believe this is due to the desire to publish papers suitable for obtaining academic fame and promotion. In our opinion, the building and use of verification systems is largely an *engineering* enterprise, the key scientific breaktthroughs for which were made many years ago by people like Frege, Skolem, Church, Goedel, Turing, von Neumann, and McCarthy.

As noted above, Nqthm's logic is very weak.[5] We are personally amazed at how far you can go towards systems verification with a few discrete data structures and a little recursion. We cite the success of the Nqthm project as evidence that the need for some new, powerful, universal logic is not the bottleneck. The bottleneck is getting people to do the hard work of formalizing and explicating computational models with formal reasoning tools.

Our belief that this approach can be ‘‘scaled’’ into industrial application is based on the fact that explicated computational models and shared, reusable theories make system verification easier. As time goes on, less work is involved in creating the models because so many components are readily available. [38, 39, 40] Less work is involved in proving new theorems because the strategic work done to prove old ones is often directly applicable. This is especially true in the context of ‘‘minor’’ modifications to previously verified components. Finally, the nature of the enterprise—which requires, encourages and *most importantly* rewards the explicit specification of clean interfaces between system components—permits system verification to be carried on in a concurrent fashion between independent groups which may share a few common models and theories but are otherwise free to adopt their own most productive strategies. The cost of system verification will drop substantially when there are a

[5] It is obtained roughly by throwing out the quantifiers from first order predicate calculus with equality and induction and adding a principle of recursive definition based on Shoenfield's [37].

number of verified systems and computational models upon which to build.

The biggest obstacle to the successful application of verification technology to commercial systems is the lack of money and trained technicians.

The key to the industrialization of formal verification is cost effectiveness. It would be useful, we believe, for sponsors to attempt to quantify the cost of verification today, as a benchmark against which progress can be measured.

But cost effectiveness requires assigning dollar values to expected benefits and these are harder to evaluate. What is the cost to the U.S. economy of unreliable software? Because of the ubiquity of software, this is an exceedingly difficult question to answer. Clearly one should consider the cost to the manufacturer of fixing bugs, recalling products, etc. But one must also consider the cost of down time and recalls to customers and their industries. How much productivity is lost due to faulty applications software? What is the cost of protecting one's business against such losses (e.g., via redundant, 19th century backup accounting, filing and inventory systems as well as excessive investment in electronic spares, maintenance, etc.)? How much money is lost or stolen due to software flaws? How much money is lost simply by delays caused by unreliability or redundancy? What is the value of data stolen via software flaws? What is cost of lost sales opportunities due to chronic poor reliability? What is the cost of a life lost to a software error? What happens to the U.S. if all digital computers stopped running for just one particular hour? We believe that sponsors should undertake a study to answer these questions so that it is possible to answer the larger one, ''Is verification cost-effective?''

Finally, in our opinion, the sponsors of verification research ought to hold a workshop to figure out how much it might cost to, say, generate a mechanically verified Mach, and then decide whether they want to pay that much for one. If so, they ought to organize a suitably funded effort.

8. Acknowledgements

This paper could not have been written (except as an unconvincing theoretical exercise) were it not for the work of our colleagues. We would especially like to acknowledge Bill Bevier, Bob Boyer, Bishop Brock, Art Flatau, David Goldschlag, Warren Hunt, Bill Schelter, Natarajan Shankar, Ann Siebert, Larry Smith, Mike Smith, David Russinoff, Matt Wilding, Bill Young, and Yuan Yu for their faith in this approach and their hard work to demonstrate its viability. The opinions expressed herein are strictly those of the authors.

References

1. M. Kaufmann and J Strother Moore, ‘‘Should We Begin a Standardization Process for Interface Logics?’’, Tech. report CLI Technical Report 72, Computational Logic, Inc., 1717 W. Sixth Street, Suite 290, Austin, TX 78703, January 1992.

2. R. S. Boyer and J S. Moore, *A Computational Logic Handbook*, Academic Press, New York, 1988.

3. Matt Kaufmann, ‘‘A User’s Manual for an Interactive Enhancement to the Boyer-Moore Theorem Prover’’, Technical Report 19, Computational Logic, Inc., May 1988.

4. W.R. Bevier, W.A. Hunt, J S. Moore, and W.D. Young, ‘‘Special Issue on System Verification’’, *Journal of Automated Reasoning*, Vol. 5, No. 4, 1989, pp. 409-530.

5. J S. Moore, ‘‘Piton: A Verified Assembly Level Language’’, Tech. report 22, Computational Logic, Inc., 1717 West Sixth Street, Suite 290 Austin, TX 78703, 1988.

6. J S. Moore, ‘‘Mechanically Verified Hardware Implementing an 8-Bit Parallel IO Byzantine Agreement Processor’’, Tech. report Technical Report 69, Computational Logic, Inc., 1717 W. Sixth Street, Suite 290, Austin, TX 78703, August 1991.

7. Warren A. Hunt, Jr. and Bishop Brock , ‘‘A Formal HDL and its use in the FM9001 Verification’’, *Proceedings of the Royal Society*, 1992, to appear April 1992

8. J S. Moore, ‘‘A Formal Model of Asynchronous Communication and Its Use in Mechanically Verifying a Biphase Mark Protocol’’, Tech. report CLI Technical Report 68, Computational Logic, Inc., 1717 W. Sixth Street, Suite 290, Austin, TX 78703, August 1991.

9. R. S. Boyer and J S. Moore, ‘‘A Mechanical Proof of the Turing Completeness of Pure Lisp’’, in *Automated Theorem Proving: After 25 Years*, W.W. Bledsoe and D.W. Loveland, eds., American Mathematical Society, Providence, R.I., 1984, pp. 133-167.

10. N. Shankar, ‘‘Towards Mechanical Metamathematics’’, *Journal of Automated Reasoning*, Vol. 1, No. 1, 1985.

11. W.A. Hunt, ‘‘FM8501: A Verified Microprocessor’’, PhD Thesis, University of Texas at Austin, December 1985, Also available through Computational Logic, Inc., Suite 290, 1717 West Sixth Street, Austin, TX 78703.

12. R.S. Boyer and Y. Yu, ‘‘Automated Correctness Proofs of Machine Code Programs for a Commercial Microprocessor’’, Tech. report TR-91-33, Computer Sciences Department, University of Texas, Austin, November 1991.

13. W. Young, ‘‘A Verified Code-Generator for a Subset of Gypsy’’, PhD Thesis, University of Texas at Austin, 1988, Also available through Computational Logic, Inc., Suite 290, 1717 West Sixth Street, Austin, TX 78703.

14. Donald I. Good, Ann E. Siebert, William D. Young, ‘‘Middle Gypsy 2.05 Definition’’, Tech. report CLI Technical Report 59, Computational Logic, Inc., 1717 W. Sixth Street, Suite 290, Austin, TX 78703, June 1990.

15. Michael K. Smith, Dan Craigen, and Mark Saaltink, ‘‘The nanoAVA Definition’’, Tech. report CLI Technical Report 21, Computational Logic, Inc., 1717 W. Sixth Street, Suite 290, Austin, TX 78703, May 1988.

16. W. Bevier, ‘‘A Verified Operating System Kernel’’, PhD Thesis, University of Texas at Austin, 1987, Also available through Computational Logic, Inc., Suite 290, 1717 West Sixth Street, Austin, TX 78703.

17. D.M. Goldschlag, ‘‘Mechanizing Unity’’, in *Programming Concepts and Methods,* M. Broy and C. B. Jones, eds., North Holland, Amsterdam, 1990.

18. Motorola, Inc., *MC68020 32-bit Microprocessor User’s Manual*, 1989.

19. N. Shankar, ‘‘Proof Checking Metamathematics’’, PhD Thesis, University of Texas at Austin,

1986, Also available through Computational Logic, Inc., Suite 290, 1717 West Sixth Street, Austin, TX 78703.

20. Kaufmann, Matt J. and David Basin, ‘‘The Boyer-Moore Prover and Nuprl: An Experimental Comparison’’, Proceedings of Workshop for Basic Research Action, Logical Frameworks, Antibes, France. Also published as CLI Technical Report 58

21. D. Kapur and H. Zhang, ‘‘RRL: A Rewrite Rule Laboratory -- A User’s Manual’’, Tech. report, General Electric Research and Development Center, Schenectady, N.Y., June 1987.

22. J. Rushby and F. von Henke, ‘‘Formal Verification of the Interactive Convergence Clock Synchronization Algorithm using EHDM’’, Tech. report, Computer Science Laboratory, SRI International, Menlo Park, CA 94025, January 1989, Draft

23. W.D. Young, ‘‘Verifying the Interactive Convergence Clock Synchronization Algorithm Using the Boyer-Moore Theorem Prover ’’, Internal Note 199, Computational Logic, Inc., 1717 W. Sixth Street, Suite 290, Austin, TX 78703, January 1991.

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

25. B. Pase and S. Kromodimoeljo, ‘‘m-NEVER User’s Manual’’, Tech. report TR-87-5420-13, I.P. Sharpe Associates Ltd., 265 Carling Ave., Ottawa Canada, November 1987.

26. R. S. Boyer and J S. Moore, ‘‘Integrating Decision Procedures into Heuristic Theorem Provers: A Case Study with Linear Arithmetic’’, in *Machine Intelligence 11*, Oxford University Press, 1988, Also available through Computational Logic, Inc., Suite 290, 1717 West Sixth Street, Austin, TX 78703.

27. L. Hodes, ‘‘Solving Problems by Formula Manipulation’’, *Proc. Second Inter. Joint Conf. on Art. Intell.*, The British Computer Society, 1971, pp. 553-559.

28. W. W. Bledsoe, ‘‘A New Method for Proving Certain Presburger Formulas’’, *Advance Papers, Fourth Int. Joint Conf. on Art. Intell.*, Tbilisi, Georgia, U.S.S.R., September 1975, pp. 15-20.

29. R. Shostak, ‘‘On the SUP-INF Method for Provign Presburger Formulas’’, *JACM*, Vol. 24, No. 4, 1977, pp. 529-543.

30. G. Nelson and D. C. Oppen, ‘‘Simplification by Cooperating Decision Procedures’’, *ACM Transactions of Programming Languages*, Vol. 1, No. 2, 1979, pp. 245-257.

31. Randal E. Bryant, ‘‘Graph-Based Algorithms for Boolean Function Manipulation’’, *IEEE Transactions on Computers*, Vol. C-35, No. 8, August 1986, pp. 677--691.

32. Joshua D. Guttman, ‘‘A Proposed Interface Logic for Verification Environments’’, Tech. report M91-19, The MITRE Corporation, March 1991.

33. R.S. Boyer, D. Goldschlag, M. Kaufmann, J S. Moore, ‘‘Functional Instantiation in First Order Logic’’, in *Artificial Intelligence and Mathematical Theory of Computation: Papers in Honor of John McCarthy*, V. Lifschitz, ed., Academic Press, 1991, pp. 7-26.

34. Matt Kaufmann, ‘‘An Extension of the Boyer-Moore Theorem Prover to Support First-Order Quantification’’, Tech. report CLI Technical Report 43, Computational Logic, Inc., 1717 W. Sixth Street, Suite 290, Austin, TX 78703, May 1989, To appear in J. of Automated Reasoning

35. M. Gordon, ‘‘HOL: A Proof Generating System for Higher-Order Logic’’, Tech. report 103, University of Cambridge, Computer Laboratory, 1987.

36. K.M. Chandy and J. Misra, *Parallel Program Design: A Foundation*, Addison Wesley, Massachusetts, 1988.

37. J. R. Shoenfield, *Mathematical Logic*, Addison-Wesley, Reading, Ma., 1967.

38. Bill Bevier, ‘‘A Library for Hardware Verification’’, Internal Note 57, Computational Logic, Inc., 1717 W. Sixth Street, Suite 290, Austin, TX 78703, June 1988.

39. Matt Kaufmann, ‘‘An Integer Library for NQTHM’’, Internal Note 182, Computational Logic,

Inc., 1717 W. Sixth Street, Suite 290, Austin, TX 78703, 1990.

40. Matthew Wilding, ''Events for a Reproof of the Search Theorems Using a New Rationals Library'', Internal Note 223, Computational Logic, Inc., 1717 W. Sixth Street, Suite 290, Austin, TX 78703, March 1991.

Table of Contents