[8] Matt Kaufmann, "A User's Manual for an Interactive Enhancement to the Boyer-Moore Theorem Prover," Technical Report 19, Computational Logic, Inc., 1988.

[9] J Strother Moore, "A Mechanically Verified Language Implemenation," *Journal of Automated Reasoning*, 5(4), December, 1989.

[10] James H. Paul, Gregory C. Simon, "Bugs in the Program, Problems in Federal Government Procurement Regulation," U. S. House of Representatives, September, 1989.

[11] L. Robinson and K. Levitt, "Proof Techniques for Hierarchically Structured Programs," *CACM*, 20(4): April, 1977.

[12] Larry Wos and William McCune, "Challenge Problems Focusing on Equality and Combinatory Logic: Evaluating Automated Theorem-Proving Programs," Springer-Verlag Lecture Notes in Computer Science 310, *Proceedings of the 9th International Conference on Automated Deduction*, May, 1988.

[13] William D. Young, "A Mechanically Verified Code Generator," *Journal of Automated Reasoning*, 5(4), December, 1989.

- A verified system may be functionally entirely correct but intolerably inefficient; efficiency concerns are seldom addressed in formal specification.
- The verification process may be flawed or not carried out carefully enough.
- The tools supporting the verification process may not provide a correct implementation of the underlying logic.
- The verification may have been carried out to a certain level of abstraction but the system fail at some even lower level.

These and other important concerns must be addressed; important research issues are still to be faced.

The methods currently available certainly are not mature enough to make any substantial contribution to enhancing the level of assurance in, say, the Strategic Defense Initiative. However for smaller critical applications, they can make a real contribution of enhanced assurance. This potential contribution has long been recognized in specialized areas, notably secure computing and safety critical applications. The work on the verified stack has shown that by following a systematic mathematical approach supported by a powerful automated proof tool, highly reliable systems can be constructed. Additional applications of such methods will undoubtedly emerge.

References

[1] Robert L. Akers, Bret A. Hartman, Lawrence M. Smith, Millard C. Taylor, William D. Young, *Gypsy Verification Environment User's Manual.* Technical Report 61, Computational Logic, Inc., 1990.

[2] William R. Bevier, "Kit and the Short Stack," *Journal of Automated Reasoning*, 5(4), December, 1989.

[3] William R. Bevier, Warren A. Hunt, Jr., J Strother Moore, William D. Young, "An Approach to Systems Verification," *Journal of Automated Reasoning*, 5(4), December, 1989.

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

[5] R. S. Boyer and J S. Moore, *A Computational Logic Handbook.* Academic Press, Boston, 1988.

[6] Warren A. Hunt, Jr., "Microprocessor Design Verification," *Journal of Automated Reasoning*, 5(4), December, 1989.

[7] Jeffery Joyce, Graham Birtwistle, and Mike Gordon, "Proving a Computer Correct in Higher Order Logic," University of Calgary, Department of Computer Science, August, 1985.

environment of his program is undermining those abstractions. We have built several simple application programs in Micro-Gypsy and used the verified stack to translate them to a semantically equivalent load image for the FM8502 microprocessor.

Using the same basic approach described above we have also implemented and proved correct a simple operating system, called *Kit*[2]. Kit is a small operating system kernel written for a uniprocessor von Neumann machine, and is proved to implement a fixed number of conceptually distributed communicating processes on this shared computer. In addition to implementing processes, Kit provides the following verified services: process scheduling, error handling, message passing, and an interface to asynchronous devices.

The proof of correctness involves showing that a certain block of about 3K 16-bit words when executed by the target machine implements a fixed number of isolated target machines capable of communicating only through shared I/O buffers. While Kit is not big enough to be considered a kernel for a general purpose operating system, it does confront some important operating system phenomena. It is adequate for a small special purpose system such as a communications processor.

The uniprocessor for Kit is very similar to the FM8502 microprocessor but was developed concurrently and more or less independently. A consequence of this is that Kit does not fit precisely into our verified stack. But because of the similarity of the two machines, we are confident that we could produce a verified stack with a multiprocessing operating system sitting between the machine code level and the Piton level.

We hope to be able to eventually incorporate Kit-like operating system capabilities into the verified stack between the machine code level and the Piton level. This will allow us to run several communicating parallel Piton processes. It may also permit Micro-Gypsy to be extended to include Gypsy concurrent processing capabilities and I/O.

The verified stack is far from a state-of-the-art software development environment. The components are special-purpose and limited in functionality. However, it serves as a convincing proof of principle. Formal methods can be used in the construction of highly reliable computing systems of at least moderate complexity. Verified components can be assembled into a system that is more than simply the sum of its parts, one that provides a level of assurance that any piece alone could not provide.

7 Conclusions

Formal methods are not a complete solution to the "software crisis." Even a hierarchically verified system is not guaranteed to be absolutely reliable. A number of things can go wrong.

- The specifications may have been wrong or incomplete.

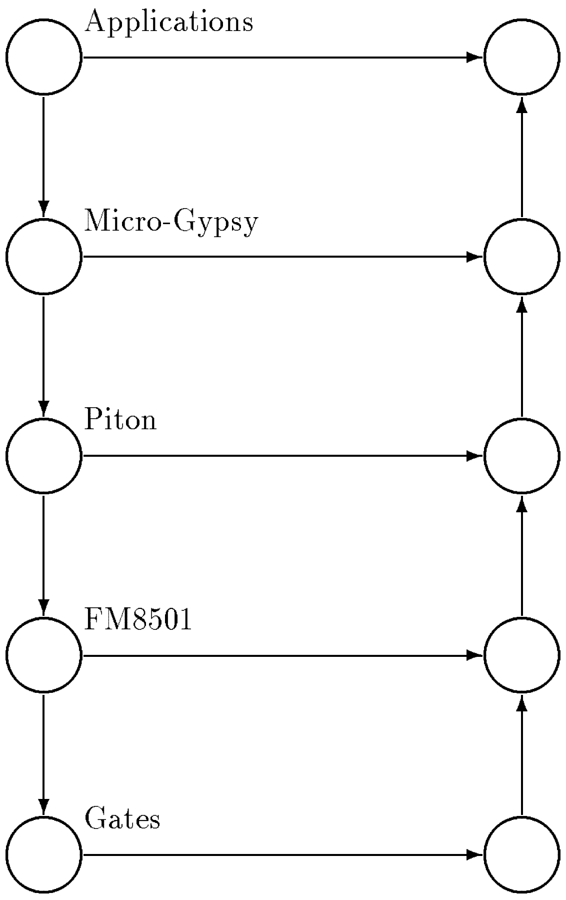

Figure 4: The CLI Stack

ROM containing microcode instructions. The step function uses combinational logic to determine the new contents of the various state-holding devices as a function of the current values and signals arriving on various input lines such as the **`reset`**, the **`data acknowledgement`** and the **`data in`** lines.

Relating each pair of adjacent machines is an *implementation* or *MapDown* function, represented formally as a function in the logic, that maps a higher-level state into a lower-level state. The implementation function is known as a "compiler" for the step from Micro-Gypsy to Piton, but as a "link-assembler" for the step from Piton to FM8502. In addition, for each such pair of machines we define the *MapUp* function that is a partial inverse of the implementation. The *MapUp* function for the Piton implementation, for example, interprets bit strings in the FM8502 memory as elements of the various data types of Piton.

The correctness of each implementation is characterized by a theorem in the Boyer-Moore logic representing an appropriate commutative diagram as explained above. Each of the implementations has been proved correct:

- the correctness of a gate-level register-transfer model of a machine code machine,
- the correctness of a link-assembler from an assembly level language to binary machine code,
- the correctness of a compiler from Micro-Gypsy to assembly language, and

These proofs were all constructed by the Boyer-Moore theorem prover.

We have proved the additional results necessary to let us "stack" the three correctness results. For example, a legal Micro-Gypsy program that executes without high-level errors is compiled into a legal Piton program that executes without Piton-level errors, etc. We thus obtain a theorem that tells us that the error-free execution of a legal Micro-Gypsy program can equivalently be carried out by the microcode machine on the low level state obtained by compiling, link-assembling, loading, and resetting the gate-level machine. Furthermore, we constructively characterize the number of microcycles required. Figure 4 illustrates the current verified stack. We add applications to the "top" of the stack since each verified application written in Micro-Gypsy is really an extension to the stack and possibly provides additional abstractions above what is directly provided in that language.

The result of tying all these efforts together into a unified whole is that an applications programmer can devise a solution to his programming problem and reason rigorously about his efforts within the framework of the high-level language, confident in the knowledge that his program will be translated into a semantically equivalent representation running on hardware at a much, much lower-level of abstraction. He retains the conceptual benefits of the abstraction provided by a high-level language without the worry that the execution abstract machine definitions "in the middle" correspond exactly. That is, the concrete-level machine for the upper diagram must match exactly the abstract-level machine for the lower diagram. Also, $MapDown_1$ must always yield a state that satifies the `GOOD-STATE` predicate for machine `M2`. If these conditions are satisfied, we will have established that the high level machine `M1` is correctly implemented on the machine `M3`, two levels below. It should be apparent that any number of proofs of the type we have described can be "stacked" in this fashion. This makes it possible to establish the correct implementation of a machine in terms of another machine that is any number of levels less abstract.

6 The CLI Stack

The approach described in the previous section has been used to establish formally the correctness of an abstract machine provided by a simple high-level language in terms of the abstract machine representing the implementation of a micro-processor by a collection of hardware gates. The intermediate level machines represent an assembly language, machine language, and hardware functional design. This hierarchical system contains the following components.

- Micro-Gypsy[13]–a high-level programming language. The state consists of a current expression, local variable bindings, and a condition code, together with a static collection of programs in symbolic form. The step function is the recursive statement evaluation for a language providing if-then-else, begin-when blocks, iteration, sequencing, procedure call, and condition signaling.

- Piton[9]–a high-level assembly language. The state consists of an execute-only program space containing named programs in symbolic form, a stack, a read-write global data space organized as disjoint, symbolically named variables and 1-dimensional arrays, and some control information. The step function is the "single stepper" for a stack-based language, providing seven different data types including integers, Booleans, data addresses, and program addresses.

- FM8502[6]–a machine code interpreter. The state consists of eight 32-bit wide registers, four condition code bits, and 2^{32} 32-bit words of memory. The step function is the "single stepper" for a machine code that provides a conventional, orthogonally organized 2-address instruction set. This machine is comparable to a PDP-11 in the complexity of the ALU and instruction set. However, the machine lacks interrupts, supervisor/user modes, and support for virtual memory management.

- Gates–a register-transfer model of a microcoded machine. The state is a collection of 32-bit wide registers and latches, various one bit flags and latches, an array of 2^{32} 32-bit wide words representing memory, and a

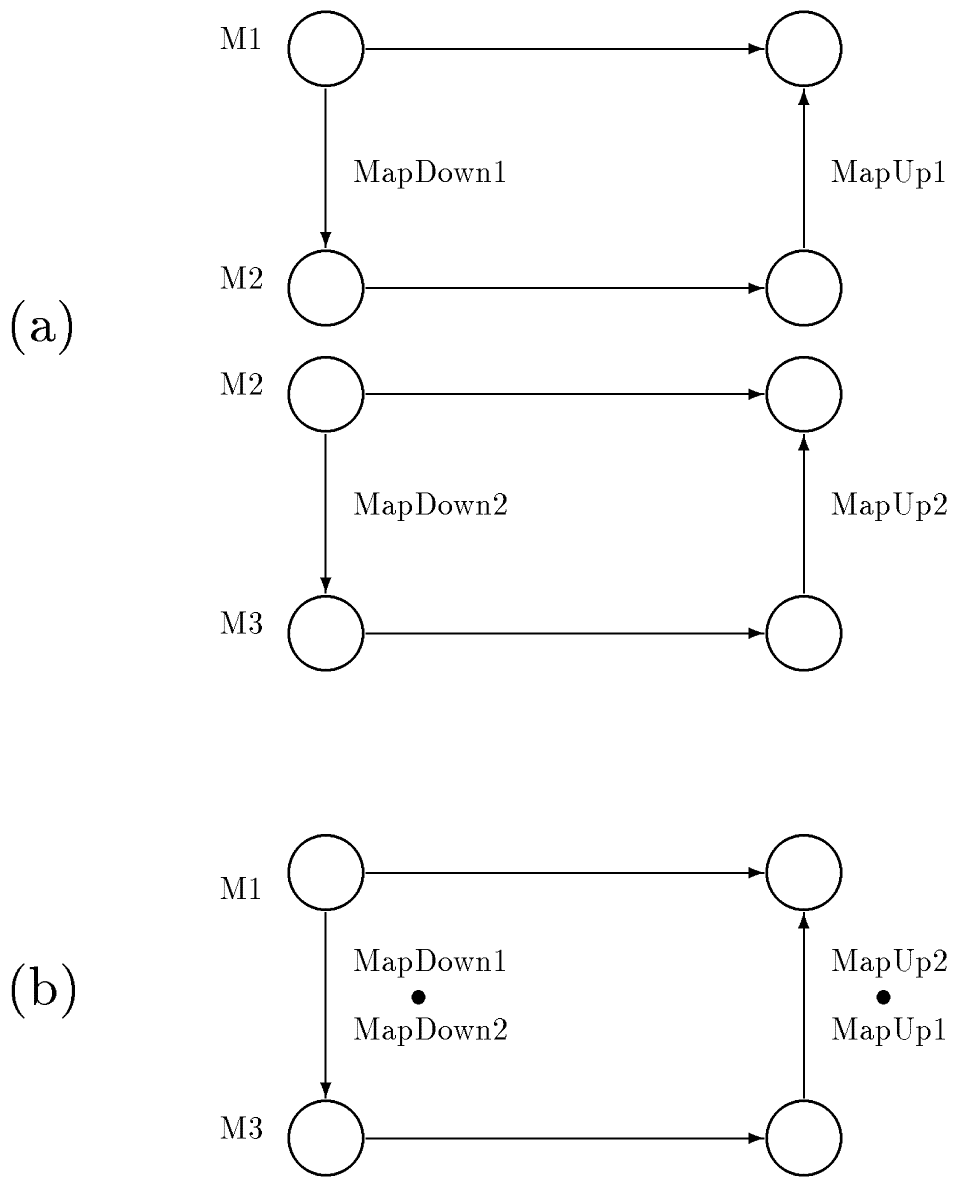

Figure 3: Composing Equivalence Theorems

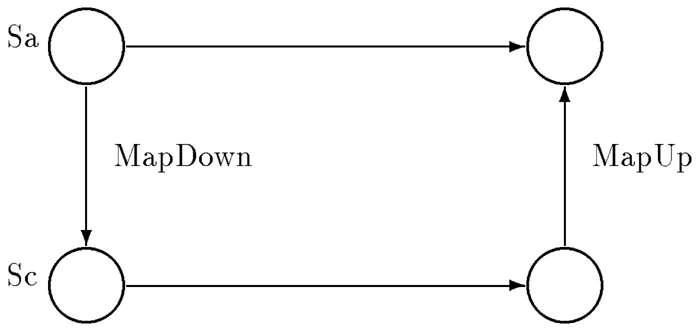

Figure 2: Equivalence of Machines

1. run the abstract interpreter directly on the initial abstract state and view the results;
2. map the abstract state down to a corresponding concrete state, run the concrete interpreter, and map the results back up to the abstract world.

If these two approaches always yield the same result, then the abstract layer is correctly implemented in the concrete layer. With a little thought, it is easy to see that this is a reasonable abstraction of what is usually meant by saying that a compiler is correct or that a piece of hardware is correctly implemented in some technology.

In the Boyer-Moore logic the correctness theorem is stated roughly as follows:

```
Theorem: IMPLEMENTS-RELATION
(IMPLIES (GOOD-STATE ASTATE)
         (EQUAL (MAPUP (INTERPRET-C (MAPDOWN ASTATE)))
                (INTERPRET-A ASTATE)))
```

The hypothesis `(GOOD-STATE ASTATE)` assures that the initial abstract state is a legitimate state from which to begin. Obviously, specific layers are only fully defined when we have completely defined the functions `GOOD-STATE`, `INTERPRET-C`, `INTERPRET-A`, `MAPUP`, and `MAPDOWN`.

Suppose that we can prove two such theorems establishing the correct implementation of machine `M1` on `M2` and of machine `M2` on `M3`. These theorems are represented by the commuting diagrams in figure 3-(a). Under certain conditions, it is possible to compose the machine definitions to obtain the theorem represented by the commuting diagram in figure 3-(b). To be able to compose the commuting diagrams in this fashion it must be the case that the two provides convenient computational abstractions that are not available directly on the hardware platform that supports it. The language is implemented by a compiler that translates high-level language programs to semantically equivalent assembly level language programs. That is, the abstractions of the high-level language are really provided in terms of the abstractions available at the assembly level (and possibly by the operating system of the machine). The abstractions of the assembly language are provided in turn by an assembler on top of the abstractions of the hardware. The abstractions of the hardware model are provided by some collection of hardware gates, wires, and registers.

Thus an applications programmer is typically relying upon a rather sizable collection of system software and hardware, including compilers, assemblers, operating systems, and the underlying machine hardware. The "correctness" of an applications program, then, ultimately depends upon the correctness of the underlying support software and hardware, over which the applications programmer has little control. This means that effort expending in applying even the best formal methods to guaranteeing the "correctness" of a program may be vitiated if the supporting layers are flawed.

One approach to dealing with this problem is to apply our methods to the analysis of the underlying support software and hardware and building applications on top of a "stack" of verified components. This is called *systems verification*[3]–the "layering" of verified components to construct highly reliable hierarchically-structured computing systems. Each "layer" (except the lowest) is proven to be correctly implemented on the next lower layer in the hierarchy.

Using abstract machine models we can characterize the semantics of the individual layers in our stack. To connect adjacent layers, we need to have a way of formally establishing that a lower level machine "implements" a higher level machine. Formally, let $Int_A : S_A \rightarrow S_A$ and $Int_C : S_C \rightarrow S_C$ be interpreter functions that define two machines M_A and M_C. (The subscripts A and C are chosen to suggest *abstract* and *concrete* machines, the higher and lower level machines, respectively.) Let $MapUp : S_C \rightarrow S_A$ be an abstraction function that maps a concrete state to an abstract state, and let $MapDown : S_A \rightarrow S_C$ map an abstract state to a concrete state. *MapDown* is the function that takes us from the higher level abstraction to the lower level. For our high-level language, this would be the compiler that translates a high-level state containing a program and its data structures into a low-level state containing a corresponding assembly language program with its data structures. The *MapUp* function in turn takes a low-level state and "reads out" the results in terms that are meaningful in the high-level context. An example of *MapUp* is the function that takes bit strings in a computer memory and prints them out as numbers or characters depending on the context.

We say that M_C *implements* M_A if the following theorem holds.

This theorem is a formalization of the "commuting diagram" depicted in figure 2. Intuitively, it asserts that there are two possible ways to obtain the results of an (abstract) computation:

- **if b then stmt1 else stmt2**: if **b** is true execute **stmt1**, otherwise execute **stmt2**.

Conceptually, the interpreter function for this language might take the form:

```
(INTERPRET PROGRAM STATE)
=
(IF PROGRAM is <skip>
    STATE
(IF PROGRAM is <stmt1; stmt2>
    (INTERPRET <stmt2> (INTERPRET <stmt> STATE))
(IF PROGRAM is <while b do stmt>
    (IF <b> evaluates to TRUE in STATE
        (INTERPRET <while b do stmt>
                   (INTERPRET <stmt> STATE))
        STATE)
(IF PROGRAM is <if b then stmt1 else stmt2>
    (IF <b> evaluates to TRUE in STATE
        (INTERPRET <stmt1> STATE)
        (INTERPRET <stmt2> STATE))
 <other constructs of the language> ))).
```

Such interpreter semantics or *operational semantics* have been written for programming languages for many years. As early as 1962, this approach was used by John McCarthy to describe the semantics of the Lisp programming language.

Given an accurate operational characterization of a machine or a programming language, we can precisely describe the effects of a program running on that machine or written in that language. Moveover, we can state and prove interesting properties of the language itself. For example, the following rather trivial theorem is easily provable of the language characterized by the `INTERPRET` function.

```
Theorem: EQUIVALENCE-OF-WHILE-AND-IF
(IMPLIES <b> evaluates to FALSE in STATE
         (EQUAL (INTERPRET <while b do stmt> STATE)
                (INTERPRET <skip> STATE))).
```

One use of this theorem is to sanction a compiler to replace any occurrence of a **while** statement by a **skip**, assuming the compiler can establish statically that the test of the **while** is false. Such ability to reason formally about the language, as opposed to reasoning about programs in the language, has been largely neglected.

Given a mathematical definition of a system–a language or a machine–we can also potentially verify that the system is correctly *implemented* on lower-level machines. Almost any system is constructed "on top of" another conceptually lower-level system. A high-level language, for example, is useful because it then terminates returning the state which results. Termination is guaranteed since the (natural number valued) counter `N` is decremented in each recursive call until it reaches zero.

The function `STEP` defined as follows.

```
(STEP STATE) = (EXECUTE (FETCH STATE) STATE)
```

Notice that the functions `MACHINE` and `STEP` characterize a broad class of computing systems.

The details of fetching and executing particular instructions is buried within the functions `FETCH` and `EXECUTE`; Specific machines are defined by formalizing these functions. Typically, the `EXECUTE` function is merely a large `IF` expression that describes the effect of each of the legal types of instructions for the machine and the effect of each on the state of the machine. For example, it might take the form:

```
(EXECUTE INST STATE)
=
(IF (IS-HALT-INSTRUCTION INST)
    STATE
(IF (IS-ADD-INSTRUCTION INST)
    <perform add operation>
(IF (IS-MULT-INSTRUCTION INST)
    <perform multiplication operation>
    <other possible operations> )))
```

Models such as this have been used to formally characterize a number of hardware devices, including several microprocessors[6,7].

In addition to the modeling of hardware devices, similar techniques can be used to define the semantics of programming languages. Here we think of the defining function as an "interpreter" for the language and the state of the machine as the memory or collection of data structures on which the program is operating.

For an assembly level language in which a program is merely a list of simple instructions, the form of the interpreter is very similar to the function `MACHINE` above. For a higher-level language in which various statement types contain other statements as subparts, the structure of the interpreter becomes more complicated. Consider a high-level programming language that contains, among others, the following constructs:

- **skip**: do nothing;
- **stmt1; stmt2**: execute **stmt1** and then **stmt2** in the resulting state;
- **while b do stmt**: as long as **b** remains true, repeatedly execute **stmt**;

chores and record keeping for which the human is ill-suited. This is not to say that mechanical provers could not make a genuine contribution to the more "creative" aspects of mathematics. For example, mechanical theorem provers developed by L. Wos and his colleagues at the Argonne National Laboratories have answered a number of previously open questions in mathematics[12]. For the foreseeable future, however, mechanical provers are likely to remain a useful tool of human mathematicians rather than a replacements for them.

In assuring the correctness of computing system, a powerful prover such as the Boyer-Moore prover can play several useful roles. Many of the theorems that arise in context of program verification are relatively uninteresting from a mathematical perspective yet may be quite tedious to prove, requiring the handling of numerous quite similar cases. One useful role for a mechanical prover is in assuring that all cases are covered and in handling much of the low-level proof effort. That is, the prover can be used as the "proof engine" in a system that processes specifications and programs, even if those specification and programs are not expressed directly within the Boyer-Moore logic. It need only be possible to translate conjectures into the Boyer-Moore logic for presentation to the theorem prover. An earlier version of the Boyer-Moore prover was used in this fashion in the Hierarchical Development Methodology[11] of Stanford Research Institute.

An alternative approach is to use the Boyer-Moore logic directly as a specification language. Within the logic it is possible to model interesting computing systems and to state desired properties of them. With the support of the theorem prover we can then develop proofs of these properties, i.e. verify the system with respect to its specification.

An approach that has been used successfully by the author and others has been to model computing system as abstract "machines" defined as function in the logic. Such an abstract machine is a mathematical model characterizing each possible operation in the system by its effect on a "state"; this model gives an *operational* characterization of the behavior of the machine.

5 Mathematical Modeling of Digital Systems

Consider modeling a typical computer at the assembly language level. At each step in the execution of a program, the machine retrieves an instruction from memory and executes it; this is the so-called "fetch-execute cycle" of the machine. To model the execution of the machine for N steps we define the function:

```
(MACHINE STATE N) = (IF (ZEROP N)
                        STATE
                      (MACHINE (STEP STATE) (SUB1 N))),
```

where STATE contains the abstract state of the machine including the memory, registers, error flags, program counter, etc. This function "runs" for N steps and system's behavior is influenced by the data base of lemmas that have already been formulated by the user and proved by the system. Each conjecture, once proved, is converted into one or more rules and stored in the prover's database to guide the theorem prover's actions in subsequent proof attempts. Often these are rewrite rules, but there are other specialized types of rules as well.

A data base is thus more than a logical theory; it is a set of rules for proving theorems in the given theory. The user leads the theorem prover to difficult proofs by programming its rule base. Given a goal theorem, the user generally discovers a proof himself, identifies the key steps in the proof, and then formulates them as lemmas, paying particular attention to their interpretation as rules. The key role of the user in the system is guiding the theorem prover to proofs by the strategic selection of the sequence of theorems to prove and the proper formulation of those theorems. Successful users of the system must know how to prove theorems in the logic and must understand how the theorem prover interprets them as rules.

Using this approach the Boyer-Moore prover has been used to check the proofs of some quite deep theorems.[1] For example, some theorems from traditional mathematics that have been mechanically checked using the system include proofs of: the existence and uniqueness of prime factorizations; Gauss' law of quadratic reciprocity; the Church-Rosser theorem for lambda calculus; the infinite Ramsey theorem for the exponent 2 case; and Goedel's incompleteness theorem. Somewhat outside the range of traditional mathematics, the theorem prover has been used to check: the recursive unsolvability of the halting problem for Pure Lisp; the proof of invertibility of a widely used public key encryption algorithm; the correctness of metatheoretic simplifiers for the logic; the correctness of a simple real-time control algorithm; the optimality of a transformation for introducing concurrency into sorting networks; and the correctness of an implementation of an algorithm for achieving agreement among concurrently executing processes in the presence of faults. When connected to a specialized front-end for Fortran, the system has also proved the correctness of Fortran implementations of a fast string searching algorithm and a linear time majority vote algorithm. Many other interesting theorems have been proven as well.

It is important to note that all of these proofs were checked by the same general purpose theorem prover, not a number of specialized routines optimized for specific problems. Still, despite its apparent versatility and mathematical acumen, the Boyer-Moore theorem prover does not have any deep mathematical insight. The proof of each of these quite difficult theorems was achieved through the clever automated heuristic application of some simple proof strategies and through the development of powerful theories under the guidance of a human mathematician. The theorem prover's contribution is in assuring the soundness and consistency of the result and in attending to many of the low-level proof

[1]Some of these proofs used the interactive enhancement to the Boyer-Moore prover developed by Matt Kaufmann[8].

```
(prove-lemma associativity-of-append (rewrite)
       (equal (append (append x y) z)
      (append x (append y z))))

     Call the conjecture *1.

     Perhaps we can prove it by induction.  Three inductions are
suggested by terms in the conjecture.  They merge into two likely
candidate inductions.  However, only one is unflawed.  We will
induct according to the following scheme:

      (AND (IMPLIES (AND (LISTP X) (p (CDR X) Y Z))
                    (p X Y Z))
           (IMPLIES (NOT (LISTP X)) (p X Y Z))).

Linear arithmetic and the lemma CDR-LESSP can be used to prove
that the measure (COUNT X) decreases according to the well-founded
relation LESSP in each induction step of the scheme.  The above
induction scheme leads to two new goals:

Case 2. (IMPLIES (AND (LISTP X)
                      (EQUAL (APPEND (APPEND (CDR X) Y) Z)
                             (APPEND (CDR X) (APPEND Y Z))))
                 (EQUAL (APPEND (APPEND X Y) Z)
                        (APPEND X (APPEND Y Z)))),

  which simplifies, applying the lemmas CDR-CONS and CAR-CONS, and
  opening up the definition of APPEND, to:

        T.

Case 1. (IMPLIES (NOT (LISTP X))
                 (EQUAL (APPEND (APPEND X Y) Z)
                        (APPEND X (APPEND Y Z)))),

  which simplifies, unfolding the function APPEND, to:

        T.

     That finishes the proof of *1.  Q.E.D.
```

Figure 1: Proof of the lemma ASSOCIATIVITY-OF-APPEND

interesting properties of them. Since the logic contains rules of inference, it is also possible to construct proofs of these properties as in any of the more familiar formal mathematical systems. However, as mathematical objects, models of large scale computing systems tend to be quite complex. Proofs of their properties are often highly repetitive, involving a large number of cases and sometimes tedious low-level reasoning. As noted earlier, humans are notoriously poor at managing this type of complexity. Automated proof assistance can help in managing complexity and in guaranteeing that cases are not overlooked. Much of the most tedious reasoning can be done mechanically.

A powerful mechanical theorem proving program has been written by Boyer and Moore for reasoning about expressions in their logic (and distributed free of charge via anonymous file transfer). The Boyer-Moore theorem prover is a computer program that takes as input a conjecture formalized as a term in the logic and attempts to prove it by repeatedly transforming and simplifying it. The theorem prover employs eight basic transformations:

- decision procedures for propositional calculus, equality, and linear arithmetic;
- rewriting based on axioms, definitions and previously proved lemmas;
- automatic application of user-supplied simplification procedures that have been proven correct;
- elimination of calls to certain functions in favor of others that are "better" from a proof perspective;
- heuristic use of equality hypotheses;
- generalization by the replacement of terms by variables;
- elimination of apparently irrelevant hypotheses; and
- mathematical induction.

The theorem prover contains many heuristics to control the orchestration of these basic techniques.

The system displays a script of the proof attempt allowing the user to follow the progress of the proof and take steps to abort misdirected proof attempts. From the script it is often apparent to the skilled user how to improve the prover's knowledge base so that a subsequent proof attempt will succeed. The script printed by the prover in discovering the proof of the lemma `ASSOCIATIVITY-OF-APPEND` is shown in Figure 1.

In a shallow sense, the prover is fully automatic; the system accepts no advice or directives from the user once a proof attempt has started. The only way the user can alter the behavior of the system during a proof attempt is to abort the attempt. However, in a deeper sense, the theorem prover is interactive; the

The logic also provides a principle of recursive definition under which new function symbols may be introduced. The following, for example, is a definition within the logic of a list concatenation function `APPEND`.

```
(APPEND X Y) = (IF (LISTP X)
                   (CONS (CAR X) (APPEND (CDR X) Y))
                  Y).
```

This equation submitted as a definition is accepted as a new axiom under certain conditions that guarantee that one and only one function satisfies the equation. One of these conditions is that certain derived formulas be theorems. Intuitively, these formulas insure that the recursion terminates by exhibiting a "measure" of the arguments that decreases, in a well-founded sense, in each recursive call of the function.

The rules of inference of the logic, in addition to those of propositional calculus and equality, include mathematical induction. The formulation of the induction principle is similar to that of the definitional principle. To justify an induction schema it is necessary to prove certain theorems that establish that, under a given measure, the inductive hypotheses are about "smaller" objects than the conclusion. Using induction it is possible to prove such theorems as the associativity of the `APPEND` function defined above; this can be stated as a theorem in the logic.

```
Theorem ASSOCIATIVITY-OF-APPEND
(EQUAL (APPEND (APPEND A B) C)
       (APPEND A (APPEND B C)))
```

Notice that this theorem provides a partial *specification* of the `APPEND` function. It is one of myriad properties of this function and its relation to others that can be defined and proved within the logic. Notice also that this theorem, once proven, becomes available for use in the proofs of subsequent theorems. In particular, we can use it as a *rewrite rule*–i.e., we can interpret it as sanctioning the replacement of any expression that matches the left hand side of the equality by the corresponding instance of the right hand side. Thus, for example, the expression

```
        (APPEND (APPEND V (APPEND W X)) (APPEND Y Z))
```

can be readily shown to be equal to

```
        (APPEND V (APPEND W (APPEND X (APPEND Y Z))))
```

by two applications of `ASSOCIATIVITY-OF-APPEND` interpreted as a rewrite rule. By defining and proving rewrite rules, it is possible to build up a powerful theory for proving interesting facts about a specific domain.

As we will illustrate shortly, the Boyer-Moore logic can be used to build mathematical models of even quite complex computing systems and to state may permit a program to be "proven" to meet its specifications when it actually does not do so.

Often, automated assistance is available for recognizing (parsing) specifications and programs, for processing and storing them, and for reasoning about them. These range from simple parsers for checking the syntactic adequacy of specifications to complete verification systems providing automated support for all phases of the verification process. For example, one of the most mature implementations of formal methods for software development, is the Gypsy Verification Environment[1]. This system provides automated support for interactively editing programs and specifications, parsing them, generating logical formulas sufficient to assure that a program satisfies its specifications, checking the proofs of those formulas, and interactively modifying a program and its specifications while maintaining an accurate view of the changing proof status of the program.

4 The Boyer-Moore Logic and Theorem Prover

A wide diversity of methods and automated systems that support their use are available. To illustrate the utility of formal methods, we concentrate in this paper on one approach used at Computational Logic, Inc. in developing a highly reliable *hierarchically verified* computing system. This is the logic devised by Robert S. Boyer and J Strother Moore and its supporting proof system[4,5].

The Boyer-Moore logic is a simple quantifier-free, first-order logic resembling in syntax and semantics the Lisp programming language. Terms in the logic are written using a prefix syntax–we write `(PLUS I J)` where others might write `PLUS(I,J)` or `I+J`. The logic is formally defined as an extension of propositional calculus with variables, function symbols, and the equality relation. Axioms are added defining the following:

- the Boolean (logical) constants `(TRUE)` and `(FALSE)`, abbreviated `T` and `F`;
- the if-then-else function, `IF`, with the property that `(IF x y z)` is `z` if `x` is `F` and `y` otherwise;
- the Boolean connectives `AND`, `OR`, `NOT`, and `IMPLIES`;
- the equality function `EQUAL`, with the property that `(EQUAL x y)` is `T` or `F` according to whether `x` is `y`;
- and inductively constructed objects including natural numbers, ordered pairs, and literal atoms.

In addition, there is the ability within the logic to add user-defined inductive data structures.

the specification be captured in some standardized notation if possible. Of course, there are likely to be less readily quantifiable or formalizable aspects of the design as well, eg. involving aesthetic concerns and the degree to which the design harmonizes with its intended setting. This means that formal specifications are usually only partial specifications. For any realistic system, there are desirable properties that it is not possible or desirable to formalize.

Over the past few years a variety of specification languages have been devised; some of the most widely used being Z, VDM, Larch, Gypsy, and OBJ. Though some of them have been used for specifying systems other than computing systems, for our purposes we view the role of a specification language as providing a well-defined and unambiguous formalism for expressing desired properties of computing systems. Some of these languages take a *declarative* approach in which the desired properties of the system are stated without reference to the mechanisms by which these ends are attained. Other languages take a more *procedural* approach in which the properties are stated in terms of an "abstract implementation" of the system displaying one way in which the desired results can be obtained. Available formalisms range from the structured version of first-order logic and elementary set theory found in Z to the full-blown programming language including concurrency and data abstraction found in Gypsy.

Verification is the process of establishing rigorously that a design meets or satisfies (some of) its specifications. For a bridge design, "verification" may involve computation of the stress that the design will bear and determination whether the resulting safety factor is adequate for the intended traffic. For a program it means showing through rigorous analysis that every execution of the program will have the desired effect. Some of the desirable properties of a design may not be suitable for verification. The aesthetic properties of a bridge or the efficiency of a computing system may be left unspecified, and hence are "verified" only informally.

Finally, transformation involves the derivation of a more complete design from a less complete one or from a specification. In civil engineering, an earlier design may be adapted to meet a later need, or a chief engineer's design may be elaborated by his or her assistants according to accepted engineering principles and practice. In programming, it may mean the modification of a program via various semantics-preserving transformations to gain additional efficiency or functionality, or the instantiation of a program "schema" to gain an instance suitable for some programming task.

To allow either verification or transformation, there must be an underlying *logic* for reasoning about specifications and programs and proving appropriate relationships between them. That is, a specification language by itself is not very useful unless there are rules of inference by which the specification can be related to an implementation. The most crucial property of such a logic is *soundness*–it should be impossible to derive any false conclusion from true premises using the rules of inference of the logic. Lack of soundness in the logic with the Tacoma Narrows bridge, can be catastrophic.

Still, no modern engineer would seriously suggest abandoning mathematical modeling in civil engineering. The use of mathematical techniques to investigate properties of a design *before* it is realized in steel and concrete yields too many significant benefits. These include the obvious benefits of cost reduction and enhanced assurance that the design is safe. Less obvious are the concomitant benefits including: more efficient use of materials; discovery and elimination of errors early in the design process; the ability to efficiently investigate multiple design options; the possibility of automated assistance in the design process; enhanced communication between the design and construction teams; and better documentation of design decisions. Moreover, a widely used common formal notation for expressing designs aids the development of a convenient and accessible archive of engineering successes that other designers can study and emulate; that is, designs become more readily reusable. Because of these benefits of mathematical modeling, a modern civil engineer would never adopt the approach to bridge design of "let's build it and see if it works." Yet amazingly this is a common approach to the construction of complex digital systems.

3 Applying Mathematical Modeling to Digital System Design

The field of formal methods attempts to gain the benefits of mathematical modeling in the design of digital systems in much the same way that modeling is used in design in more established engineering disciplines. Exactly what falls under the umbrella of "formal methods" is widely debated, but it includes at least:

- *specification*: the use of a standard notation to describe desired behavior or properties of a system;
- *verification*: the application of rigorous reasoning techniques to assure that a program satisfies its specification;
- *transformation*: use of (correct) semantics-preserving transitions to develop an implementation from a specification.

Notice from our previous discussion that each of these activities have analogues in traditional civil engineering practice.

Specification is merely the statement of desired properties of a design, whether that design is of a bridge or of a computing system. Those properties may be stated informally in English or some other natural language, or formally in the language of mathematical logic or another notation. Since a natural language like English is notoriously ambiguous, it is desirable for significant designs that contribution of mathematical models to software engineering, consider the role of models in a more traditional engineering discipline such as civil engineering, as in the design and construction of a bridge.

The civil engineer brings to this task a wealth of knowledge cultivated through a long process of education, apprenticeship, and practice. He or she has a keen grasp of the principles of structural dynamics, stress analysis, properties of materials, and diverse other fields relevant to this discipline. Moreover, our engineer has available an extensive catalogue of sound principles of design, well-tested engineering techniques, and previously successful design examples. On the basis of this solid foundation of knowledge and experience, the engineer produces an initial design (model) that specifies the structure of the bridge, length of spans, the size and composition of supporting members, etc. The development of this design is a highly creative process guided by the overriding goal of a structurally sound, functionally suitable, and aesthetically pleasing end product.

With an initial model or design in hand, by the application of well-understood mathematical techniques the engineer can *calculate* whether a bridge built according to this design will perform its intended function and support the expected load. This process of verifying mathematically the properties of the design is essentially a partial "proof of correctness" of the design, i.e, rigorous evidence that the design meets its specifications. Often these calculations will show the engineer's initial design to be perfectly adequate. This is not too surprising since the design process itself was guided at every step by good engineering principles and accompanied by an ongoing process of mathematically verifying portions of the design before they were incorporated into the evolving whole. A design "proven" in this manner can guide the construction of the bridge with a high degree of confidence that the resulting structure will stand the load.

Certainly, structurally sound bridges were built before sophisticated analytic techniques were available. The *Pons Fabricus* of Rome is still in use almost twenty centuries after its construction. But its longevity is due largely to the fact that, from a modern standpoint, it was massively "over-engineered" for its expected load of ancient carts and foot traffic. The Roman engineers did not have available the mathematical modeling techniques that allow their modern counterparts to calculate stresses in their designs and hence compute safety factors. The result was obviously quite durable, but also much more massive and costly to build than other structures that would have sufficed.

Using mathematical modeling in any engineering discipline is not infallible, of course. A memorable experience in many engineering students' educations is viewing the riveting 1940 film of the collapse of the Tacoma Narrows bridge. This film is a spectacular reminder of the limitations of even good engineering practice. Moreover, it illustrates a fundamental limitation of modeling; by definition a model is an abstraction of reality. A model may abstract away or simply overlook factors that ultimately prove to be important. The results, as ior. Programs are tested on a selection of potential inputs; software modules that behave adequately for these inputs are deemed suitably reliable. Modules that misbehave on test cases are "patched" until adequate behavior is attained. Critics of this approach to enhancing software reliability often quote the famous aphorism of Edsger Dijkstra that testing "can be used to show the presence of bugs but never to show their absence." Of course, some small programs can be exhaustively tested. It is usually infeasible, however, to assure the absence of errors by testing because, except for the most trivial programs, the potential input space is simply too large for exhaustive testing. Hence, according to well-accepted industry figures, software produced by standard "good" software engineering practice typically contains around 1-3 errors per 1000 lines of code. At this rate, a software system of 1,000,000 lines of code could be expected to contain 1,000 to 3,000 errors. Though there is a large research community investigating ways to improve the efficacy of testing, testing alone is clearly limited.

The field that has come to be known loosely as *formal methods* attempts to achieve software reliability by applying mathematical modeling in the construction of digital systems. Using rigorous and mathematically-based techniques that model programs and computing systems as mathematical entities, practitioners of formal methods attempt to *prove* that program models meet their specifications for all potential inputs. This approach augments traditional testing-based software engineering practice with *deductive* approaches to predicting software behavior and offers promise for enhancing the quality of software, at least in selected applications.

Of course, actual computing systems–physical machines and the software that runs on them–are not mathematical abstractions. It is only *models* of these physical systems that are accessible to rigorous mathematical reasoning techniques. Care must be taken in validating that such models capture relevant aspects of our systems and in extrapolating the results of the mathematical analysis to the physical reality. But, carefully applied, these methods can significantly improve the predictability and reliability of computing systems.

In the remainder of this paper we explore the role of formal methods in digital systems engineering and illustrate one significant application of formal methods–the construction of a hierarchical verified systems. This system comprises a compiler for a simple high-level language, an assembler and link-loader for an assembly level language, and a microprocessor with its gate level realization. We also describe a verified multi-tasking operating system.

2 Mathematical Modeling in Engineering

The key concept underlying the application of formal methods is the notion of a mathematical *model* of a computing system. We will shortly discuss models of several moderately complex systems. But first, to understand the potential

1 Introduction

As the effects of electronic computing become increasingly pervasive, the potential for serious negative consequences caused by malfunctioning or incorrect computing systems also increases. Errors in digital systems usually result from the failure of the designers to accurately *predict* the behavior of their system under intended circumstances of use, or to *foresee* novel conditions of use for which the system was not originally intended.

The potential for failure of software, particularly, has increased in recent years as software has grown explosively in size and complexity. Commercial aircraft often have tens or hundreds of thousands of lines of operational code. Accounting and weapons systems routinely have code measured in millions of line. This explosion in software size has reached what some sceptics would consider absurd heights with the proposed Strategic Defense Initiative, certainly the most ambitious software project ever conceived. Its designers have envisioned fielding a system estimated to contain up to 40,000,000 lines of code. Merely coordinating such a vast endeavor and assuring that the myriad pieces fit together smoothly is a monumental task, leading many to question whether such a project could ever succeed.

In addition to the complexity arising from the sheer size of the code, inherently more complex application domains are coming increasingly under automatic control. Modern concurrent, distributed, and real-time applications often involve subtle time-dependent and sequence-dependent interactions among multiple processors or interactions with an unpredictable physical environment. Such applications severely tax the analytic powers of the human charged with assuring the quality and "correctness" of the code. Humans are notoriously poor at managing the kinds of complexity found in these applications. When the stakes are raised with the addition of critical safety and security concerns, effectively eliminating any margin for error, the point is quickly reached where the types of informal analysis traditionally applied to achieve software quality becomes totally inadequate to gain acceptable levels of assurance.

This inability to gain adequate assurance in the quality of software contributes to the much-lamented "software crisis" described as follows in a U.S. congressional staff report.

> As the complexity of systems increases, Government managers find that the software they buy or develop does not achieve the capabilities contracted for, that it is not delivered at the time specified, and that the cost is significantly greater than anticipated. Indeed, many recent examples of cost overruns and degraded capability cited as examples of Government waste, fraud and abuse can be related to problems in the development of computer software.[10]

The standard approach to attaining assurance in the correctness of software is to test it. Testing is an *inductive* approach to predicting software behav-

Abstract. We explore the role of mathematical modeling in digital systems development as a way to gain enhanced assurance in their correctness. We concentrate on systems verification–the "layering" of verified components to construct highly reliable hierarchically-structured computing systems. We illustrate with an existing hierarchically verified system, the CLI stack, comprising a compiler for a simple high-level language, an assembler and link-loader for an assembly level language, a microprocessor with its gate level realization, and a simple multi-tasking operating system.

Mathematical Methods for Digital Systems Development

Donald I. Good and William D. Young

Technical Report 67 August 28, 1991

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

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

EMAIL: good@cli.com, young@cli.com

This paper is to appear in the proceedings of VDM'91 held in The Netherlands, October 21-25, 1991. Much of the work described was the individual and collective efforts of our colleagues at Computational Logic, Inc., particularly Bill Bevier, Bishop Brock, J Moore, and Warren Hunt. This work was supported in part at Computational Logic, Inc., by the Defense Advanced Research Projects Agency, ARPA order number 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.