An Approach to Systems Verification

William R. Bevier,
Warren A. Hunt, Jr.,
J Strother Moore,
William D. Young

Technical Report 41 April, 1989

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 Orders 6082 and 9151. 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.

1. Introduction

A program which is proved correct in a high-level language may not behave as expected when executed on a particular computer. This problem is due in part to a "semantic gap" between the high-level language and the computer's instruction set. The data types and operations of the high-level language are different from those which occur in the machine code, and correct execution of the program depends on correct translation between these two domains. The semantic gap does not stop at the machine instruction level. The correct execution of the program also relies on the correct implementation of the processor in hardware, the semantics of which is defined in terms of boolean operations and memory elements. The problem is also due to the fact that a program proof ignores many of the operations which take place in the computer system on which the program executes. The underlying run-time support handles many functions, such as response to interrupts and multi-tasking, which are supposed to be invisible to a program.

The answer to the problem is to verify the *execution environment* of a program: the software and hardware which is reponsible for the correct execution of a high-level language program. We use the term *systems verification* to refer to the specification and proof of correctness of these components of a computing system. The purpose of verifying a compiler is to ensure that the machine code representation of the program makes transitions which are semantically faithful to those of the high-level language. The purpose of verifying a hardware processor is to carry the proof of correct translation down to the level of gates and flip-flops. The purpose of verifying an operating system is to ensure that *only* the transitions permitted by the high-level language are visible to the program.

In this paper we outline our approach to systems verification, and summarize the application of our approach to several systems components. These components consist of a code generator for a simple high-level language [16], an assembler and linking loader [14], a simple operating system kernel [2, 1], and a microprocessor design [11]. Each of these is formally specified by an abstract finite state machine, and each is proved correct by showing that a lower level machine implements the abstract machine. In the case of the compiler and the assembler, a translation function is verified to correctly map an arbitrary "legal" abstract state to an implementation state. The low-level instructions generated by the translators are proved to correctly implement high-level operations. In the case of the operating system and microprocessor, the mapping function plays a less central role. A particular low-level machine is proved to correctly implement a particular abstract machine.

All of the specifications were written in Boyer-Moore logic. Because of the common formal framework, and because the implementation machine of each layer (except the bottom) was chosen to be the specification machine of the next lower layer, it was possible to combine several of the components into a verified "stack" in which each component is implemented on top of another. The resulting level of integration is far beyond anything previously attained with verified components. Our collection of verified components permits a high-level language program to be compiled, the resulting assembly language program assembled, loaded and run on a microprocessor which is itself verified to the gate level. Each translation step is formally certified to preserve the semantics of the input program. The result is a significant bridging of the so-called "semantic gap."

The system components which have been verified are remarkable only in the degree to which formal methods have been applied to their design, verification, and integration. The simple high-level language which sits on top of our stack, for example, is very limited in elegance and expressive power. We view the current compiler and other system components as prototypes which have been quite useful for illuminating the difficulties involved in specifying and proving system software and hardware and in stacking components. We refer to our current stack as the "short stack" because it is much less ambitious than others we have in mind. We believe that the approach to systems verification described in this paper can be applied to more realistic systems components.

This paper is organized as follows. In section 2 we briefly describe the Boyer-Moore logic, the formal framework in which each of the specifications and proofs was constructed. We also describe the Boyer-Moore theorem prover and an interactive interface for the prover used for some of the proofs. Section 3 describes our approach to specifying system components as abstract machines characterized by interpreter functions. We show the form of the correctness results proved about our abstract machines and how these results can be "stacked" to form a verified hierarchy of successively more detailed implementations. In section 4 we briefly describe the components of our verified stack and how they can be used to construct highly reliable computing systems.

2. The Boyer-Moore Logic and Theorem Prover

The formalism in which we define our abstract machines and prove theorems about them is the computational logic of Boyer and Moore, a simple quantifier-free, first-order logic resembling Pure Lisp. The Boyer-Moore theorem prover is a computer program that can be used interactively to prove theorems in the Boyer-Moore logic. Both the formal logic and the Boyer-Moore theorem prover are fully described in [8]. In the current section we give a short overview of the logic and the theorem prover.

Another theorem proving tool which was used for part of the work reported in this series of papers is an interactive interface to the Boyer-Moore prover written by Matt Kaufmann and fully described in [12]. We briefly describe this as well.

2.1 The Boyer-Moore Logic

Terms in the Boyer-Moore logic are written using a prefix syntax similar to Pure Lisp. For example, 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 objects `(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 "connector functions" `AND`, `OR`, `NOT`, and `IMPLIES`; for example, `(NOT P)` is `T` if `P` is `F` and `F` otherwise;
- the equality function `EQUAL`, with the property that `(EQUAL X Y)` is `T` or `F` according to whether `X` is `Y`;
- inductively constructed objects, including:
  - Natural Numbers. Natural numbers are built from the constant `(ZERO)` by successive applications of the constructor function `ADD1`. The function `NUMBERP` recognizes natural numbers, e.g., is `T` or `F` according to whether its argument is a natural number or not. The function `SUB1` returns the predecessor of a non-`0` natural number.
  - Ordered Pairs. Given two arbitrary objects, the function `CONS` returns an ordered pair containing them. The function `LISTP` recognizes such pairs. The functions `CAR` and `CDR` return the two components of such a pair.
  - Literal Atoms. Given an arbitrary object, the function `PACK` constructs an atomic symbol with the given object as its "print name." `LITATOM` recognizes such objects and `UNPACK` returns the print name.
- Each of the classes above is called a "shell." `T` and `F` are each considered the elements of two singleton shells. Axioms insure that all shell classes are disjoint;
- the definitions of several useful functions, including:
  - `LESSP` which, when applied to two natural numbers, returns `T` or `F` according to whether the first is smaller than the second;
  - `LEX2`, which, when applied to two pairs of naturals, returns `T` or `F` according as whether the first is lexicographically smaller than the second; and
  - `COUNT` which, when applied to an inductively constructed object, returns its "size;" for example, the `COUNT` of an ordered pair is one greater than the sum of the `COUNT`s of the components.

The logic provides a principle under which the user can extend it by the addition of new shells. By instantiating a set of axiom schemas the user can obtain a set of axioms describing a new class of inductively constructed `N`-tuples with type-restrictions on each component. For each shell there is a recognizer (e.g., `LISTP` for the ordered pair shell), a constructor (e.g., `CONS`), an optional empty object (e.g., there is none for the ordered pairs but (ZERO) is the empty natural number), and N accessors (e.g., CAR and CDR).

The logic provides a principle of recursive definition under which new function symbols may be introduced. Consider the definition of the following list concatenation function.

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

The equations submitted as definitions are accepted as new axioms under certain conditions that guarantee that one and only one function satisfies the equation. One of the 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 recursion. Here is a suitable derived formula for APPEND.

```
(IMPLIES (LISTP X)
         (LESSP (COUNT (CDR X)) (COUNT X)))
```

However, in general the user of the logic is permitted to choose an arbitrary measure function (COUNT was chosen above) and one of several relations (LESSP above).

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 APPEND.

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

2.2 The Mechanization of the Logic

The Boyer-Moore theorem prover is a computer program that takes as input a term in the logic and repeatedly transforms it in an effort to reduce it to non-F. The theorem prover employs eight basic transformations:

- decision procedures for propositional calculus, equality, and linear arithmetic;
- term rewriting based on axioms, definitions and previously proved lemmas;
- application of verified user-supplied simplifiers called "metafunctions;"
- renaming of variables to eliminate "destructive" functions in favor of "constructive" ones;

- heuristic use of equality hypotheses;
- generalization by the replacement of terms by type-restricted variables;
- elimination of apparently irrelevant hypotheses; and
- mathematical induction.

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

In a shallow sense, the theorem 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 proof attempt. However, in a deeper sense, the theorem prover is interactive: the system's behavior is influenced by the data base of lemmas which have already been formulated by the user and proved by the system. Each conjecture, once proved, is converted into one or more "rules" which guide the theorem prover's actions in subsequent proof attempts.

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 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.

The system has been used to formalize and prove a wide variety of problems including the gate-level hardware designs, operating system functionality, and language implementations described in this series of papers. Other interesting theorems proved by the system are: Gauss' law of quadratic reciprocity [8], the so-called "jewel of elementary number theory;" the Church-Rosser theorem for lambda calculus [15]; and Goedel's incompleteness theorem, the deepest and most stunning theorem in 20^{th} century mathematics [15]. The theorem prover has checked the proof of invertibility of the RSA public key encryption algorithm [7], the correctness of metatheoretic simplifiers for the logic [3], and the optimality of a transformation for introducing concurrency into sorting networks [13]. When connected to a verification condition generator for Fortran [5], the system has proved the correctness of Fortran implementations of the Boyer-Moore fast string searching algorithm [4, 5] and Moore's linear time majority vote algorithm [6].

2.3 An Interactive Enhancement to the Prover

Also available for assistance in proving theorems stated in the Boyer-Moore logic is an interactive interface to the prover written by Matt Kaufmann [12]. The purpose of the interface is to give the user more precise control of the theorem prover in the search for a proof.

The user can give commands at a low level (such as deleting a hypothesis) or at a high level (such as calling the Boyer-Moore Theorem Prover). The system is goal-directed: a proof is completed when the main goal and all subgoals have been proved. A notion of *macro commands* lets the user create compound commands, in the spirit of the *tactics* and *tacticals* of LCF [10]. Upon completion of an interactive proof, the lemma proved may be stored in the database of the Boyer-Moore theorem prover. The interactively constructed proof is also stored so that it is possible to "replay" the proof in the event that the evolving database is modified in the future. Interactively proved lemmas are otherwise indistinguishable from automatically proved ones and the rules generated from both types of lemmas are freely available to the various rule-driven systems in the theorem prover and in the proof checker.

Some users appeal to the proof checker only as a last resort, when their attempts to guide the theorem prover via the construction of rules has ended in frustration. Others use the proof checker exclusively, subjugating all calls to the Boyer-Moore theorem prover to proof checker commands. Still other users find the proof checker best for exploring a search space in a given problem domain—tentatively constructing many proofs with it—and then codify their insights as rules in a database that contain no explicit use of the proof checker.

The proof checker was used extensively in the proof of our code generator [16]. It has also been used to check theorems stating the correctness of a transitive closure program, a Towers of Hanoi program, a ground resolution prover, irrationality of the square root of 2, an algorithm of Gries for finding the largest "true square" submatrix of a boolean matrix, and a finite exponent two version of Ramsey’s Theorem. A modified version supporting first-order quantification has been used to prove the infinite exponent two version of Ramsey’s Theorem.

3. Interpreters and Interpreter Equivalence Theorems

We verify components in our execution environment by establishing a relationship between the behavior of two finite state machines. An abstract finite state machine serves as an operational specification for each component, while a second abstract machine serves as a definition of the implementation of a component. A correspondence theorem states that an implementation machine simulates, or *implements*, its specification. When the implementation machine of one component is identical to the specification machine of another component, the two *implements* theorems can be composed to prove a single theorem spanning several machines.

This section details our definition of finite state machines and describes the form of the correspondence theorem between two machines. Also shown is how correspondence theorems for several machines can be combined into a proof of a "stack" of verified components.

3.1 Finite State Machines

Defining a finite state machine requires a description of the set of machine states and a definition of each transition on a machine state. The set of machine states is defined by a predicate which recognizes a member of the state set, a so-called *good state*. The transitions on a finite state machine are defined by an *interpreter function*.

We usually define the state of a machine as a record structure The state of a von Neumann computer, for example, might contain the fields `MEMORY`, `REGS`, `FLAGS`, `PC`. We use the following notation in the Boyer-Moore logic to describe this record.

```
Shell Definition.
Add the shell MACHINE with recognizer MACHINEP,
defining the record structure
<MEMORY, REGS, FLAGS, PC>.
```

The expression `(MACHINE M R FL PC)` represents a machine state with memory `M`, registers `R`, flags `FL` and program counter `PC`. The expressions `(MEMORY X)`, `(REGS X)`, `(FLAGS X)` and `(PC X)` respectively access the memory, registers, flags and program-counter fields of a machine `X`. If `X` is a machine as defined above, then the following expression represents the machine state equal to `X` in all fields but the program counter, which in this case takes on a value one less than `X`'s program counter.

```
(MACHINE (MEMORY X)
         (REGISTERS X)
         (FLAGS X)
         (SUB1 (PROGRAM-COUNTER X)))
```

Type restrictions are placed on the fields of a record structure by defining a predicate that constrains each field. In the `MACHINE` example, we might restrict each field to be either a machine word of a given size, or a finite array of machine words. The predicate `GOOD-MACHINE-STATE` in the following example constrains memory to be an array of length 2^{32} of 32-bit words, the registers to be an array of length 8 of 32-bit words, the flags to be a 4-bit word and the program counter to be a 32-bit word. (Assume `WORDP` recognizes a word of a particular size, and `WORD-ARRAY` recognizes an array of words of a particular size.)

```
DEFINITION
(GOOD-MACHINE-STATE X)
=
(AND (WORD-ARRAY (MEMORY X) 32)
     (EQUAL (LENGTH (MEMORY X)) (EXP 2 32))
     (WORD-ARRAY (REGS X) 32)
     (EQUAL (LENGTH (REGS X)) 8)
     (WORDP (FLAGS X) 4)
     (WORDP (PC X) 32))
```

An interpreter function models the behavior of a machine over a finite but arbitrary time span. It is a dyadic function of the form *Int*: $S \times O \rightarrow S$, where S a set of machine states and O a set of oracles for a machine. The two roles of an oracle are to determine the finite time span of the operation of a machine invocation, and to introduce non-deterministic state changes into a machine that include communication with other machines.

The set of natural numbers N can be chosen as the oracle set in a simple situation. An interpreter of the form *Int*: $S \times N \rightarrow S$ simulates a machine that operates in isolation. Such a machine is defined in the Boyer-Moore logic as shown below. The state of this machine progresses through the function `STEP`. The expression `(MACHINE1 STATE N)` is obtained by applying `N` successive applications of `STEP` to `STATE`.

```
DEFINITION
(MACHINE1 STATE N)
=
(IF (ZEROP N)
    STATE
    (MACHINE1 (STEP STATE) (SUB1 N)))
```

If we wish to stress that a machine state contains a stored program, we can define the step function as a fetch and an execute operation. In the function below, `FETCH` takes a machine state as an argument and returns an instruction, possibly a machine word. `EXECUTE` takes an instruction and a machine state and returns the machine state which results from executing the instruction.

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

In another situation, an oracle can be a list representing a finite time-sequenced series of external events impinging on a machine. The length of the oracle determines the time span of the machine operation. An element of the oracle may either be a single external event, or a symbol such as `'TICK` that indicates no event takes place. At each step, the interpreter consumes the next element of the oracle, continuing to run until the oracle is completely used up. The form of such an interpreter is defined by `MACHINE2`. In this example, as the function `CONSUME-INPUT` consumes the next element of the oracle, it combines it into the state of the machine so that the input is distinguishable to STEP. (The function CAR returns the first element of a list, while CDR returns everything but the first element of a list.)

```
DEFINITION
(MACHINE2 STATE ORACLE)
=
(IF (NOT (LISTP ORACLE))
    STATE
    (MACHINE2 (STEP (CONSUME-INPUT STATE (CAR ORACLE)))
              (CDR ORACLE)))
```

In this example, the step function can be set to model the interrupt structure of a machine. The function EXTERNAL-EVENTP recognizes conditions that require a response, such as a raised interrupt bit. For a von Neumann machine, RESPOND-TO-EVENT is most often a simple interrupt transition that carries out a partial CPU context switch.

```
DEFINITION
(STEP STATE)
=
(IF (EXTERNAL-EVENTP STATE)
    (RESPOND-TO-EVENT STATE)
    (EXECUTE (FETCH STATE) STATE))
```

3.2 Interpreter Equivalence Theorems

We wish to define an *implements* relation on two machines. Let Int_A: $S_A \times O_A \rightarrow S_A$ and Int_C: $S_C \times O_C \rightarrow S_C$ be interpreter functions which define two machines M_A and M_C. (The subscripts A and C are chosen to suggest *abstract* and *concrete* machines.) Let *MapUp*: $S_C \rightarrow S_A$ be an abstraction function which maps a concrete state to an abstract state, and let *MapDown*: $S_A \rightarrow S_C$ map an abstract state to a concrete state. We say that M_C *implements* M_A if the following theorem holds.

$$\begin{array}{ll} & \forall s_A \in S_A, \\ (1) & \forall o_A \in O_A, \\ & \exists o_C \in O_C \textit{ such that} \\ & MapUp\,(Int_C\,(MapDown\,(s_A), o_C)) = Int_A\,(s_A, o_A). \end{array}$$

This is a formalization of the commuting diagram depicted in figure 1-a.

Notice that there are various ways in which the proof of a theorem of this form can be approached. If it happens that $\forall s_A \in S_A$, $MapUp(MapDown(s_A)) = s_A$, then it is sufficient to prove

$$\begin{array}{ll} & \forall s_C \in S_C, \\ (2) & \forall o_A \in O_A, \\ & \exists o_C \in O_C \textit{ such that} \\ & MapUp\,(Int_C\,(s_C, o_C)) = Int_A\,(MapUp\,(s_C), o_A). \end{array}$$

To see this, substitute *MapDown* (s_A) for s_C in (2). Figure 1-(b) illustrates the correspondence which this

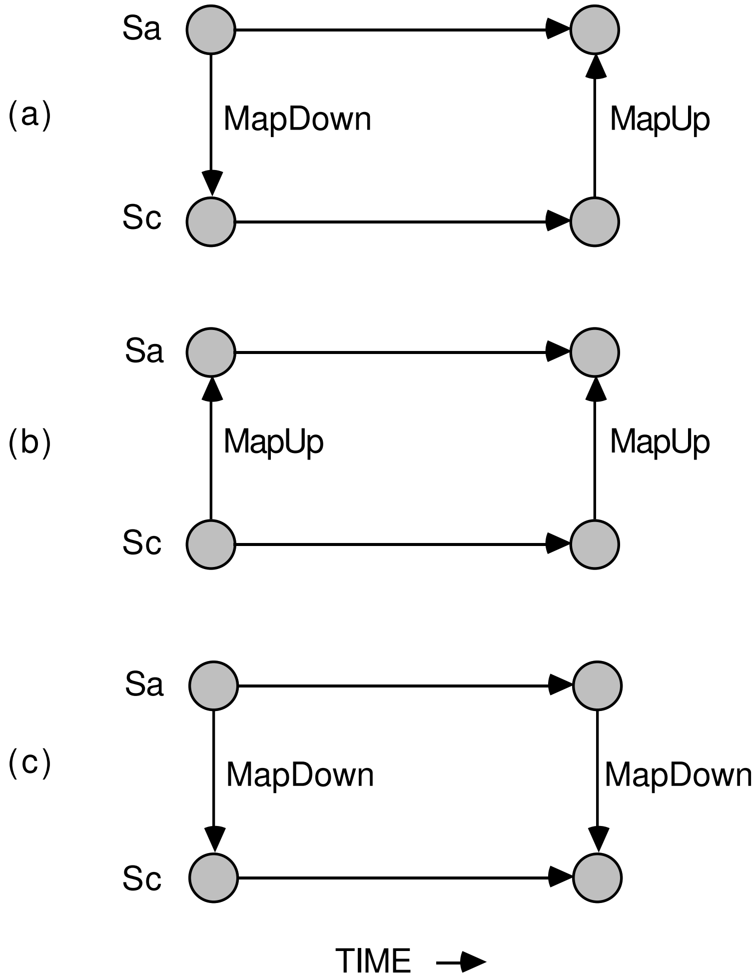

Figure 1: Interpreter Equivalence

theorem establishes. This is the general form, for example, of the interpreter equivalence theorem proven for our simple operating system Kit [2].

Similarly, if we can establish that $\forall\, s_C \in S_C$, $MapDown(MapUp(s_C)) = s_C$, then it suffices to prove

$$\begin{array}{ll} & \forall\, s_C \in S_C, \\ (3) & \forall\, o_A \in O_A, \\ & \exists\, o_C \in O_C \textit{ such that} \\ & MapDown\,(Int_A\,(s_A, o_A)) = Int_C\,(MapDown\,(s_A), o_C). \end{array}$$

This is the relation illustrated in the commuting diagram in figure 1-(c). This is the form of interpreter equivalence theorem established for our high-level language compiler [16], for example.

We cannot state any of (1), (2), or (3) in the quantifier-free Boyer-Moore logic. To address this difficulty, we replace the existential quantification with an explicit "witness" function ORACLE which computes the value of the concrete oracle. This is a function of both the initial abstract state and the initial abstract oracle.

For example, we can re-state (2) in the Boyer-Moore logic as follows. The predicate GOOD-CSTATE identifies an element of the set of concrete machine states.

```
THEOREM IMPLEMENTS-RELATION2
(IMPLIES (GOOD-CSTATE CSTATE)
         (EQUAL (MAPUP (INT-C CSTATE
                              (CORACLE ASTATE AORACLE)))
                (INT-A (MAPUP CSTATE) AORACLE)))
```

(3) is formalized analogously in the Boyer-Moore logic as follows.

```
THEOREM IMPLEMENTS-RELATION3
(IMPLIES (GOOD-ASTATE ASTATE)
         (EQUAL (MAPDOWN (INT-A ASTATE AORACLE))
                (INT-C (MAPDOWN ASTATE)
                       (CORACLE ASTATE AORACLE))))
```

3.3 Stacking Machines

Now suppose that we have interpreter equivalence results 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 2-(a). Under certain conditions, it is possible to compose the machine definitions to obtain the theorem represented by the commuting diagram in figure 2-(b). To be able to compose the commuting diagrams in this fashion it must be the case that the two abstract machine definitions "in the middle" correspond exactly. That is, the "concrete-level" machine for the upper diagram must be exactly the "abstract-level" machine for the lower diagram. Also, $\mathrm{MAPDOWN}_1$ must establish the GOOD-STATE predicate for the 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 interpreter equivalence 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 which is any number of levels less abstract. We can (and do) establish, for example, the correctness of the 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. Intermediate level machines represent the assembly language, machine language, and hardware functional design. The particular components of our verified stack are described in the following section.

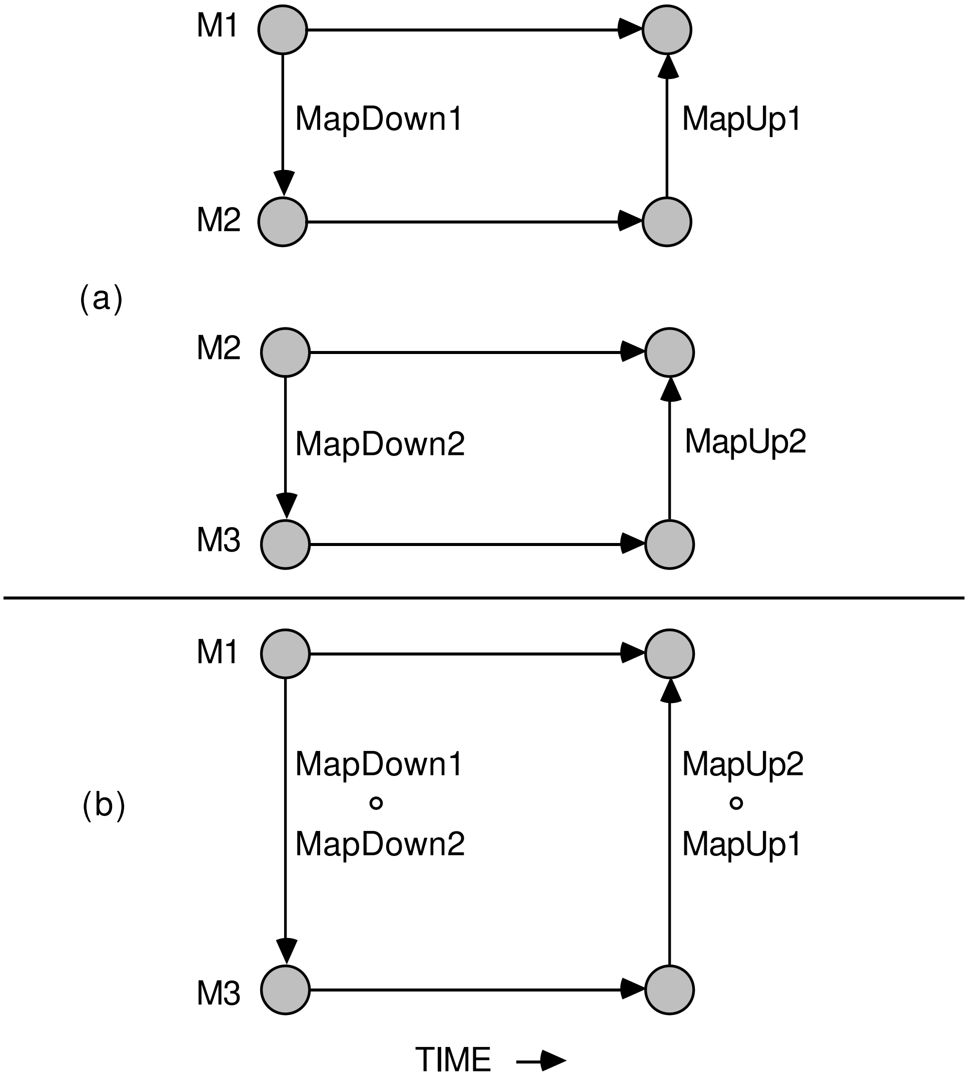

Figure 2: Composing Equivalence Theorems

4. The Short Stack and Kit

Using the techniques outlined in the previous section we have constructed and mechanically proved correct a collection of system components, several of which have been assembled into a verified *stack*. By this we mean that we have assembled a series of successively more "abstract" (farther from the hardware level) system components, each is implemented on its predecessor, and each implementation is formally specified and mechanically verified. The system we have constructed contains a microprocessor, a machine code, an assembly language, and a simple high-level programming language, together with the compiler, assembler and linker necessary to connect them.[1] We believe that our short stack is the first such verified stack.

We have also mechanically verified a small operating system kernel called *Kit* which implements a fixed number of conceptually distributed processes. The operating system kernel does not currently fit into our verified stack. Because of the similarity to the other projects and our intention to eventually fit Kit into our verified stack, we describe it here along with the other components. We discuss some of the issues of integrating Kit into the stack below.

4.1 The Short Stack

The short stack currently consists of four abstract machines:

- 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 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.
- FM8502—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, our machine lacks interrupts, supervisor/user modes, and support for virtual memory management.
- Piton—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.
- Micro-Gypsy—a high-level 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.

As described in section 3, relating each pair of adjacent machines is an *implementation*, represented 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 a MapUp function that is a partial inverse of the implementation. The MapUp functions permit one to inspect (parts of) the low level states and see high level objects "in" them.

---

[1]The compiler is actually a "cross-compiler;" it does not run on the host machine but is rather defined as a function in the logic. The assembler and linker are similarly defined.

The correctness of each implementation is formally characterized by a theorem in the Boyer-Moore logic representing an appropriate commutative diagram as explained above. We have proved each of the implementations correct. Put another way we have proved

- 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
- the correctness of a simple Micro-Gypsy program.

These proofs were all constructed by the Boyer-Moore theorem prover.[2] Complete descriptions of the relevant machines, implementations, MapUp functions, correctness theorems, and proofs are given in the accompanying papers in this collection [11, 14, 16].

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 3 illustrates the current verified short stack.

4.2 An Example of the Stack

The idea of having a formally verified connection between a program written in a high-level language and a machine code interpreter several layers away is so novel that we think it worthwhile to illustrate it.

Below is a Micro-Gypsy program in Gypsy syntax (in our actual proof Micro-Gypsy programs are represented in "internal form" and we have not yet implemented a parser — by far the easiest of the abstractions we manipulate here).

```
procedure MULT (var ANS: int;
                I, J: int) =
begin
   var K: int := 0;
   K := J;
   ANS := 0;
   loop
      if K le 0 then leave end;
      ANS := ANS + I;
      K := K - 1;
   end;
end; {mult}
```

---

[2] The proof of the Micro-Gypsy compiler used also the Kaufmann interactive interface to the prover.

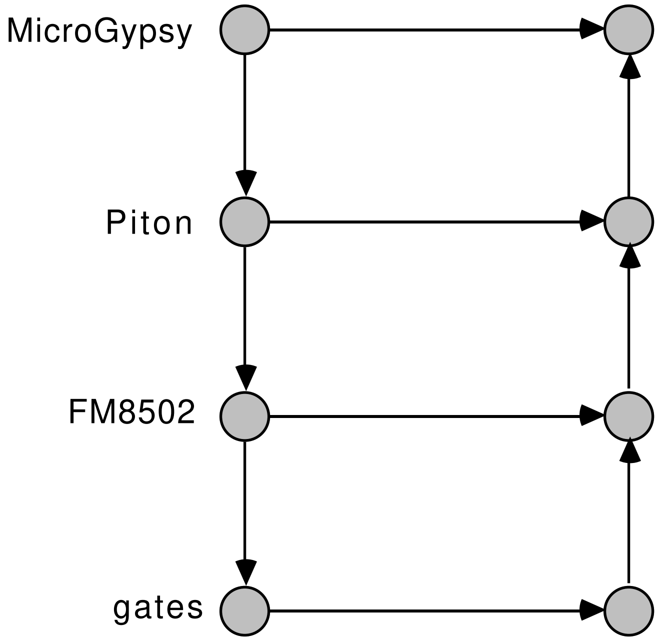

Figure 3: The Short Stack

This procedure multiplies two non-negative integers by repeated addition. The program is trivial, but that is not important.

The top-level specification of this program is that it executes without error provided `I` and `J` are sufficiently small non-negative integers, and it leaves the product, `I*J`, in the var parameter `ANS`. It is easy to prove this top-level specification by simply appealing to the formal definition of the Micro-Gypsy abstract machine in the Boyer-Moore logic.

If one executes the verified Micro-Gypsy compiler on this program one obtains a system of 15 Piton programs. If one then link-assembles that system, one obtains the FM8502 binary core-image partially shown in Figure 4.

The impact of having verified our short stack is that as soon as we have proved that the Micro-Gypsy program `MULT` multiplies, we can conclude that the binary core image in Figure 4 multiplies, and we can say how many machine instructions or microcycles it takes. Furthermore, all of this is done formally and mechanically within a single logical framework.

Figure 4: Part of an FM8502 Core Image

```
(M-STATE
'(B00000000000000000000001011000000 B00000000000000000000001111100011
  B00000000000000000000001111100000 B00000000000000000000001000100111
  B00000000000000000000000000000000 B00000000000000000000000000000000
  B00000000000000000000000000000000 B00000000000000000000000000000000)
F F F F
'(B00000000000011111000001001000001 B00000000000011111000000000100010
  B00000000000011111000001001011011 B00000000000011111000001001011011
  B00000000000011111000000010011000 B00000000000000000000000000000001
  B00000000000000110000000010000010 B00000000000011111000001001101100
  B00000000000011111000000010111011 B00000000000000010000000010100101
  B00000000000011111000000010011000 B00000000000000000000010001001101
  B00000000000011111000000010001100 B00000000000001110000000010000101
  B00000000000011111000001001101100 B00000000000011111000000010011000
  B00000000000000000000000000000000 B00000000000000110000000010000010
  B00000000000011111000001001101100 B00000000000011111000000010111011
  B00000000000000010000000010100101 B00000000000011111000000010011000
  B00000000000000000000010001001101 B00000000000011111000000010001100
  B00000000000001110000000010000101 B00000000000011111000000110011011
  B00000000000011111000000000001000 B00000000000000000000000000011100
  B00000000000011111000000001000001 B00000000000011111000000000111010
  B00000000000011111000000000011010 B00000000000011111000001001000001
  B00000000000011111000000000100010 B00000000000011111000001001011011
  B00000000000011111000001001011011 B00000000000011111000000010011000
  B00000000000000000000000000000001 B00000000000000110000000010000010
  B00000000000011111000001001101100 B00000000000011111000000010011000
  B00000000000000000000000000000000 B00000000000000110000000010000010
  B00000000000011111000001001101100 B00000000000011111000000010111011
  B00000000000000010000000010100101 B00000000000011111000000010011000
  B00000000000000000000010001001101 B00000000000011111000000010001100
  B00000000000001110000000010000101 B00000000000011111000000110011011
  B00000000000011111000000000001000 B00000000000000000000000000110100
  B00000000000011111000000001000001 B00000000000011111000000000111010
  B00000000000011111000000000011010 B00000000000011111000001001000001
  B00000000000011111000000000100010 B00000000000011111000001001011011
  B00000000000011111000001001011011 B00000000000011111000001001011011
  B00000000000011111000000010011000 B00000000000000000000000000000001
  B00000000000000110000000010000010 B00000000000011111000001001101100
  B00000000000011111000000010111011 B00000000000000010000000010100101
  B00000000000011111000000010011000 B00000000000000000000010001001101
  B00000000000011111000000010001100 B00000000000001110000000010000101
  B00000000000011111000001001101100 B00000000000011111000000010011000
  B00000000000000000000000000000010 B00000000000000110000000010000010
  B00000000000011111000001001101100 B00000000000011111000000010111011
  B00000000000000010000000010100101 B00000000000011111000000010011000
  B00000000000000000000010001001101 B00000000000011111000000010001100
  ...))
```

4.3 The Kit Operating System

Using the same techniques described above we have implemented and proved correct a simple operating system, called *Kit*. 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 "small isolated target machines" capable of communicating only through shared I/O buffers. The full specification of Kit, the correctness theorem, and its proof are described in [2]. 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 (or "target machine") is very similar to FM8502, above, but was developed concurrently and more or less independently. A consequence of this is that Kit does not fit precisely into our short 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 short 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 [9], and I/O. See [2] for a discussion of how an operating system can be placed in the stack.

5. Conclusion

The key to our approach to systems verification is the use of formally-defined abstract machines. These machines are defined explicitly as functions in the Boyer-Moore logic. Their properties can be proved as theorems in the logic, using the Boyer-Moore theorem prover.

We use formally defined abstract machines for two reasons. One is their flexibility: they can be used to formalize a wide variety of systems. The other is their uniformity: if they are all defined in a single language, then that language can be used to relate one abstract machine to another. This latter remark is the key to hierarchical system verification and the formal explanation of abstraction.

The idea of producing a verified system by stacking machine descriptions of increasing abstraction is not new. (The papers which describe the individual components of our system [16, 14, 2, 11] contain references to previous work.) What is new is the degree of integration we have been able to achieve among our verified components.

6. Acknowledgements

This work was supported in part at Computational Logic, Inc. by the Defense Advanced Research Projects Agency, ARPA Orders 6082 and 9151. The views and conclusions contained in this paper are those of the author 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.

References

1. W.R. Bevier. Kit: A Study in Operating System Verification. Tech. Rept. 28, Computational Logic, Inc., 1717 W. 6th St., Suite 290, Austin, Texas, 78703, March, 1988>.

2. W.R. Bevier. Kit and the Short Stack. To appear in The Journal of Automated Reasoning.

3. R.S. Boyer, J S. Moore. Metafunctions: Proving Them Correct and Using them Efficiently as New Proof Procedures. In R.S. Boyer, J S. Moore, Ed., *The Correctness Problem in Computer Science*, Academic Press, London, 1981.

4. R. S. Boyer and J S. Moore. "A Fast String Searching Algorithm". *Comm. ACM 20*, 10 (1977), 762-772.

5. R. S. Boyer and J S. Moore. A Verification Condition Generator for FORTRAN. In *The Correctness Problem in Computer Science*, R. S. Boyer and J S. Moore, Eds., Academic Press, London, 1981.

6. R. S. Boyer and J S. Moore. MJRTY - A Fast Majority Vote Algorithm. Tech. Rept. ICSCA-CMP-32, Institute for Computing Science and Computer Applications, University of Texas at Austin, 1982. Also available through Computational Logic, Inc., Suite 290, 1717 West Sixth Street, Austin, TX 78703..

7. R. S. Boyer and J S. Moore. "Proof Checking the RSA Public Key Encryption Algorithm". *American Mathematical Monthly 91*, 3 (1984), 181-189.

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

9. D.I. Good, R.M. Cohen, J. Keeton-Williams. Principles of Proving Concurrent Programs in Gypsy. Tech. Rept. ICSCA-CMP-15, Institute for Computer Science and Computing Applications, The University of Texas at Austin, January, 1979.

10. M. J. Gordon, A. J. Milner, and C. P. Wadsworth. *Edinburgh LCF.* Springer-Verlag, New York, 1979.

11. W.A. Hunt. Microprocessor Design Verification. To appear in The Journal of Automated Reasoning.

12. Matt Kaufmann. A User's Manual for an Interactive Enhancement to the Boyer-Moore Theorem Prover. Tech. Rept. 19, Computational Logic, Inc., 1717 W. 6th St., Suite 290, Austin, Texas, 78703, May, 1988.

13. Lengauer, C., and Huang, C.-H. A Mechanically Certified Theorem about Optimal Concurrency of Sorting Networks. Proc. 13th Ann. ACM Symp. on Principles of Programming Languages, 1986, pp. 307-317.

14. J S. Moore. A Mechanically Verified Language Implementation. To appear in The Journal of Automated Reasoning. Also available as Technical Report 30, Computational Logic, Inc., 1717 W. 6th St., Suite 290, Austin, Texas, 78703.

15. N. Shankar. Proof Checking Metamathematics. University of Texas at Austin, 1986. Also available through Computational Logic, Inc., Suite 290, 1717 West Sixth Street, Austin, TX 78703..

16. W.D. Young. A Mechanically Verified Code Generator. To appear in The Journal of Automated Reasoning. Also available as Technical Report 36, Computational Logic, Inc., 1717 W. 6th St., Suite 290, Austin, Texas, 78703.

Table of Contents

List of Figures