4 Mechanized Formal Reasoning about Programs and Computing Machines

Robert S. Boyer
University of Texas at Austin

J Strother Moore
Computational Logic, Inc., Austin, Texas

The design of a new processor often requires the invention and use of a new machine-level programming language, especially when the processor is meant to serve some special purpose. It is possible to specify formally the semantics of such a programming language so that one obtains a simulator for the new language from the formal semantics. Furthermore, it is possible to configure some mechanical theorem provers so that they can be used directly to reason about the behavior of programs in the new language, permitting the expeditious formal modeling of the new design as well as experimentation with and mechanically checked proofs about new programs. We here study a very simple machine-level language to illustrate how such modeling, experimentation, and reasoning may be done using the ACL2 automated reasoning system. Of particular importance is how we control the reasoning system so that it can deal with the complexity of the machine being modeled. The methodology we describe has been used on industrial problems and has been shown to scale to the complexity of state-of-the-art processors.

4.1 Historical Background

Why write about how to formalize the semantics of a simple programming language? The answer is that it is a skill necessary to the application of formal reasoning tools to industrial microprocessor design. We advocate, in fact, the use of an operational semantics expressed in a computational logic, e.g., one supporting execution. We do so for two reasons: such a semantics is usually accessible to the engineers involved and is unusually effective in enabling mechanized proof. The last two decades of our research can be seen as an effort to demonstrate the latter claim.

In the late-1970's we were concerned with proving the correctness of a program written in the machine code for the Bendix BDX930 computer [10]. At the time we wrote:

> To capture the semantics of the instruction set, we encoded in our logic a recursive function that describes the state changes induced by each BDX930 instruction. Thirty pages are required to describe the top level

> driver and the state changes induced by each instruction (in terms of certain still undefined bit-level functions such as the 8-bit signed addition function). We encountered difficulty getting the mechanical theorem-prover to process such a large definition. However, the system was improved and the function was eventually admitted. We still anticipate great difficulty proving anything about the function because of its large size.

The formal system we used in the Bendix 930 work was the early version of Nqthm [2, 4]. The "large size" of our BDX930 specification drove much of our work on Nqthm's implementation and how to use a formal logic to specify a von Neumann computing machine. By the mid-1980's, Nqthm was able to handle an RTL-level description of a "home-grown" microprocessor, the FM8501, a specification of its machine code, and the proof that the RTL description implemented the machine code [11]. Later, the FM8502 was produced, a 32-bit wide version of the 16-bit wide FM8501. By the late 1980's, Nqthm had

been used to verify the "CLI Stack," which provided the FM8502 with a linking assembler and a compiler for a simple high-level language [1]. The stack also provided a simple operating system kernel for a machine similar to the FM8502. The stack work was carried out by the two authors, our colleague Matt Kaufmann, and our students Bill Bevier, Warren Hunt, and Bill Young.

However, as late as 1990, the CLI stack was still only a theoretical exercise because the FM8502 could not be practically fabricated from its RTL-level description. This changed in 1991, when Bishop Brock and Warren Hunt produced a verified microprocessor, the FM9001 [12]. They formalized the Netlist Description Language (NDL) of LSI Logic, Inc., in the Nqthm logic and then used that formal language to describe a microprocessor similar to the FM8502. After the correspondence of the NDL description and the machine code specification was proved, the NDL description was sent to LSI Logic, Inc., and a chip was fabricated. The rest of the stack was ported to the FM9001 by re-targeting and re-verifying the link assembler, a process that was almost entirely automatic [19]. In addition, our student Matt Wilding implemented some applications programs on the FM9001 (using the link assembler) and proved them correct [23].

Finally, in 1991, our student Yuan Yu used Nqthm to formalize 80% of the user mode instruction set of a commercial microprocessor, the Motorola MC68020 [25, 5] and then used the formal model to verify many binary machine code programs produced by commercial compilers from source code in such high-level languages as Ada, Lisp and C. For example, Yu verified the MC68020 binary code produced by the "gcc" compiler for 21 of the 22 C programs in the Berkeley string library.

Mechanized formal reasoning is now being used in an experimental fashion in support of industrial microprocessor design. Such projects are carried out in collaborations between the industrial design team and the developers of the formal reasoning systems.

The PVS system [21] was used to specify about half of the instructions on the AAMP5, an avionics processor developed by Rockwell-Collins, and the microcode of about a dozen instructions was verified against the formal specification [17]. The ACL2 system [13, 14, 9]—a successor of Nqthm—was used to specify Motorola's Complex Arithmetic Processor (CAP) and many theorems were proved about it, including the correctness of an abstraction that eliminates the pipeline and the correctness of several microcode programs involved in digital signal processing (DSP) [9]. In addition, ACL2 has been used to verify the microcode for both floating-point division [20] and square root [22] on the AMD5$_K$86, the first Pentium-class microprocessor produced by Advanced Micro Devices, Inc.

Common to these projects is the formalization of a machine-code-like programming language and the configuration of a mechanized reasoning system so that it can be used directly in the construction of proofs about programs in that language. The development of reasoning tools for standard languages such as VHDL will undoubtedly remove some of the necessity of "rolling your own" formal semantics. The Brock-Hunt NDL formalization for FM9001, for example, can be reused to verify other hardware designs [18]. But as long as microprocessors support new, special-purpose machine languages—i.e., as long as designers produce special-purpose microprocessors such as the now-old fashioned BDX930, or the brand new AAMP5 and CAP—formal mechanical verification will require the formalization of new machine-level languages and the construction, one way or another, of mechanical aids to reasoning.

We hope that this formalization will someday be done by the engineers on the project. This is not a farfetched notion. After all, it is the engineers on the project who are routinely "rolling their own" languages; they simply do not so often write down the semantics precisely. One great benefit of doing so is that it increases the clarity and precision of the communication between team members when discussing changes to the evolving design. We believe that an operational semantics, where one defines the effect of each instruction, is often quite similar to the informal notions the team members would otherwise employ to think about their language. A formal operational semantics can immediately provide another great benefit to the design team: an execution or simulation capability for the new language. Therefore, we think it is both feasible and desirable for a design team to "roll their own" formal semantics.

But special-purpose processors—especially those in which mechanically checked proofs are seen as desirable—often have arcane and complex instruction sets, for otherwise a standard commercial chip would have been used. For example, the CAP has six independent memories, four multiplier-accumulators, over 250 programmer-visible registers, and allows instructions which simultaneously modify well over 100 registers. The microcode has a 64-bit instruction word that decodes to a 317-bit control word internally. In is not unusual in typical CAP DSP application programs to encounter instructions that simultaneously modify several dozen registers. Indeed, it is that very complexity that encourages the design team to seek reassurance in mechanically checked arguments.

Once the formal semantics is written down, it is often desirable to have mechanical aid to support formal reasoning. A wonderful and obvious solution would be to construct a special-purpose GUI integrated into the team's CAD tools and which uses a theorem prover "behind the scenes." Unfortunately, such systems take a great deal of time to develop but are needed *before* the language/machine is completely designed. Hence it is often expedient to define the semantics in a formal logic in the first place and then use an existing theorem proving tool directly. The direct use of a general-purpose theorem prover driven by the evolving formal semantics of the machine is in conflict with the "wonderful and obvious" solution in which the formal machinery is hidden from the user. But until mechanical theorem provers are smarter or the pace slows down so that applications- and language-specific tools can be built, we believe this is the most cost-effective scenario.

That it is possible to so use a mechanical theorem prover has now been demonstrated repeatedly in the projects cited above. We cite especially the case study of the CAP described in [9]. Such demonstrations are eloquent testimony to the advances in automated reasoning since the late '70s. But we are left with a problem: somebody on the team must be prepared to formalize an evolving complex language and configure an existing theorem prover to make it relatively straightforward to reason about the new programs being written. There is no doubt, today, that the person must be intimately familiar with the theorem prover, but we hope that will change. We believe tools can be built to make the process easier. But in the meantime, it is important to explain how it is done. The formalization of such complex languages is difficult to do. It is even more difficult to learn how to do formalization by studying the industrial examples cited above. Hence this exposition.

We describe and then formalize the semantics of a simple language and then give practical advice for how to prove theorems about programs written in the language.

The approach we describe is essentially that used in the Nqthm and ACL2 projects described above. Furthermore, the Nqthm and ACL2 users above were taught this method of formalization via examples very similar to this one, primarily in our graduate class, *Recursion and Induction*, at the University of Texas at Austin. That this technique scales up to languages that are many orders of magnitude more complicated than this one is demonstrated by [5, 9]. Therefore, simplicity *here* should be looked upon as a virtue.

We will use ACL2 as the formal system in which the semantics is expressed and the proof advice is given. ACL2 is merely an axiomatization of an applicative subset of Common Lisp. The reader familiar with some Lisp will have no trouble understanding our formalization. For readers unfamiliar with Lisp we informally paraphrase each important formula. We believe that the value of this approach is independent of ACL2 or any particular formalism. Therefore, we urge readers who use other systems to read on.

To obtain ACL2 by ftp, first connect to ftp.cli.com by anonymous login. Then 'cd' to /pub/acl2/v1-8, 'get README' and follow the directions therein. Alternatively, one may obtain ACL2 via the World Wide Web with the URL http://www.cli.com/-software/acl2/index.html. The ACL2 system includes extensive hyper-linked on-line documentation available via ACL2 documentation commands, HTML browsers, Emacs Info, and in hardcopy form. See [13]. Contact the authors to obtain the file of ACL2 definitions and theorems described here.

4.2 A Simple Machine-Level Language

4.2.1 States

The state of our machine will be represented by a 5-tuple. We define the function `statep` to recognize lists of length 5,

```
(defun statep (x)
  (and (true-listp x)
       (equal (len x) 5))) .
```

This definition actually introduces an axiom defining a new function, `statep`, of one argument, `x`. The function returns `t` (true) if `x` is a "true list" (a binary tree whose right-most branch terminates in `nil`) and its length is 5. Otherwise `statep` returns `nil`. In Lisp notation, the application of the function symbol `statep` to its single argument, `x`, is written `(statep x)` as opposed to the more traditional `statep`(x) notation.

Our states will consist of a program counter, `pc`; a control stack of suspended program counters, `stk`; the data memory, `mem`; a flag, `halt`, indicating whether the machine is halted; and an "execute only" program memory, `code`. Our machine does not have a separate "register file;" instead we use low memory addresses as though they named registers.

In Common Lisp we define the five "accessors" as follows.

```
(defun pc   (s) (nth 0 s))          ; pc(s)   = nth(0,s)
(defun stk  (s) (nth 1 s))          ; stk(s)  = nth(1,s)
(defun mem  (s) (nth 2 s))          ; mem(s)  = nth(2,s)
(defun halt (s) (nth 3 s))          ; halt(s) = nth(3,s)
(defun code (s) (nth 4 s))          ; code(s) = nth(4,s)
```

For example, the first `defun` defines the function `pc` to take one argument, `s` and to return the 0^{th} element of `s`. Remarks after semicolons are comments.

We could write `(list` x_1 x_2 x_3 x_4 x_5`)` to denote a state whose components are the five objects x_1, ..., x_5. But such positional notation can be difficult to decipher. We prefer the notation `(st :pc` x_1 `:stk` x_2 `:mem` x_3 `:halt` x_4 `:code` x_5`)` simply because it reminds us of what each component represents. The order of the key/value pairs is unimportant.

New states are often obtained from old ones by "changing" only a few components and leaving the others unchanged. We introduce notation so that, for example, `(modify` s `:pc` x_1 `:halt` x_2`)` is the state whose components are the same as those of s except that the `pc` is x_1 and the `halt` flag is x_2. That is, the `modify` expression above denotes `(list` x_1 `(stk` s`) (mem` s`)` x_2 `(code` s`))`. We omit the definition of `modify`.

4.2.2 Memory, Code, and Program Counters

The machine we have in mind provides a finite (but arbitrarily sized) memory. We enumerate the memory locations from 0; each location contains an arbitrary object. Hence a memory can be represented by a finite list of the objects it contains. To refer to the contents of location `n` in memory `mem` we use `(nth n mem)`. To change the contents of location `n` to `v` we use `(put n v mem)`. The function `put` is a simple recursively defined list processing function.

```
(defun put (n v mem)
  (if (zp n)
      (cons v (cdr mem))
      (cons (car mem) (put (- n 1) v (cdr mem)))))
```

Roughly speaking, to `put` `v` at position `n` in `mem`, put it at the front if `n` is 0 and otherwise put it into position `n-1` in `(cdr mem)` (the list containing all but the first element of `mem`) and then add `(car mem)` (the first element of `mem`) onto the front. For example, to put 77 at position 2 in `'(0 1 2 3 4)` we use `(put 2 77 '(0 1 2 3 4))` , which is equal to `'(0 1 77 3 4)`.

What does `put` do if `n` is negative or not an integer? The answer is that it acts as though `n` were 0. How? `(Zp n)` is defined to be true if `n` not a positive integer. What if `n` $\geq$ `(len mem)`? In this case we "cdr off the end" of `mem`. But `cdr` is a total function that returns `nil` if its argument is not a list. Hence, `put` essentially "coerces" `mem` to be long enough by extending it with `nil`s. This allows many theorems about `put` to be stated accurately without having to have hypotheses about `n` and `mem`. The simplicity of both the definition of `put` and theorems about it is one of the strengths of both Nqthm and ACL2.

In our machine, the "program memory," `code`, will be represented by an association list in which each program name is paired with the list of instructions in it. Thus, for example, a program memory containing three programs, named `times`, `expt` and `main`, will be represented by the list

```
'((times ins_{0,0} ins_{0,1} ... ins_{0,k_0})
  (expt  ins_{1,0} ins_{1,1} ... ins_{1,k_1})
  (main  ins_{2,0} ins_{2,1} ... ins_{2,k_2})) .
```

Given the name of a program, `name` and a program memory, `code`, `(cdr (assoc name code))` returns the associated list of instructions.

Individual instructions will be lists. Abstractly every instruction will have an `opcode` and two arguments, `a` and `b`.

```
(defun opcode (ins) (nth 0 ins))
(defun a (ins) (nth 1 ins))
(defun b (ins) (nth 2 ins))
```

Because `nth`, like `put`, extends its list argument with `nil`s, we can write instructions in three formats: (*op*), (*op a*), and (*op a b*) and omitted arguments default to `nil`.

For example, the constant

```
(times (movi 2 0)    ; 0   mem[2] ← 0
       (jumpz 0 5)   ; 1   if mem[0]=0, go to 5
       (add 2 1)     ; 2   mem[2] ← mem[1] + mem[2]
       (subi 0 1)    ; 3   mem[0] ← mem[0] - 1
       (jump 1)      ; 4   go to 1
       (ret)))       ; 5   return to caller
```

defines one program in our language. The constant is a list of seven elements. The first, `times`, is the name of the program and the other six elements are the instructions. For example, the first instruction is `(movi 2 0)`, which has an `opcode` of `movi`, an `a` argument of `2` and a `b` argument of `0`; the last instruction is `(ret)`, which has an `opcode` of `ret` and `a` and `b` arguments of `nil`. A typical `code` memory will contain many such lists, one for each program in the machine. We will use `times` repeatedly and we define the function `times-program` to be the constant function which returns the above constant. That is,

```
(defun times-program ()
  '(times (movi 2 0) ... (ret)))
```

and so `(times-program)` is equal to the constant above. We discuss the program later.

Returning now to the formalization of our machine, the program counter, `pc`, will be a pair, `(`*name* `.` *i*`)`, indicating that the "current instruction" is the i^{th} instruction in the program named *name*. Thus, to fetch an instruction we use

```
(defun fetch (pc code)
  (nth (cdr pc)
       (cdr (assoc (car pc) code)))) ,
```

and so the current instruction of a state is

```
(defun current-instruction (s)
  (fetch (pc s) (code s))) .
```

To construct the program counter for the first instruction in the program named `name` we use `(cons name 0)`. To increment a program counter by one we use the function

```
(defun pc+1 (pc)
  (cons (car pc) (+ 1 (cdr pc)))) .
```

Thus, if `pc` is `'(times . 3)` then `(pc+1 pc)` is `'(times . 4)`.

4.2.3 Instructions Semantics

We can now begin the main task, namely the specification of the individual instructions on our machine. We will define only as many instructions as it takes to write a couple of simple programs.

It will be our convention to define a function, called the *semantic function*, for each instruction. The function will have the same name as the opcode of the instruction. The function will take one more argument than the instruction does and that extra argument shall be the machine's state.

Consider the `move` instruction, which has two arguments, the target address and the source address, and is written as the list constant `(move` *a b*`)`. Here is the semantic function for the `move` operation,

```
(defun move (a b s)
  (modify s
          :pc (pc+1 (pc s))
          :mem (put a (nth b (mem s)) (mem s)))) .
```

The function modifies `s` in two places. The `pc` is incremented by one and the contents of location `b` is deposited into location `a`.

Contrast this with the semantics of the "move immediate" instruction, `movi`,

```
(defun movi (a b s)
  (modify s
          :pc (pc+1 (pc s))
          :mem (put a b (mem s)))) ,
```

which deposits `b`, not its contents, into location `a`.

We provide two arithmetic instructions.

```
(defun add (a b s)
  (modify s
          :pc (pc+1 (pc s))
          :mem (put a
                    (+ (nth a (mem s))
                       (nth b (mem s)))
                    (mem s))))
```

`Add` adds the contents of the two locations and deposits the sum in the first location.

```
(defun subi (a b s)
  (modify s
          :pc (pc+1 (pc s))
          :mem (put a
                    (- (nth a (mem s)) b)
                    (mem s))))
```

`Subi` subtracts the second argument, not its contents, from the contents of the first and deposits the difference in the first.

To branch unconditionally within the current program we provide

```
(defun jump (a s)
  (modify s :pc (cons (car (pc s)) a))) .
```

Here, the argument `a` is the location within the current program to which control is to be transferred. Observe that we stay within the current program since we do not change the name component of the `pc`.

To branch conditionally within the current program we provide

```
(defun jumpz (a b s)
  (modify s
          :pc (if (zp (nth a (mem s)))
                  (cons (car (pc s)) b)
                  (pc+1 (pc s))))) .
```

That is, if the contents of `a` is 0[1], jump to location `b` in the current program; else increment the `pc` by 1.

To call a subroutine we provide

```
(defun call (a s)
  (modify s
          :pc (cons a 0)
          :stk (cons (pc+1 (pc s)) (stk s)))) .
```

`Call` modifies the stack by incrementing the current `pc` by 1—to create the desired return program counter—and then pushing it onto the stack of suspended program counters. The `pc` is set to the first instruction in the program named `a`.

Finally, to return from a subroutine call (or to halt) we provide

```
(defun ret (s)
  (if (endp (stk s))
      (modify s :halt t)
      (modify s
              :pc (car (stk s))
              :stk (cdr (stk s))))) .
```

Observe that if the stack is empty when `ret` is executed, the `halt` flag is set. Otherwise, the stack is popped and the top-most return program counter from it becomes the new `pc`.

4.2.4 The Fetch-Execute Cycle

To tie the semantic functions to their instructions we define the function `execute` which takes an instruction `ins` and a state `s` and executes the (semantic function for) `ins` on `s`.

[1] We find it convenient actually to test `(zp a)`, which is true if `a` is not a positive integer.

```
(defun execute (ins s)
  (let ((op (opcode ins))
        (a (a ins))
        (b (b ins)))
    (case op
          (move  (move a b s))
          (movi  (movi a b s))
          (add   (add a b s))
          (subi  (subi a b s))
          (jumpz (jumpz a b s))
          (jump  (jump a s))
          (call  (call a s))
          (ret   (ret s))
          (otherwise s))))
```

This syntax may be read "In the following, let `op`, `a` and `b` be the corresponding parts of the instruction `ins`. If `op` is `'move` the result is `(move a b s)`, if `op` is `'movi`, the result ..., otherwise the result is `s`." Observe that if an unrecognized instruction is executed it is a no-op.

The machine's fetch-execute operation or "single stepper" is

```
(defun step (s)
  (if (halt s)
      s
      (execute (current-instruction s) s))) .
```

That is, if the `halt` flag is set, we return `s` and otherwise we execute the current instruction on `s`.

We can step the state `s` `n` times with

```
(defun sm (s n)
  (if (zp n)
      s
      (sm (step s) (- n 1)))) .
```

The name `sm` stands for "small machine" and represents the formal model of the machine's fetch-execute cycle.

4.3 A Simulation Capability

Recall

```
(defun times-program ()
 '(times (movi 2 0)    ; 0   mem[2] ← 0
(jumpz 0 5)   ; 1   if mem[0]=0, go to 5
(add 2 1)     ; 2   mem[2] ← mem[1] + mem[2]
(subi 0 1)    ; 3   mem[0] ← mem[0] - 1
(jump 1)      ; 4   go to 1
(ret))))      ; 5   return to caller.
```

Informally, the specification of this program is that it multiplies the contents of memory location 0 times that of location 1 and leaves the result in location 2. The program uses the method of successive addition of location 1 into location 2, which is initially cleared. The comments explain further. Note that the program loops through `pc`s 1–4, decrementing location 0.

We can use the definition of `sm` to run this program. That is, by virtue of having defined the semantics of our machine in a programming language, we get a simulation capability "for free." Below we define a constant,

```
(defun demo-state ()
  (st :pc   '(times . 0)
      :stk   nil
      :mem  '(7 11 3 4 5)
      :halt nil
      :code (list (times-program)))) ,
```

which is a state poised to multiply 7 times 11. Note first the `code` memory; it contains exactly one program, namely that for `times`. The `pc` points to the first instruction in `times`. The stack is empty, so the `ret` instruction in `times` will halt the machine. The memory has five locations, with 7 and 11 in the first two and 3, 4, and 5 in the remaining three. The `halt` flag is off.

How many steps does it take to run this program on this data? One tick executes the initialization at `pc` 0 and gets us to the top of the loop at `pc` 1. The loop takes four ticks and is executed 7 times, leaving us at the top of the loop with location 0 set to 0. One more tick executes the `jumpz` which transfers control out of the loop to the `ret`. One more tick executes the `ret` and halts the machine. Thus, it takes $1 + (7 \times 4) + 2 = 31$ ticks to execute the program here.

What is the final state? Obviously, the `pc` will point to the `ret`, the stack will (still) be empty, the `halt` flag will be set and the `code` will be unchanged. What about the memory? Location 0 will have been counted down to 0. Location 1 will be unchanged at 11. Location 2 will have accumulated $7 \times 11 = 77$. And locations 4 and 5 are untouched. We can actually prove that this is the final state, simply by evaluating `(sm (demo-state) 31)` and looking at the answer. This can be stated as a theorem,

```
(defthm demo-theorem
  (equal (sm (demo-state) 31)
         (st :pc   '(times . 5)
             :stk  nil
             :mem  '(0 11 77 4 5)
             :halt t
             :code (list (times-program))))) .
```

This theorem is trivial to prove by evaluation. That is, `(sm (demo-state) 31)`, if executed in any Common Lisp containing ACL2 and the definitions shown above, will create the state shown on the right-hand side of the equation above.

4.4 A Specification of the Program

Intuitively, `times` multiplies the contents of memory locations 0 and 1, clears location 0, and writes the product into location 2. If a state is poised to execute a call of `times` and we step it exactly enough to execute through the return statement for that call, then the effect on memory is as just described and the program counter is incremented by one. This can be said much more precisely as follows. Let `s0` be a state whose memory contains at least three locations (in general we wish to make sure all the memory references in the program are legal). Let `i` and `j` be the contents of the first two locations and suppose that `i` and `j` are natural numbers. Suppose that the current instruction in `s0` is a call of `times`, where `times` is defined as in `times-program`. Finally, suppose the `halt` flag is off, so the state is runnable. Then if we run `sm` on `s0` for a certain number of ticks, namely `(times-clock i)`, defined below, the result is the state obtained more simply by incrementing the program counter of `s0` by one, writing a 0 into memory location 0, and writing the product of `i` and `j`, `(* i j)`, into memory location 2.

A formal rendering of this specification of `times` is shown below.

```
(defthm times-correct
  (implies (and (statep s0)
                (< 2 (len (mem s0)))
                (equal i (nth 0 (mem s0)))
                (equal j (nth 1 (mem s0)))
                (natp i)
                (natp j)
                (equal (current-instruction s0) '(call times))
                (equal (assoc 'times (code s0)) (times-program))
                (not (halt s0)))
           (equal (sm s0 (times-clock i))
(modify s0
        :pc (pc+1 (pc s0))
        :mem (put 0 0
                  (put 2 (* i j) (mem s0)))))))
```

The expression `(times-clock i)` is supposed to measure how many clock ticks it takes to execute `times` to multiply `i` times `j`. We call `times-clock` the *clock function* for the program `times`. Recall that in `demo-theorem` the time taken to run `times` on the 7×11 problem was $1 + (7 \times 4) + 2 = 31$. In that example the run was started at the top of the `times` program itself, at pc `(times . 0)`. In our more general specification, the run starts at a call of `times`, thus adding one instruction to the count. Therefore, `(times-clock i)` is equal to `(+ 2 (+ (* i 4) 2))`.

Note that our specification of `times` not only describes its functional behavior completely (i.e., every effect on the state) but also characterizes the number of clock ticks it takes. It is often easier to prove such strong theorems than to prove weaker ones. Indeed, the clock plays a crucial role in our proof strategy.

In what follows, we will show how to prove `times-correct` and other such theorems. Of particular importance is the *system decomposition* problem: how can we verify complex programs by verifying their component subroutines? Furthermore, we are not so much interested in proving theorems as in proving them mechanically without building any special purpose reasoning system for that task (beyond a theorem prover already presumed to exist).

4.5 How to Prove the Program Correct

To prove `times` correct we do symbolic computation, using induction to get around the loop. That is the obvious thing to do; the question is how do we carry it out formally and mechanically within the logic?

4.5.1 The ACL2 Proof Style

ACL2, like Nqthm, employs two main proof techniques, induction and rule-driven simplification. For Nqthm, these techniques are discussed thoroughly in [2]; ACL2's techniques are quite similar and are described in the ACL2 documentation [13].

The system inducts only when its other heuristics do not apply. To choose an induction, the system inspects the recursive definitions of the functions involved in the conjecture and selects an induction designed to "unwind" one or more of those functions. We discuss induction further when we describe a particular use of it below.

Of more concern here is rule-driven simplification. ACL2's rewriter attempts to simplify formulas by rewriting them using axioms, definitions, and previously proved theorems as rewrite rules. When the rewriter is called, it is given a list of "assumptions," terms which are assumed true. When the rewriter descends through `(if` *test a b*`)`, it adds *test* to the assumptions while rewriting a and `(not` *test*`)` while rewriting b. Heuristics control the "expansion" of recursive function definitions; the basic idea is to expand a function call into its body provided the simplified body is "simpler" or the simplified arguments to all recursive calls already occur elsewhere in the problem. Axioms and theorems are transformed into rewrite rules by relatively simple syntactic rules. A theorem of the form `(implies (and` $h_1 ... h_n$`) (equal` *lhs rhs*`))`, once proved, is interpreted as the rule "replace each instance of *lhs* by the corresponding instance of *rhs*, provided the corresponding instances of the hypotheses, h_i, can be rewritten to true." The rewriter applies rewrite rules to a term from the inside out. Thus, the term `(`f a_1 ... a_k`)` is first rewritten to `(`f a'_1 ... a'_k`)` by rewriting each argument, a_i, to a'_i, and then rules whose left-hand sides match `(`f a'_1 ... a'_k`)` are tried.

To simplify a formula, `(implies (and` $h_1 ... h_n$`)` *concl*`)`, we regard the formula as a clause, {`(not` h_1`)` ... `(not` h_n`)` *concl*} and rewrite each literal, in turn, assuming the other literals false.

The user of ACL2 must be cognizant of the interpretation of a formula as a rule when each theorem is stated. In a sense, the job of the user is to "program" the simplifier so that it carries out an appropriate simplification strategy for the problem domain at hand.

4.5.2 Formal Symbolic Computation

Formal symbolic computation is nothing more than using the symbolic definition of the machine and certain axioms and lemmas to simplify an expression like `(sm s0 (times-clock i))` to a symbolic state. A key to our approach is to use the clock as a means of driving the expansion of `sm`.

For example, since `(times-clock i)` is equal to `(+ 2 (+ (* i 4) 2))`, `(sm s0 (times-clock i))` is `(sm s0 (+ 2 (+ (* i 4) 2)))`, by substitution of equals for equals. Now we can appeal twice to the easily proved lemma

```
(defthm sm-+
  (implies (and (natp i) (natp j))
           (equal (sm s (+ i j))
                  (sm (sm s i) j))))
```

to reduce the "big" computation to three smaller ones `(sm (sm (sm s0 2) (* i 4)) 2)`. The representation of the clock expression allows the user to tell the system how to decompose the computation into its "natural" paths, in this case a 2-step prelude, followed by a `(* i 4)`-step inductive loop, and finishing with a 2-step postlude.

However, there is a problem with this: in the presence of the heavy-duty arithmetic rules necessary to carry out proofs about practical programs, the "natural" clock expression `(+ 2 (+ (* i 4) 2))` is liable to be rewritten to some algebraically equivalent but pragmatically useless form like `(* 4 (+ i 1))`. For that reason we use special *clock arithmetic operators* when we write clock expressions. For example, the actual definition of `times-clock` is

```
(defun times-clock (i)
  (cplus 2 (cplus (ctimes i 4) 2))) ,
```

where `cplus` is simply `+` (on the naturals) and `ctimes` is simply `*` (on the naturals) but the theorem prover is not generally allowed to use those facts. Nor is the theorem prover "aware" that `cplus` and `ctimes` enjoy the algebraic properties of associativity, commutativity, etc. The theorem `sm-+` shown above is actually proved with `cplus` in place of `+`.

Having taken the user's "hint" that we should see the computation as a composition of three smaller executions, `(sm (sm (sm s0 2) (ctimes i 4)) 2)`, we focus—as always—on the innermost expression. By simplifying from the inside out we are, in this case, doing a "forward execution" of the program.

It is easy to arrange for `(sm s0 2)` to simplify very quickly to `(step (step s0))` using

```
(defthm sm-opener
  (and (equal (sm s 0) s)
       (implies (natp i)
                (equal (sm s (+ 1 i))
                       (sm (step s) i))))) .
```

We do not allow `step` to enter the problem any other way. Thus, we `step` only when we have the clock's permission (and thus the user's permission) to do so.

Now the naive expansion of the `step` function produces a catastrophic case explosion because `step` (actually, its sub-function `execute`) does a case split based on all possible instructions. Therefore, another key to our approach is to expand a `step` expression only when the current instruction of the state is known. We do this in ACL2 with the lemma

```
(defthm step-opener
  (and (implies (halt s) (equal (step s) s))
        (implies (consp (current-instruction s))
                 (equal (step s)
                        (if (halt s)
                            s
                            (execute (current-instruction s) s)))))) .
```

This is merely the definition of `step`, but after proving it we "disable" the definition so that ACL2 cannot use it. The hypothesis of the second conjunct above is irrelevant; `(step s)` is equal to the `if`-expression even without it. But by proving this weaker theorem we arrange for ACL2 to expand step only when it can prove that the `current-instruction` is a `consp`. This is only a "hack" but it works.

Now what do we know about the current instruction in `s0`, the term we want to `step` first in `(step (step s0))`? We know exactly what it is: `(call times)`. So `step-opener` applies and rewrites `(step s0)` first to `(call 'times s0)` (using the fact that we know the `halt` flag is off in `s0` and the definition of `execute`), and thence to

```
(modify s0
        :pc '(times . 0)
        :stk (cons (pc+1 (pc s0)) (stk s0))) .
```

Call this state `s1`.

Now we wish to simplify `(step s1)`. All we can use is `step-opener`. Do we know the current instruction? Yes! The `pc` in `s1` is `(times . 0)`, so the current instruction is the first one in `times` and we know the code for `times`! The current instruction of `s1` is `(movi 2 0)`. Technically this deduction is trivial, but in practice, when program counters and code memory are large, it requires that the theorem prover efficiently execute such ground expressions as `(fetch '(times . 0) (list (times-program)))`.

Applying `step-opener` and simplifying produces

```
(modify s1
        :pc '(times . 1)
        :mem (put 2 0 (mem s1))) ,
```

which, in terms of `s0`, is

```
(modify s0
        :pc '(times . 1)
        :stk (cons (pc+1 (pc s0)) (stk s0))
        :mem (put 2 0 (mem s0))) .
```

Call this state `s2`.

Recall now how we got here. We simplified (sm s0 (times-clock i)) to (sm (sm (sm s0 2) (ctimes i 4)) 2) and now we have simplified the (sm s0 2) to s2 above. We would now like to step s2 (ctimes i 4) times, but how many is that?

So far, all of the lemmas cited are program-independent: they need be proved only once for the given machine and cause the theorem prover to do symbolic computation driven by the clock expression. But now we need a program-dependent lemma, namely one that tells us what the loop in times is doing. We discuss it below.

But first, suppose we had a lemma that explained the loop, e.g., that says "(sm s (ctimes i 4)) changes memory so that location 0 is 0 and location 2 contains the sum of its old value and the product of the first two memory locations." Then we are virtually done because our s2, above, has a 0 in location 2 and so by the lemma, (sm s2 (ctimes i 4)), which we will call state s3, is simply s2 with a 0 in location 0 and the desired product in location 2. Then we conclude the proof by computing (sm s3 2). This symbolically executes the jumpz at the top of the loop and the ret, since location 0 is now 0. The ret pops the stack back into the pc and we are left with a state in which the pc is (pc+1 (pc s0)), the memory has a 0 in 0 and the product in 2, and all other components are unchanged. That completes the proof of times-correct, but we assumed we had a crucial lemma about the behavior of the loop.

4.5.3 Loop Invariants

So we now turn to the crucial lemma describing what happens when the pc is (times . 1) and the clock is (ctimes i 4), i.e., (ctimes (nth 0 (mem s)) 4). The lemma we prove is generally easy for the user to "invent" and always follows the same general pattern. Because of the clock decomposition, the situation is perfect for induction because the pc points to the top of a loop, the loop decrements location 0 once each time around, the loop takes 4 instructions to traverse once, and the clock is precisely 4 times the initial value of location 0. Here is the lemma we prove.

```
(defthm times-correct-lemma
  (implies (and (statep s)
                (< 2 (len (mem s)))
                (equal i (nth 0 (mem s)))
                (natp i)
                (equal (pc s) '(times . 1))
                (equal (assoc 'times (code s)) (times-program))
                (not (halt s)))
           (equal (sm s (ctimes i 4))
                  (modify s :mem (times-fn-mem s)))))
```

The lemma tells us what the loop does to memory. It addresses itself to a completely general entrance to the loop with just enough time on the clock to finish the loop. We can paraphrase, from the top, as follows. Let `s` be a state with at least two locations. Suppose location 0, which we call `i`, contains a natural. Suppose the `pc` points to the top of the loop in our `times` program and the state is not halted. Then executing the state `(ctimes i 4)` times modifies the memory as described by `(times-fn-mem s)`. We define that function below.

Before we define `times-fn-mem` we draw attention to a subtle aspect of the statement of `times-correct-lemma`. The left-hand side of the conclusion, which is the target pattern when the lemma is used as a rewrite rule, is `(sm s (ctimes i 4))` whereas it could have been equivalently stated as `(sm s (ctimes (nth 0 (mem s)) 4))`. The preferred statement contains two variables, `s` and `i`, whereas the equivalent alternative contains only `s`. Practically speaking (at least for ACL2) the preferred statement is more general because it allows the rule to fire on `(sm` s' `(ctimes (nth 0 (mem` s`)) 4))` provided the contents of location 0 in s' is equal to that of location 0 in s. This is the most common triggering expression, in fact, because s' is generally derived from s by initializing certain locations (e.g., location 2). The equivalent alternative formulation does not unify with the common triggering expression and hence would not be used.

It only remains to say what `times-fn-mem` is. Roughly speaking, it is a recursive function that, in effect, performs the same series of writes to memory that executing the loop does.

```
(defun times-fn-mem (s)
  (let ((m0 (nth 0 (mem s)))
        (m1 (nth 1 (mem s)))
        (m2 (nth 2 (mem s))))
    (if (zp m0)
        (mem s)
        (times-fn-mem
         (modify s
                 :mem
                 (put 0 (+ m0 -1)
                  (put 2 (+ m1 m2) (mem s))))))))
```

Note that the function operates on a state, `s`, and returns a modified version of the memory of `s`. The modification is obtained by repeatedly decrementing memory location 0 and adding the contents of location 1 into 2 until location 0 is 0 (or not a natural). The function terminates because the contents of location 0 decreases.

Many of the basic ideas we have presented here for describing computation mathematically, including the idea of "derived functions" such as `times-fn-mem`, were inspired by techniques first developed by John McCarthy [15, 16] and his students.

The recursion in `times-fn-mem` suggests an induction on `s`: the base case is when location 0 contains a 0 and the induction step assumes the theorem for `s'` and proves the theorem for `s`, when location 0 is non-0 and the `s'` is obtained from `s` by decrementing location 0 and incrementing location 2 by the contents of location 1. Proving `times-correct-lemma` by this suggested induction is completely straightforward, given the previously described arrangements for symbolic computation.[2]

Of course one reason `times-correct-lemma` is easy to prove is because of what it does *not* say. It does not tell us that the loop computes the product of locations 0 and 1. It tells us that it modifies memory as described by `times-fn-mem`. We prove as a separate lemma that a product is produced.

```
(defthm times-fn-mem-is-times
  (implies (and (< 2 (len (mem s)))
                (equal m0 (nth 0 (mem s)))
                (equal m1 (nth 1 (mem s)))
                (equal m2 (nth 2 (mem s)))
                (natp m0)
                (natp m1)
                (natp m2))
           (equal (times-fn-mem s)
                  (put 0 0
                   (put 2 (+ m2 (* m0 m1)) (mem s))))))
```

This lemma says that the sequence of writes done by `times-fn-mem` is just the same as writing 0 into location 0 and writing `(+ m2 (* m0 m1))` into location 2, where `m0`, `m1` and `m2` are the contents of locations 0, 1 and 2 respectively. This lemma is the mathematical crux of the entire proof: iterated addition is multiplication. But note that the theorem makes no mention of `sm` or of the semantics of instructions.

The introduction of `times-fn-mem` is thus another key element in the methodology. Roughly put, one should introduce the recursive function that does the same sequence of state modifications as the program and then break the proof into two parts. In part one, e.g., `times-correct-lemma`, prove that the program has the claimed operational behavior. In part two, e.g., `times-fn-mem-is-times`, prove that the operational behavior satisfies the desired specification. The first part is

[2]In order to do the proof with the suggested induction, first eliminate the use of `i` by substituting `(nth 0 (mem s))` for `i`. Otherwise, one must slightly modify the induction to accommodate `i`.

complicated only by the size of the machine and the program; since this complication is often considerable, it is convenient to deal with it in isolation from the specification. The second part is complicated only by the difference between the operational behavior and the specification; since this difference can be great, it is convenient to deal with it in isolation from the machine.

4.5.4 Memory Expression Management

The discussion above omits one last class of "generic" problems that must be faced, having to do with the normalization of such memory expressions as `(put 0 x (put 2 y mem))`. That expression naturally arises if the `put` to location 2 is done first and then the `put` to location 0 is done. That is what happens in `times`. Note that we might have done them in the other order `(put 2 y (put 0 x mem))` and produced the very same memory. It is not uncommon for the program to do the `put`s in one order and for the specification to do them in another. Furthermore, one needs to be able to fetch the contents of a memory location from a modified memory, e.g., to realize that `(nth 1 (put 0 x mem))` is `(nth 1 mem)`.

If the memory locations are always small numbers, as in this example, the proof can be done by expanding `put` so that both `put`-nests above become `(list* x (cadr mem) y (cdddr mem))`. But if memory locations are large or symbolic this is not practical. We use the following rules to normalize memory expressions.

The following rule resolves memory references.

```
(defthm nth-put
  (implies (and (natp i)
                (natp j))
           (equal (nth i (put j val mem))
                  (cond ((equal i j) val)
                        (t (nth i mem))))))
```

To eliminate unnecessary `put`s we use

```
(defthm put-put-0
  (implies (and (natp i)
                (< i (len mem))
                (equal (nth i mem) val))
           (equal (put i val mem) mem))) .
```

The next rule eliminates redundant `put`s.

```
(defthm put-put-1
  (equal (put i val2 (put i val1 mem))
         (put i val2 mem)))
```

The following rule, when properly used, orders nests of `puts` so that the addresses are listed lexicographically, e.g., so that `(put 0 x (put 2 y mem))` is preferred over the other ordering.

```
(defthm put-put-2
  (implies (and (natp i)
                (natp j)
                (not (equal i j)))
           (equal (put i vali (put j valj mem))
                  (put j valj (put i vali mem)))))
```

In ACL2 we control this rule by implementing a verified metafunction, cf. [3], which recognizes and rewrites expressions that are out of order.

Finally, we must be able to determine the length of the memory after doing a write.

```
(defthm len-put
  (implies (and (natp i)
                (< i (len mem)))
           (equal (len (put i val mem)) (len mem))))
```

4.5.5 System Decomposition, Revisited

Suppose we have a program, say `expt`, that uses our `times` program and we wish to prove `expt` correct. We here discuss how the form of `times-correct` permits it to be used within the scheme just sketched. Recall the just proved

```
(defthm times-correct
  (implies (and (statep s0)
                (< 2 (len (mem s0)))
                (equal i (nth 0 (mem s0)))
                (equal j (nth 1 (mem s0)))
                (natp i)
                (natp j)
                (equal (current-instruction s0) '(call times))
                (equal (assoc 'times (code s0)) (times-program))
                (not (halt s0)))
           (equal (sm s0 (times-clock i))
                  (modify s0
                          :pc (pc+1 (pc s0))
                          :mem (put 0 0
                                 (put 2 (* i j) (mem s0))))))) .
```

Imagine we are proving a conjecture about `expt`. The program contains a `(call times)` instruction, say at `pc (expt . 5)`. The hypotheses of the conjecture will presumably contain the assumption that `times` is defined as by our `times-program` (or else `expt` is calling a different `times`). The hypotheses will include analogous assumptions about the definitions of all other subroutines used. The user is presumed to have structured the clock expression in the new conjecture so that when the `(call times)` instruction is hit in the symbolic computation, the clock will be `(times-clock` *i*`)` (for whatever *i* is in location 0 of the memory). Thus, during the symbolic computation for the proof of the `expt` conjecture the term `(sm` *s* `(times-clock` *i*`))` will arise. In the next section we illustrate an `expt` that uses `times` as a subroutine; there we will also illustrate the proper construction of a superior clock expression.

The above `times-correct` theorem is a candidate for use; indeed, it will likely be the only rewrite rule we have telling us about how to expand `sm` for `(times-clock i)` steps. We will have to relieve the hypotheses but this will generally be straightforward if `times` is being applied to two naturals. Note that `times-correct` does not require that `times` be the only program in the `code` memory; it would be unusable if it did. If the hypotheses can be relieved, the lemma replaces `(sm` *s* `(times-clock` *i*`))` by

```
(modify s
        :pc '(expt . 6)
        :mem (put 0 0
              (put 2 (* i j) (mem s))))
```

Note in particular that the `pc` is incremented to `(expt . 6)`, memory location 0 is cleared, and memory location 2 is assigned the product. The code for `times` is not entered or considered.

`Times-correct` essentially extends the abstract machine so that `(call times)` is a new primitive instruction (that takes `(times-clock` *i*`)` ticks). The above described methodology for constructing proofs uses `times-correct` exactly as needed. Furthermore, if a mistake has been made so that, for example, the clock expression for the call is not `(times-clock` *i*`)` or the preconditions for `times` are not satisfied, the symbolic computation stops at `(call times)` because no other rules apply.

4.5.6 Summary of the Methodology

Our methodology can be summarized as follows. We define the semantics of the new machine operationally. Then we prove the symbolic computation theorems and the memory expression management theorems. We also provide rules for the primitive data types on the machine, e.g., arithmetic, bit vectors, etc. All of this work is done more or less as the machine is being formalized and is independent of applications programs.

When an applications program is introduced, we specify it in the style described. We define its clock function so as to make explicit the natural decomposition of the computation. We define the recursive function that does the same series of writes to memory. We then prove two lemmas about the program (assuming it has exactly one loop). In the first, we prove that the execution of the loop in the program does the same thing as the recursive function. In the second, we prove that the recursive function satisfies the general specification. These two lemmas allow us to prove that the program meets its specification.

4.6 Another Example

We conclude with a concrete example, discussed very briefly. We define a suitable `expt` and prove that it exponentiates, following exactly the methodology outlined for `times`. The program is given by

```
(defun expt-program nil
  '(expt (move 3 0)    ; 0   mem[3] ← mem[0] (save args)
         (move 4 1)    ; 1   mem[4] ← mem[1]
         (movi 1 1)    ; 2   mem[1] ← 1      (initialize ans)
         (jumpz 4 9)   ; 3   if mem[4]=0, go to 9
         (move 0 3)    ; 4   mem[0] ← mem[3] (prepare for times)
         (call times)  ; 5   mem[2] ← mem[0] * mem[1]
         (move 1 2)    ; 6   mem[1] ← mem[2]
         (subi 4 1)    ; 7   mem[4] ← mem[4]-1
         (jump 3)      ; 8   go to 3
         (ret)))       ; 9   return.
```

The theorem we wish to prove about `expt` is shown below. Note that we require the memory to have at least five locations to insure that all the writes are legal. We also require the code memory to contain our definitions of both `expt` and of `times`, but we do not say which is first or whether others are present.

```
(defthm expt-correct
  (implies (and (statep s0)
                (< 4 (len (mem s0)))
                (equal i (nth 0 (mem s0)))
                (equal j (nth 1 (mem s0)))
                (natp i)
                (natp j)
                (equal (current-instruction s0) '(call expt))
                (equal (assoc 'expt (code s0)) (expt-program))
                (equal (assoc 'times (code s0)) (times-program))
                (not (halt s0)))
           (equal (sm s0 (expt-clock i j))
                  (modify s0
                          :pc (pc+1 (pc s0))
                          :mem
                          (if (zp j)
                              (put 1 (expt i j)
                               (put 3 i
                                (put 4 0 (mem s0))))
                              (put 0 0
                               (put 1 (expt i j)
                                (put 2 (expt i j)
                                 (put 3 i
                                  (put 4 0 (mem s0)))))))))))
```

This formula is more complicated than that for `times` because `expt` is a more complicated program, not because the complexity of `times` is “spilling over.”

The clock function for `expt` is

```
(defun expt-clock (i j)
  (cplus 4
         (cplus (ctimes j (cplus 2 (cplus (times-clock i) 3)))
                2))) .
```

Four instructions are used to get from `(call expt)` to the loop at `pc` 3. Then the loop is traversed `j` times, where `j` is the initial value of location 1. On each traversal, two instructions are used to get to `(call times)`, then `(times-clock i)` instructions are used to execute that (as previously established), and then three more instructions are used to get back to the top of the loop. Upon finishing the loop, two instructions are used to get past the `ret`.

The derived function, the recursive description of the loop (`pc`s 3–8), is defined as

```
(defun expt-fn-mem (s)
  (let ((m1 (nth 1 (mem s)))
        (m3 (nth 3 (mem s)))
        (m4 (nth 4 (mem s))))
    (if (zp m4)
        (mem s)
        (expt-fn-mem
         (modify s
                 :mem
                 (put 0 0
                  (put 1 (* m3 m1)
                  (put 2 (* m3 m1)
                   (put 4 (- m4 1) (mem s)))))))))) .
```

The first lemma we must prove is that the loop computes the derived function, `expt-fn-mem`.

```
(defthm expt-correct-lemma
  (implies (and (statep s)
                (< 4 (len (mem s)))
                (equal m3 (nth 3 (mem s)))
                (equal m4 (nth 4 (mem s)))
                (natp (nth 1 (mem s)))
                (natp m3)
                (natp m4)
                (equal (pc s) '(expt . 3))
                (equal (assoc 'expt (code s)) (expt-program))
                (equal (assoc 'times (code s)) (times-program))
                (not (halt s)))
           (equal (sm s
                      (ctimes m4
                              (cplus 2 (cplus (times-clock m3) 3))))
                  (modify s :mem (expt-fn-mem s)))))
```

The second lemma is that `expt-fn-mem` computes the exponential function (and `put`s the result and several others into certain memory locations).

```
(defthm expt-fn-mem-is-expt
  (implies (and (< 4 (len (mem s)))
                (equal m1 (nth 1 (mem s)))
                (equal m3 (nth 3 (mem s)))
                (equal m4 (nth 4 (mem s)))
                (natp m1)
                (natp m3)
                (natp m4))
           (equal (expt-fn-mem s)
                  (if (zp m4)
                      (mem s)
                      (put 0 0
                       (put 1 (* m1 (expt m3 m4))
                        (put 2 (* m1 (expt m3 m4))
                         (put 4 0 (mem s)))))))))
```

The two lemmas are sufficient, together with the symbolic computation and memory expression management lemmas, to allow ACL2 to prove that `expt` is correct.

4.7 Extensions and Advice

Our small machine illustrates a few elementary modeling techniques including state representation, the fetch-execute cycle, timing and termination, and several classes of instruction including data movement, arithmetic, branching, and subroutine call and return. Of course, practical languages and machines contain far more complexity.

The most obvious omission from our machine is any restriction on its resources. Fabricated machines have limits on all physical resources, e.g., word size, stack size, data memory size, and program memory size. We generally include these limits as components in the state and generalize the handling of the `halt` flag so that it can be used to hold an "error message." We then define the semantic functions so that when the resource limits are violated the `halt` flag is set appropriately, e.g., to a pair containing the program counter and an error string such as `"arithmetic overflow"` or `"stack overflow"`.

Other complicating aspects of practical machines include interrupts, io, and pipelining, to name a few. Such features can be accommodated within the general scheme described here. The statement and proof of program properties on such machines is more difficult than shown here. But because practical machines are complex, this difficulty is not an artifact of formalization and must be faced if one is to derive confidence in code from the proofs. The techniques used to manage this complexity in the theorem prover are similar to those illustrated here — the normalization of symbolic states, expansion of complicated functions only under strict controls, the provision of rules that work or fail quickly. Examples of these techniques are described in some of our larger-scale projects, such as [19, 24, 25, 6, 7, 8].

We offer three pieces of general advice. First, start small. Most successful projects have started with a "toy" version of the machine and refined the basic approach. For example, start with 5 instructions and add the other 195 later. To add new features, such as interrupts or a pipeline, return to the "toy" and integrate simplified versions of the new feature there. The benefit of having a "toy" version of the model is that one can experiment with new features relatively quickly; often several approaches might be tried and abandoned before a suitable one is found and then elaborated to full complexity. When experimenting with "toys" keep in mind that the proof techniques being used must not depend on the small scale.

Second, consider the introduction of intermediate levels of abstractions. The "right" model might be a hierarchy of abstract machines rather than a single abstract machine. For example, one might produce a finite resource model and an infinite resource model of the abstract machine, or a pipelined model and a simpler sequential model. The two levels of abstraction are then proved equivalent under certain conditions and this allows some program proofs to be carried out in a simpler setting. Often designers hold several different views of the machine. These different abstractions may not be recognized or given names by the design team but are extremely helpful when identified. The machine's programmers may create a new abstract machine via programming conventions (e.g., "we treat register 3 as a stack"); identifying these conventions and formalizing the appropriate abstract machine makes proofs simpler.

Finally, think carefully about how the theorem prover will use the definitions and rules you provide it. Recall for example our discussion of the difference between `(sm s (ctimes (nth 0 (mem s)) 4))` and `(sm s (ctimes i 4))`. It was not luck or brilliance that led us to state our rules the way we did; it was careful thought about how the rules would be used.

4.8 Conclusion

State-of-the-art microprocessors and the machine languages they provide can be formally modeled operationally. This operational semantics provides a simulation capability for the new machine or language, provided it is done in a computational logic, such as ACL2, which provides execution. In addition, it is possible to configure a mechanical theorem prover, such as Nqthm or ACL2, to use the semantics effectively in carrying out proofs of programs written in the new language.

We have explained how we do this in a very simple setting and can assure the reader that it scales up.

The reason one might want to do this is simple: it is the most expedient way to track an evolving machine design and verify programs written in the evolving programming language.

4.9 Acknowledgments

The techniques described here evolved over many years. We are therefore indebted to our students and colleagues, especially those who experimented with these techniques and helped refine and demonstrate them: Bill Bevier, Bishop Brock, Art Flatau, Warren Hunt, David Goldschlag, Matt Kaufmann, Ken Kunen, David Russinoff, Natarajan Shankar, Sakthikumar Subramanian, Carolyn Talcott, Matt Wilding, Bill Young, and Yuan Yu. Most of these techniques were first developed with the Nqthm theorem prover. Nqthm was supported primarily with grants and contracts from the National Science Foundation and the Office of Naval Research. The formalization and proof methodology was transferred from Nqthm to the ACL2 theorem prover, as shown here. It has been carried out on a grand scale with that system by Bishop Brock in his CAP work.

The ACL2 theorem prover was supported in part at Computational Logic, Inc., by the Defense Advanced Research Projects Agency, ARPA Order 7406, and the Office of Naval Research, Contract N00014-94-C-0193. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of Computational Logic, Inc., the Defense Advanced Research Projects Agency, the Office of Naval Research, or the U.S. Government.

References

[1] W. R. Bevier, W. A. Hunt, J S. Moore, and W. D. Young. Special Issue on System Verification. *Journal of Automated Reasoning*, 5(4):409–530, December, 1989.

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

[3] R. S. Boyer and J S. Moore. Metafunctions: Proving Them Correct and Using Them Efficiently as New Proof Procedures. In R. S. Boyer and J S. Moore, editors, *The Correctness Problem in Computer Science*, pp. 103–184, Academic Press, London, 1981.

[4] R. S. Boyer and J S. Moore. *A Computational Logic Handbook*. Academic Press, New York, 1988. The most recent release of the Nqthm prover, Nqthm-1992, may be found at either ftp://ftp.cli.com/pub/nqthm/nqthm-1992/nqthm-1992.tar.Z or ftp://ftp.cs.utexas.edu/pub-/boyer/nqthm-1992.tar.Z.

[5] R. S. Boyer and Y. Yu. Automated Proofs of Object Code for a Widely Used Microprocessor. *JACM*, 43(1):166–192, January 1996. http://www.cs.utexas.edu/users/boyer/mc-rev3.ps.Z.

[6] B. Brock. *The CAP 94 Specification*, CAP Technical Report 8, Computational Logic, Inc., 1717 W. 6th, Austin, TX 78703, July, 1995.

[7] B. Brock. *Formal Analysis of the CAP Instruction Pipeline*, CAP Technical Report 10, Computational Logic, Inc., 1717 W. 6th, Austin, TX 78703, June, 1996.

[8] B. Brock. *Formal Verification of CAP Applications*, CAP Technical Report 15, Computational Logic, Inc., 1717 W. 6th, Austin, TX 78703, June, 1996.

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

[10] J. Goldberg, W. Kautz, P.M. Melliar-Smith, M. Green, K. Levitt, R. Schwartz, and C. Weinstock. Development and Analysis of the Software Implemented Fault-Tolerance (SIFT) Computer. Technical Report NASA Contractor Report 172146, National Aeronautics and Space Administration, Langley Research Center, Hampton, Va. 23665, 1984.

[11] W. A. Hunt. FM8501: A Verified Microprocessor. Phd thesis, University of Texas at Austin, December 1985. Lecture Notes in Computer Science 795, Springer-Verlag, 1994.

[12] W. A. Hunt and B. Brock. A Formal HDL and Its Use in the FM9001 Verification. *Proceedings of the Royal Society*, Series A, Vol. 339, 1992.

[13] M. Kaufmann and J S. Moore. *ACL2: A Computational Logic for Applicative Common Lisp, The User's Manual (Version 1.8).* ftp://ftp.cli.com/pub/acl2/v1-8/acl2-sources/doc-/HTML/acl2-doc.html, 1995.

[14] M. Kaufmann and J S. Moore. ACL2: An Industrial Strength Version of Nqthm. In *Proceedings of the Eleventh Annual Conference on Computer Assurance (COMPASS-96)*, pages 23–34. IEEE Computer Society Press, June 1996.

[15] J. McCarthy. Towards a Mathematical Science of Computation. *Proceedings of IFIP Congress*, North-Holland, pages 21–28, 1962. http://www-formal.stanford.edu/jmc/towards-.html.

[16] J. McCarthy. A Basis for a Mathematical Theory of Computation. In *Computer Programming and Formal Systems*, P. Braffort and D. Hershberg, eds., North-Holland Publishing Company, 1963. http://www-formal.stanford.edu/jmc/basis.html.

[17] S. P. Miller and M. Srivas. Formal Verification of the AAMP5 Microprocessor: A Case Study in the Industrial Use of Formal Methods. In *WIFT '95: Workshop on Industrial-Strength Formal Specification Techniques*, pages 2–16, Boca Raton, FL, 1995. IEEECS.

[18] J S. Moore. Mechanically Verified Hardware Implementing an 8-Bit Parallel IO Byzantine Agreement Processor. Technical Report NASA CR-189588, NASA, 1992.

[19] J S. Moore. *Piton: A Mechanically Verified Assembly-Level Language.* Automated Reasoning Series, Kluwer Academic Publishers, 1996.

[20] J S. Moore, T. Lynch, and M. Kaufmann. A Mechanically Checked Proof of the Correctness of the AMD5K86 Floating Point Division Algorithm. Submitted, 1996. http://devil.ece.-utexas.edu:80/~lynch/divide/divide.html.

[21] S. Owre, J. Rushby, and N. Shankar. PVS: A Prototype Verification System. In D. Kapur, editor, *11th International Conference on Automated Deduction (CADE)*, pages 748–752. Lecture Notes in Artificial Intelligence, Vol. 607, Springer-Verlag, June 1992.

[22] D. Russinoff. A Mechanically Checked Proof of the Correctness of the AMD K5 Floating-Point Square Root Algorithm. 1106 W. 9th St., Austin, TX 78703, July 1996.

[23] M. Wilding. *Machine-Checked Real-Time System Verification.* PhD thesis, University of Texas, 1996. ftp://ftp.cs.utexas.edu/pub/boyer/wilding-diss.ps.gz.

[24] W. D. Young. *A Verified Code-Generator for a Subset of Gypsy.* PhD thesis, University of Texas, 1988.

[25] Y. Yu. *Automated Proofs of Object Code For a Widely Used Microprocessor.* PhD thesis, University of Texas at Austin, 1992. Lecture Notes in Computer Science, Springer-Verlag (to appear). ftp://ftp.cs.utexas.edu/pub/techreports/tr93-09.ps.Z.