Symbolic Simulation: an ACL2 Approach

J Strother Moore[1]

Department of Computer Sciences
The University of Texas at Austin
Austin, TX 78712-1188
moore@cs.utexas.edu

Abstract. Executable formal specification can allow engineers to test (or *simulate*) the specified system on concrete data before the system is implemented. This is beginning to gain acceptance and is just the formal analogue of the standard practice of building simulators in conventional programming languages such as C. A largely unexplored but potentially very useful next step is *symbolic simulation*, the "execution" of the formal specification on indeterminant data. With the right interface, this need not require much additional training of the engineers using the tool. It allows many tests to be collapsed into one. Furthermore, it familiarizes the working engineer with the abstractions and notation used in the design, thus allowing team members to speak clearly to one another. We illustrate these ideas with a formal specification of a simple computing machine in ACL2. We sketch some requirements on the interface, which we call a *symbolic spreadsheet*.

1 Introduction

The use of formal methods requires relatively high up-front costs to create a formal specification of the desired system or component. The cost of doing proofs, e.g., to relate a formal specification to an implementation or lower-level model, is even higher. Still more effort is required to construct mechanically checked proofs. These concerns are formidable barriers to the adoption of formal methods by industry.

Those of us in the formal methods community recognize that specification without proof is still valuable; we also recognize that careful mathematical argument without mechanically checked formal proof is valuable. The more carefully one records and analyzes design decisions, the more likely bugs will be found. The earlier the process starts, the earlier the bugs will be found. While valid, these arguments justify what might be called "informal formal methods," the use of conventional mathematical techniques in the service of hardware and software design, without mechanized support.

There is no doubt that such methods are effective and economical in the hands of experts – virtually all of mankind's deep mathematics has been done without mechanized support. But the promise of formal methods is to harness mathematical rigor, precision, and abstraction to help the man or woman in the cubicle. Thus, many of us have focused on tools.

We are working on making our tools "smarter" and more convenient to use. This work includes better proof procedures, better interfaces, better tutorials and introductory material, and more comprehensive libraries of previously-formalized results. With some moderate amount of training in the use of the tools, this will make it possible for lead engineers to create formal design specification documents. There is no need, in my opinion, for the average engineer to have the skills necessary to do this.

However, it is important that these formal design documents be useful to a much wider audience than those who write them. One mechanism is provided by the possibility that formal specifications can be executable. This idea is somewhat contrarian in the theorem proving and formal methods communities, which tend to favor abstraction over other attributes of a logic. But in the hardware design community, executable specifications are almost standard practice, if one regards the ubiquitous simulators, written in C and other conventional programming languages, as specifications.

These simulators play a dual role. First, they allow engineers to run the design on sample input, for example to help debug the requirements. Second, the source code may be inspected to clarify or disambiguate the informal design documents produced by the lead engineers. Formal specifications, written in executable logics such as Pure Lisp or other functional programming languages, can serve the same purposes while encouraging a somewhat more abstract specification style and providing a migration path to proof. Because evaluation and simulation are already well-understood by engineers, the use of the formal specification requires little or no training beyond the logical notation used for constants.

A more radical suggestion is that the formal specifications be used to provide a "symbolic simulation" tool. Of course, symbolic simulation (or "symbolic evaluation") is not new. Programmers have been using it since the earliest days — indeed, symbolic evaluation of a program is often easier than execution when one is limited to paper and pencil methods. Symbolic evaluation played a key role in the first version of the Boyer-Moore Pure Lisp theorem prover [15, 2] where it was called, simply, "evaluation." It was also used in the SELECT system [3], where it was combined with path assertions and counterexample generation for linear arithmetic constraints to produce an extended program testing environment.

With a symbolic simulation capability an engineer can "run" a design on certain kinds of indeterminant data, thereby covering more cases with one test. Because of its close connection with simulation, symbolic simulation is easy to grasp; indeed, it so naturally follows simulation that one may not notice its power at first.

With such tools, properly packaged, the formal specification could be inspected and analyzed by many people with knowledge of the design issues and applications. This has two beneficial effects. First, "bugs" or other undesirable features of the design are more liable to be found early. Second, it will both educate the work force and raise expectations of clarity and abstraction. In particular, engineers will learn to read the specification notation used by the lead engineers. Language influences how we think. Seeing the abstract ideas of a design rendered into syntax is helpful. Given examples, people can generalize from them and will use the notation informally to communicate – and to *reason* – about the design. Expectations are raised in the sense that engineers will come to value the clarity and abstraction of formal specifications, once the language is familiar.

Symbolic simulation tools are thus an important bridge between current practice and the more wide-spread use of formal methods. I believe that when symbolic simulation tools are widely available, industry will find in its ranks a larger-than-expected number of engineers who are able to exploit the expressive power of formal notation to produce cleaner and more reliable designs. Furthermore, I believe this clarity, combined with the re-usability of previously formalized notions, will make it possible to create new designs faster than is currently done.

Greve makes many of these same points in [9], where he discusses a symbolic simulator for the JEM1 microprocessor. His paper gives specific examples of actual design bugs found by engineers using a symbolic simulator.

In the rest of this paper, I use one particular formal logic and theorem prover to illustrate the points just made. The system I use is ACL2. "ACL2" stands for "A Computational Logic for Applicative Common Lisp" [6, 14]. It was developed by Matt Kaufmann and me as a successor to the Boyer-Moore theorem prover, Nqthm [5]. The main idea was to replace the home-grown Pure Lisp of Nqthm with applicative Common Lisp so that formal models could be executed more efficiently and on a wider variety of platforms. However, symbolic simulation as a general technique can probably be provided by virtually any theorem proving system (see, for example, [9] which uses PVS [7]) that provides automated term rewriting, because at the logical level symbolic simulation is just "simplification."

2 Formalizing Computing Machines

Consider a simple computing machine whose state is given by a program counter, a control stack of suspended program counters, a memory, a status flag, and a program ROM. How might we formalize such a machine in ACL2? The most commonly used approach is described in [4]. We only sketch the formal model here. The ACL2 script corresponding to the results of this paper is available at http://www.cs.utexas.edu/moore/publications/symsim-script/index.html.

We represent the state of the machine as a 5-tuple. The five components of the state are accessed by functions named, respectively, `pc`, `stk`, `mem`, `halt`, and `code`. Each is defined in the obvious way to retrieve the appropriate element of a linear list. New states are constructed by making a list of the five components, e.g., `(list` *pc stk mem halt code*`)`. Invariants ("guards") are maintained to insure that certain relationships hold among the components. For example, `mem` is always a list of integers. Because we generally construct new states by modifying a few fields in old states, we use a Common Lisp "macro" to write most of our states. 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`))`. Note that, despite the name, `modify` does not destructively change the state but constructs a new "copy." ACL2 is an applicative language. We omit the definition of `modify`.

Individual instructions at the ISA level are given semantics by defining functions that appropriately modify the current state of the machine. For example, `(MOVE 2 0)` is an instruction. At the level of abstraction used in this example, we represent instructions as lists, e.g., `'(MOVE 2 0)`. Informally, the `MOVE` instruction takes two addresses and moves the contents of the second into the first. We formalize the semantics of `MOVE` by defining a function that takes one additional argument, the current state of the machine. The function returns the state

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

obtained by incrementing the program counter and changing memory as described.

Once such a function is defined for every instruction, the "execute" part of the machine's "fetch-execute" cycle is defined by case analysis on the opcode of the given instruction.

```
(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)))) .
```

The "fetch-execute" step is then defined by composition, with suitable handling of the `halt` status flag.

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

Finally, the machine's basic cyclic behavior is then defined

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

as an "iterated step function". It steps the state s n times. The name "**sm**" stands for "small machine."

3 ACL2 as an Execution Engine

This model is easily programmed in applicative Common Lisp. One immediate consequence is that the model can be executed. That is, if you supply an explicit initial state and some number of instructions to execute, **sm** can be executed on any Common Lisp host to return the final state. Using evaluation you can test the system specification.

Below we show a particular program in the **sm** language. The program is named **TIMES** and it computes the product of two natural numbers by repeated addition. The comments explain how it works.

```
(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
```

If called with two naturals i and j in memory locations 0 and 1, the program leaves $i \times j$ in memory location 2 and clears location 0 (by "counting i down"). The list constant shown above will be denoted by π. It represents a typical entry in the code component of a state.

Consider the following explicit state. Call this state α.

```
(st :pc   '(TIMES . 0)
    :stk  nil
    :mem  '(7 11 3 4 5)
    :halt nil
    :code '(π))
```

The program counter, **pc**, of α is a pair containing the symbol **TIMES** and a 0, indicating that the next instruction is the 0^{th} instruction of the **TIMES** program in the code of the state. The stack component, **stk**, of α is empty. The α state has only five memory locations, containing, respectively, 7, 11, 3, 4, and 5. The **halt** flag is **nil**.

If we evaluate **(step** α**)** in ACL2 we get

```
(st :pc '(TIMES . 1)
    :stk NIL
    :mem '(7 11 0 4 5)
    :halt nil
    :code '(π)) .
```

The `MOVI` instruction at `pc` 0 of our `TIMES` program has been executed. The program counter has been incremented by one and memory location 2 has been cleared. "Single stepping" like this is often useful. Note that we could have used the expression `(sm` α `1)` to run α one step.

To run α 31 steps, evaluate the ACL2 expression `(sm` α `31)`. This produces the following state:

```
(st :pc '(TIMES . 5)
    :stk NIL
    :mem '(0 11 77 4 5)
    :halt T
    :code '(π)) .
```

The program counter points to the 5^{th} instruction of `TIMES`. Observe that location 0 has been cleared, location 1 still contains 11, and location 2 contains 77. The halt flag has been set. The code still contains the list containing π.

This is an example of simple execution. The example illustrates looping but not subroutine `CALL`. A suitable interface would make it possible for an engineer not trained in formal methods – but familiar with the informal design documents for the `sm` machine – to use the formal specification to do tests of the design. This point was illustrated in the ACL2 demonstration accompanying Dave Hardin's talk at the 1998 Computer Aided Verification conference [12], in which an ACL2 model of the JEM1 ALU was integrated into a JEM1 simulator written in C.

The `sm` example is so small that it does not illustrate an important point: ACL2's execution capability can handle much larger system designs. Indeed, on "well-typed" (i.e., "gold" definitions [14]), ACL2's execution capability is just Common Lisp. We discuss performance measures of ACL2's execution and symbolic simulation of `sm` examples later in this paper.

In [6] we discuss a project in which Bishop Brock used ACL2 to formalize the Motorola CAP digital signal processor[8]. A model similar to the one described here was used, but it was orders of magnitude more complex. From [6]:

> The CAP design follows the 'Harvard architecture', i.e., there are separate program and data memories. The design includes 252 programmer-visible data and control registers. There are six independently addressable data and parameter memories. The data memories are logically partitioned into 'source' and 'destination' memories; the sense of the memories may be switched under program control. The arithmetic unit includes four multiplier-accumulators and a 6-adder array. The CAP executes a 64-bit instruction word, which in the arithmetic units is further decoded into a 317-bit, low-level control word. The instruction set includes no-overhead looping constructs and automatic data scaling. As many as 10 different registers are involved in the determination of the next program counter. A single instruction can simultaneously modify well over 100 registers. In practice, instructions found in typical applications simultaneously modify several dozen registers. Finally, the CAP has a three-stage instruction pipeline which contains many programmer-visible pipeline hazards.

The ACL2 specification of the CAP could be used as described above for simulation. In fact, the ACL2 model executed several times faster than the compiled SPW (Signal Processing Workbench) simulator and yet accurately modeled every bit in the processor, every cycle.

More recently, ACL2's execution capability was exploited at AMD. As part of a project to verify certain floating-point designs for the AMD-K7^{TM}, Art Flatau of AMD, wrote a mechanical translator from AMD's RTL language (essentially a subset of Verilog) to ACL2. This translator was used to produce ACL2 models of the floating-point circuits to be studied. However, before investing the time to try to prove the models correct, AMD managers insisted that the translator be "vetted" against the production RTL simulator. The ACL2 and RTL models were executed on some 80 million test vectors and found to return the same results. Only after this successful test was it deemed worthwhile to try to prove the ACL2 models correct. Such corroborative evidence would have been much harder to gather had the formal models not been executable. It is noteworthy that the 80 million test vectors failed to expose errors in the designs – errors later found by proof.

ACL2 is currently being used in an experiment at Rockwell-Collins to construct an executable specification of their JEM1, the world's first silicon Java Virtual Machine[17, 11].

4 ACL2 as a Theorem-Proving Engine

Our example makes it so clear that the definition of `sm` is "just" a Lisp program that it may be more appropriate to argue that it can be used as a specification! With ACL2 we can prove the following simple theorem about the specification.

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

This theorem shows that `sm` runs compose. The theorem is proved automatically by ACL2, by an induction on i, followed by simplification of both the base case and the induction step under the axioms and definitions involved. It takes ACL2 about 12 seconds to find the proof.

The user of ACL2 can help the theorem prover by giving it hints. For example, the proof above takes so long because, in the induction step, the system unnecessarily case splits on the instruction executed by `step`. This is obvious when one looks at ACL2's output during the proof: one sees a case for each instruction opcode. The proof would take even longer if our definition of `step` defined more opcodes. But the definition of `step` is actually irrelevant to this theorem! The system does not "know" that, but the user may – or may at least intuit it. If the user gives the system the hint to "disable `step`," which means to try to find a proof without using the definition of `step`, the system succeeds in finding a proof and only takes 0.12 seconds. This is just an example of the introduction of abstraction into the proof process. The details of `step` are irrelevant.

By exploiting such knowledge the user can dramatically speed up proofs; more importantly, the user can lead ACL2 to proofs that it would not find on its own.

The most common way for the user to give hints to the system is to build in rewrite rules about newly defined concepts. The user formulates these rules as theorems for the system to prove. Once they are proved the system interprets these theorems as rules and uses them automatically during simplification. For example, the **sm-+** theorem, above, implicitly instructs the system to rewrite all expressions of the form `(sm s (+ i j))` into the form `(sm (sm s i) j)`. To be effective at extending the rule-base, the ACL2 user must understand how the system interprets previously proved theorems as rules.

The user can collect definitions, theorems and other forms of hints and advice into "books." Books can be "certified" once and then "included" into an ACL2 session. This has the effect of configuring the ACL2 simplifier (and all other proof techniques) as specified in the book. Multiple books can be included. The interaction of independently developed rules must be considered, but there are some hooks in the system to help authors codify their strategies.

It takes a lot of expertise to develop books. It is not unlike trying to teach a new class. A lot of material must be organized in ways that, when done, seem obvious; but many other, less-effective organizations are available and have to be considered. In [4] we describe such a book for `sm`. We show how to lead ACL2 to a proof of the following theorem about the `TIMES` program.

```
(defthm  times-correct
  (implies (and (statep s0)
                (< 2 (len (mem s0)))
                (equal i (get 0 (mem s0)))
                (equal j (get 1 (mem s0)))
                (<= 0 i)
                (equal (current-instruction s0) '(CALL TIMES))
                (equal (assoc-eq 'TIMES (code s0)) 'π)
                (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))))))) .
```

This theorem can be read as follows. Consider a state with a memory containing at least three items (which, by definition of `statep`, must be integers). Let i and j be the 0^{th} and 1^{st}, respectively, and suppose $0 \leq i$. Suppose the current instruction of the state points to the instruction `(CALL TIMES)` and that `TIMES` is defined by our previously exhibited π. We can paraphrase this rather long hypothesis by saying the state is poised to execute our `TIMES` on natural numbers i and j. The theorem tells us what the state will look like if we run it a certain number of steps. The number is not explicitly given, but is computed by `times-clock` as a function of i. The resulting state is a modification of the starting one obtained by incrementing the program counter by one, depositing a 0 into location 0, and depositing $i \times j$ into location 2.

It takes ACL2 less than 2 seconds to prove the theorem above. However, even with well-designed books, proving theorems like this requires a certain amount of training in how to use the book, how to approach the proof at a high level, and how to interact with ACL2. We explain some of the techniques used in [4].

What ACL2 can achieve in the hands of an expert is illustrated by David Russinoff's work in [16]. Russinoff used ACL2 to check proofs of the correctness of the AMD-K7 hardware for floating-point addition, subtraction, multiplication, division and square root. Using the translator mentioned above, Russinoff translated AMD's HDL descriptions (at the RTL level) into ACL2 functions. Russinoff then developed books containing thousands of lemmas about floating-point arithmetic. Using these books, he checked his proofs of the compliance of the hardware to the IEEE floating point standard. Bugs were found and corrected.

5 ACL2 as a Symbolic Simulator

Can the formal specification be made accessible to engineers not wishing to do formal proofs? The answer is yes: use it to drive a symbolic simulator for the design. We now illustrate that with our `sm` model.

Consider the following state:

```
(st :pc   '(TIMES . 0)
    :stk   nil
    :mem  (list i j x y z)
    :halt nil
    :code '(π))
```

This state is like α except that the five memory locations have unspecified content. We use the variables i, j, x, y and z to denote those contents and assume them to be integers. We use β to denote the state above.

Recall the `TIMES` program π:

```
(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
```

What is the result if we start a simulation on β and run for 4 steps? Assume that i and j are natural numbers and that i is positive. Then the answer is obvious: After 4 steps, location 0 contains $i{-}1$ and location 2 contains j. In addition, the program counter is `(TIMES . 4)`, i.e., the next instruction is the `JUMP` back to 1.

Here is that problem, posed as a conjecture to ACL2:

```
(implies (and (ints i j x y z)
              (< 0 i))
         (equal (sm β 4) v))
```

Here v is a simple variable symbol. Note that its only occurrence in the conjecture is as the right-hand side of the conclusion. The conjecture could not possibly be a theorem under these circumstances (unless, of course, the hypotheses are contradictory). Nevertheless, the attempt to prove it with ACL2, using the above mentioned book, reduces to the goal of proving that v is

```
(st :pc   '(TIMES . 4)
    :stk  nil
    :mem  (list (- i 1) j j y z)
    :halt nil
    :code '(π)) .
```

That is, the rules in the book configure ACL2's simplifier into a symbolic simulator for the machine code in our specification.[1]

It is clear that an interface is required so as to hide the simplification process from the user. We have not constructed such an interface. But for the purposes of this paper we imagine one. We call it a *symbolic spreadsheet*. As its name suggests, we imagine a collection of "boxes" containing data. Boxes are linked via operations, with some boxes representing input and others representing output. However, unlike conventional spreadsheets, the data is symbolic, the links connecting boxes are formally defined logical functions, and the processing done by the spreadsheet is symbolic simplification. Familiar notation ought to be used where possible (e.g., in arithmetic expressions). We imagine being able to collect boxes together into larger structures, so that one of our states can be represented as a hierarchy of boxes on the spreadsheet. Obviously, it should be possible to hide data, i.e., to display the "memory" box by the contents of locations 0 and 2 only. Furthermore, it should be possible to have multiple states on the screen at once, so one can compare different states.

Such a spreadsheet should permit a rather simple configuration in which the user fills in a "form" to describe an initial symbolic state, such as β, and sees the result of stepping that state in another such form.

In our view, the difficulty is not so much the interface as the simplification. We argue in this paper that the ACL2 simplifier can be configured to do this job. We therefore continue to present our results as formal ACL2 terms, rather than as displayed in our imagined spreadsheet.

What if we start in β and run 4 steps, then 1 more (getting back to the top of the loop) and then 3 more? Then we should see i decremented twice and we should see the sum of two j's in location 2. Of course, this happens only if we know that i exceeds 1. Indeed, the proof attempt produces the goal to prove the unknown v equal to

[1] It is not necessary to phrase the problem as a bogus theorem-proving challenge. It is possible to invoke the ACL2 simplifier directly.

```
(st :pc   '(TIMES . 4)
    :stk  nil
    :mem  (list (- i 2) j (+ j j) y z)
    :halt nil
    :code '(π))
```

This illustrates another requirement on the spreadsheet. We need an "assumptions" box which contains assumptions about the variables. This can be menu-driven to limit the assumptions to those supported by the underlying rules.

What happens if we forget to say that *i* exceeds 1? That is, suppose we just have that *i* is positive? The result is a two-way case split. In one case, we have the additional hypothesis that `(- i 1)` exceeds 0 and the goal state shown above. In the other we have the additional hypothesis that `(- i 1)` is 0 and the goal state

```
(st :pc   '(TIMES . 5)
    :stk  nil
    :mem  (list 0 j j y z)
    :halt t
    :code '(π))
```

in which the program has halted.

In our imagined spreadsheet, the execution of this branching symbolic computation results in two copies of the output state being displayed, each with its own assumptions box. The two states might be "stacked", a visual arrangement that would immediately alert the user to the fact that the computation branched.

6 Extensibility of the Symbolic Simulator

So far we have used the symbolic simulator only to run primitive instructions. It is worthwhile to point out that it is extensible. Of course, it requires an "expert" to extend it because extension is done by adding new theorems to the database driving ACL2. But suppose that someone proves `times-correct` as stated above.

The symbolic simulator can then run calls of the `TIMES` code. For example, a run of length `(+ (times-clock i) 2)` starting in the symbolic state

```
(st :pc   '(MAIN . 0)
    :stk  nil
    :mem  (list i j x y z)
    :halt nil
    :code '(π
           (MAIN (CALL TIMES)
                 (ADD 4 2)
                 (SUBI 4 1))))
```

produces the state

```
(st :pc   '(MAIN . 3)
    :stk  nil
    :mem  (list 0 j (* i j) y (+ (* i j) z -1))
    :halt nil
    :code '(π
           (MAIN (CALL TIMES)
                 (ADD 4 2)
                 (SUBI 4 1)))) .
```

Is this correct? The **MAIN** program calls **TIMES**, multiplying i times j and leaving the result in location 2. Then the **MAIN** program adds location 2 into location 4 and subtracts 1. The final value of location 4 ought to be $(i \times j) + z - 1$. So the simulator produced the expected results, regardless of the values of the variables. Note also that the simulator run shows that location 0 is cleared by this code sequence and that locations 1 and 3 are unchanged.

Actually, for this example to work the expression specifying the length of the run should be **(cplus (times-clock** *i***) 2)**. As noted in [4], it is convenient to maintain an isolation between arithmetic expressions denoting run-lengths and other expressions, so the former can be used by user to control proof decomposition. The interface to our symbolic simulator could mitigate this somewhat by translating arithmetic operators in the "run length" box to their "clock operator" counterparts. But the user would still have to understand how to formulate "clock expressions" so as to decompose the execution. For example, the equivalent expression **(cplus 2 (times-clock** *i***))** would have a very different effect on the simulator.

7 Performance

How fast is ACL2's symbolic simulation? That is, how fast is the ACL2 rewriter? To put it in perspective, we start by measuring the performance of ACL2 evaluation. All of our measurements were conducted on the small machine model **sm** and carried out on a 200 MHz Sun Microsystems Ultra 2 with 512 MB of memory, running ACL2 Version 2.2 built on Gnu Common Lisp.

ACL2 is applicative Common Lisp, provided the Common Lisp primitives are only applied in their intended domains. For example, the Common Lisp function **car** is intended to be applied to conses and to **nil** and the Common Lisp function **+** is intended to be applied to numbers. Common Lisp implementations are not required to check at runtime whether their arguments are suitable; that is the user's responsibility. Implementations are thus efficient but not "safe."

ACL2 functions, on the other hand, are axiomatized to be total. In our axioms, **car** returns **nil** if applied outside its intended domain and **+** treats non-numeric arguments as though they were **0**. This notion of intended domain is formalized in ACL2 by the use of *guards*, arbitrary ACL2 formulas that specify the intended relationships between the input variables. By proving certain mechanically generated *guard conjectures*, ACL2 can guarantee that a given ACL2 function is Common Lisp compliant or "gold," which means that its execution on arguments satisfying its guard is "safe." See [14] for details.

If an ACL2 function is known to be Common Lisp compliant, it can be evaluated (on arguments satisfying its guard) via direct Common Lisp execution. In practice this means we execute binary code compiled from the function definition. If, on the other hand, an ACL2 function is not known to be compliant, or the actual arguments do not satisfy the guard, evaluation is performed by a purpose-built ACL2 interpreter that completes the Common Lisp primitives in accordance with the axioms. In practice, this means we run binary code compiled from a translation of the function definition in which function symbols have been mapped to completed counterparts which do runtime guard checks. When we talk of the execution speed of ACL2 functions we must specify whether we mean the speed of "possibly uncompliant" code or "compliant" code. Here we provide measures of both.

The definition of `sm` is Common Lisp compliant provided the guard on `(sm` s n`)` requires s to be a "well-formed state" and n to be a natural number. We do not exhibit the guards in this paper but they are given in the previously mentioned script available on the web. Since guards are optional, it is possible to strip them out to obtain "possibly non-compliant" code.

How much work is it to provide guards and prove compliance? Supplying guards for all of the functions in the `sm` system requires defining six predicates used nowhere but in guards (i.e., the notions of syntactically well-formed program counters, stacks, memories, instructions, programs, and systems of programs), as well as supplying a guard for each function in the `sm` system. In addition, about twenty additional lemmas have to be proved in order to lead ACL2 to the proof that `sm` and all of its subroutines are Common Lisp compliant. It took me several hours to invent appropriate guards.[2] Without guards, the small machine system can be admitted (syntax checking plus termination proofs) in less than a second. Verifying the guards requires about 5 seconds of additional proof.

How fast can ACL2 execute `sm`? We used the following expression:

```
(sm (st :pc   '(MAIN . 0)
        :stk  nil
        :mem  (list 0 0 0 0 0)
        :halt nil
        :code (list π
                    '(MAIN (MOVI 0 10000)
                           (MOVI 1 1000)
                           (CALL TIMES)
                           (RET))))
    40007)
```

[2] The guards on a function must imply the guards on all the subfunctions used in its definition, including recursive calls. In general this may be as hard as finding inductive invariants, but in practice it is not difficult. The difficulty in choosing guards is more stylistic: should one endeaver merely to insure that the Common Lisp primitives are used properly or should one strengthen the guard formulas so that they capture the correctness specification?

This requires **sm** to execute 40,007 instructions to multiply 10,000 times 1000 (by 10,000 repeated additions) leaving 10 million in memory address 2. The number 40,007 is just `(+ 2 (times-clock i) 1)`, where i is 10,000.

If **sm** is regarded as non-compliant, the computation takes 7.39 seconds. If **sm** is regarded as compliant, it takes 0.53 seconds. This illustrates the value of guard verification if execution speed is of importance.

Since **sm** is an instruction interpreter, it is convenient to translate this performance into small machine instructions per second. Non-compliant execution proceeds at 40,007/7.39 or 5,414 small machine instructions per second in this example. Compliant execution proceeds at about 75,000 small machine instructions per second in this example. Because **sm** represents memory as a linear list of values, the speed degrades as memory size increases. Our particular experiment uses a very small memory and all the writes target the first three locations, reducing "copying" time. In more realistic tests of a comparable ACL2 model, Greve, Hardin and Wilding in [10] measured simulation speeds of about 19,000 instructions per second. A model written in C of the same processor provided 2.47 million instructions per second. The authors of [10] describe modifications to ACL2 that allowed them to achieve speeds of 1.85 million instructions per second. While some of the techniques used in [10] impose a burden on the user to insure fidelity with the axioms, I highly recommend the paper to those wishing to use ACL2 to simulate formal processor models.

Now we consider symbolic simulation. Here there is no difference between compliant and non-compliant models: the computation is done by ACL2's rewrite engine. The expression we have chosen to symbolically simulate is

```
(sm (st :pc   '(MAIN . 0)
        :stk  nil
        :mem  (list 1000 j x y z)
        :halt nil
        :code (list π
                    '(MAIN (CALL TIMES) (RET))))
    (+ (* (+ 1000 1) 4) 1))
```

in a context in which the variables are assumed integral. Obviously, this simplifies to

```
(st :pc   '(MAIN . 1)
    :stk  nil
    :mem  (list 0 j (* 1000 j) y z)
    :halt T
    :code (list π
                '(MAIN (CALL TIMES) (RET))))
```

and requires the symbolic simulation of 4,005 small machine steps.

It takes ACL2 about 55 seconds to symbolically simulate this expression. This translates to about 72 symbolic instructions per second.

Since symbolic simulation is just simplification, an arbitrary amount of search (through the lemma data base) might be involved in a given symbolic simulation.

The simplication of the expression above produces, in addition to the final state shown, a list of all the rules used. If one poses the original symbolic simulation problem again, and this time gives ACL2 the hint to use only the rules listed, the time required drops to about 21 seconds.

Remarkably, if one discounts the time required to track the rules being used the time drops to about 46 seconds (with no hint) and to about 17 seconds (with the hint). The latter performance translates to 235 symbolic instructions per second.

The performance of a symbolic simulation engine is very dependent upon the data base of rules available. In addition, as the rule tracking observation above illustrates, "extraneous" aspects of a theorem prover may affect performance. We might therefore ask how many rewrites are involved in this symbolic simulation.

First, what is a "rewrite?" (a) Is it a call of a program in the simplifier which might replace a term with another term? (b) Is it the attempt to apply a conditional rewrite rule? (c) Is it the successful application of such a rule? Or (d) is it the application of such a rule on a path that actually leads to the final result? Interpretation (a) would let us count as a rewrite any call of ACL2's rewriter or type facility on a term, since any such call might return a changed term. Interpretation (b) excludes the use of built-in rules, such as the reduction of `(equal` *x x*`)` to `t`. The difference between (b) and (c) has to do with whether a rule is just "tried" (meaning we try to match the left-hand side of the rule and then try to relieve the hypotheses, etc.) or "used" (meaning the try was successful and the right-hand side of the rule was substituted for the target). Finally (d) brings to light the fact that the majority of rewrites usually happen on non-productive branches, i.e., branches in the proof search that do not lead to success and which are ultimately abandoned. The use of the hint, above, prunes out many (but not all) of these unsuccessful branches.

How many "rewrites" are involved in the symbolic evaluation of the term above? First consider the symbolic simulation without any hint. Approximately 1,000,000 calls of rewriting routines occur. Approximately 900,000 rules are tried and 425,000 are applied. However, only about 150,000 rule applications are actually involved in the final result. If we provide the hint, only about 400,000 calls occur. Approximately 160,000 rules are tried and virtually all of them are actually applied and used in the final result. These statistics are somewhat rough but give an idea of the amount of symbolic manipulation work involved in symbolic simulation.

It is useful also to map this to the number of rewrites per symbolic instruction simulated. Recall that 4,005 instructions are simulated in this experiment. So, without the hint, we try about 225 rules per instruction, actually apply about 100 and actually need about 40. In this example, the hint limits the search almost perfectly.

It should be noted that "long" symbolic simulation runs such as this one are generally impossible to do in the presence of indeterminant branching, since the answer state then grows exponentially. We timed a long run to amortize the cost through the general theorem prover entrance and to demonstrate that in appropriate contexts ACL2 can do such runs.

8 Conclusion

Our conclusions were drawn in the introduction, which might be appropriately re-read now. A symbolic simulator could be of great use to a design team, in part because it is accessible to many more people on the team than a verification tool would be. Furthermore, it leads naturally to verification and so represents a technology driver.

We have illustrated how such a simulator might be constructed with ACL2. The simple nature of the problem we tackled here may make some readers think this is an unrealistic proposal for designs of industrial scale. However, ACL2 has been used successfully to handle very large problems. Indeed, in the Motorola CAP work [6], Brock used ACL2 in exactly the fashion described producing states that sometimes required several megabytes of text to print fully. ACL2's simplifier is up to the task. One of Brock's problems was how to glean information from such large symbolic states. The spreadsheet would help render such states surveyable.

Two subtasks remain. The first is to construct an interface that invites the engineer to use it. This could become a lucrative product if successful. ACL2 is in the public domain and represents the heart of the tool.

The second subtask is to construct the books necessary to configure the simulator for a particular industrial model. The place to start is with current "heavy duty" ACL2 users who are proving theorems about large systems. It is likely that their existing books already contain the bulk of the required rules, simply because their books have been designed to do proofs and so often codify simplification of symbolic states.

9 Acknowledgments

These ideas have been kicking around in the Nqthm and ACL2 user communities for many years. A symbolic simulation capability is basically a first step in any ACL2 project aimed at code proofs and our ideas for how to harness the simplifier in this capacity have been developed by many, especially those who participated in the CLI "short stack" work[1] and its extensions, including Bill Bevier, Bob Boyer, Bishop Brock, Art Flatau, Warren Hunt, Matt Kaufmann, Matt Wilding, and Bill Young. I am especially indebted to Warren Hunt and Bishop Brock for the current view of the symbolic spreadsheet. I am also indebted to Dave Greve, Dave Hardin and Matt Wilding for their work in integrating ACL2 models into the JEM1 processor design process.

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), 1989.

2. R. S. Boyer and J S. Moore, Proving Theorems about Pure LISP Fucntions, *JACM*, **22**(1), pp. 129–144, 1975.
3. R. S. Boyer, K. N. Levitt and B. Elspas, SELECT–A Formal System for Testing and Debugging Programs, *Proceedings of the International Conference on Reliable Software*, IEEE Catalogue Number 75CHO940-7CSR, pp. 234–245, 1975.
4. R. S. Boyer and J S. Moore. Mechanized Formal Reasoning about Programs and Computing Machines. In R. Veroff (ed.), *Automated Reasoning and Its Applications: Essays in Honor of Larry Wos,* MIT Press, 1996.
5. R. S. Boyer and J S. Moore. *A Computational Logic Handbook, Second Edition.* Academic Press, London, 1997.
6. B. Brock, M. Kaufmann, and J S. Moore. ACL2 Theorems about Commercial Microprocessors. In *Proceedings of Formal Methods in Computer-Aided Design (FMCAD'96)*, M. Srivas and A. Camilleri (eds.), Springer-Verlag, November, 1996, pp. 275–293.
7. J. Crow, S. Owre, J. Rushby, N. Shankar, and M. Srivas. A Tutorial Introduction to PVS, presented at *Workshop on Industrial-Strength Formal Specification Techniques*, Boca Raton, FL, April 1995 (see http://www.csl.sri.com/pvs.html).
8. S. Gilfeather, J. Gehman, and C. Harrison. Architecture of a Complex Arithmetic Processor for Communication Signal Processing in *SPIE Proceedings, International Symposium on Optics, Imaging, and Instrumentation*, **2296** *Advanced Signal Processing: Algorithms, Architectures, and Implementations V*, July, 1994, pp. 624–625.
9. D. A. Greve, Symbolic Simulation of the JEM1 Microprocessor, Technical Report, Advanced Technology Center, Rockwell Collins Avionics and Communications, Cedar Rapids, IA 52498, April, 1998 (also appearing in this volume, *The Proceedings of FMCAD '98.*
10. D. A. Greve, D. S. Hardin and M. M. Wilding, Efficient Simulation Using a Simple Formal Processor Model, Technical Report, Advanced Technology Center, Rockwell Collins Avionics and Communications, Cedar Rapids, IA 52498, April, 1998.
11. D. A. Greve and M. M. Wilding Stack-based Java a back-to-future step, Electronic Engineering Times, Jan. 12, 1998, pp. 92.
12. D. S. Hardin, M. M. Wilding, and D. A. Greve, Transforming the Theorem Prover into a Digital Design Tool: From Concept Car to Off-Road Vehicle, in A. J. Hu and M. Y. Vardi (eds.) *Computed Aided Verification: 10th International Conference, CAV '98*, Springer-Verlag LNCS 1427, pp. 39–44, 1998.
13. M. Kaufmann. ACL2 Support for Verification Projects. In *15th International Conference on Automated Deduction (CADE)* (to appear, 1998).
14. M. Kaufmann and J S. Moore. An Industrial Strength Theorem Prover for a Logic Based on Common Lisp. In *IEEE Transactions on Software Engineering* **23**(4), April, 1997, pp. 203–213.
15. J S. Moore, *Computational Logic: Structure Sharing and Proof of Program Properties*, Ph. D. dissertation, University of Edinburgh, Scotland, 1973.
16. D. M. Russinoff. A Mechanically Checked Proof of IEEE Compliance of the Floating Point Multiplication, Division, and Square Root Algorithms of the AMD-K7^{TM} Processor URL `http://www.onr.com/user/russ/david/k7-div-sqrt.html`.
17. A. Wolfe. First Java-specific MPU Rolls Electronic Engineering Times, Sept 22, 1997, pp. 1.

This article was processed using the LaTeX macro package with LLNCS style