An Executable Formal Java Virtual Machine Thread Model

J Strother Moore*
George Porter†

October 25, 2000

Abstract

We discuss an axiomatic description of a simple abstract machine similar to the Java Virtual Machine (JVM). Our model supports classes, with fields and bytecoded methods, and a representative sampling of JVM bytecodes for basic operations for both data and control. The `GETFIELD` and `PUTFIELD` instructions accurately model inheritance, as does the `INVOKEVIRTUAL` instruction. Our model supports multiple threads, synchronized methods, and monitors. Our current model is inadequate or inaccurate in many respects (e.g., we do not formalize the JVM's finite arithmetic nor do we describe class loading and initialization). But the model is a useful tool for studying the application of formal reasoning to the JVM and to Java programs.

We demonstrate two useful aspects of an operational formal semantics. First, the model is executable: bytecoded methods can be run on the model. Second, the model allows us to prove theorems about those methods or, more generally, about the model. Because the JVM provides a relatively clean semantics for Java, our model can be thought of as a step towards Java software verification. We illustrate these points. We cite some theorems proved about our model, including a theorem involving unbounded multi-threading and mutual exclusion with `MONITORENTER` and `MONITOREXIT`. Our proofs are carried out with the ACL2 theorem prover.

Keywords: parallel, distributed computation, mutual exclusion, operational semantics, verification, JVM, bytecode verification

1 Formal Executable Models

"Formal Methods" is the name given to the computer science research area devoted to the use of formal mathematical logic to model and analyze the properties of computing systems. One advantage of modeling a system formally is that proofs about it can be checked by mechanical proof checkers. This increases the odds that the proofs are flawless. Automated mechanical theorem provers can be used to help discover proofs, which can significantly reduce the tedium of constructing formal proofs.

This is not pie-in-the-sky formal methods proposal boilerplate. It is happening. At Advanced Micro Devices, Inc., formal models of the hardware designs for the floating-point FDIV instruction on the AMD Athlon™ have been mechanically proved to be compliant with the IEEE-754 standard. Indeed, all of the elementary floating-point operations on the Athlon (including addition, subtraction, multiplication, division, and square root) have been so proved. Important security properties of the IBM 4758 secure co-processor were mechanically verified at IBM Yorktown. The correctness of an auditor that checks the output of a compiler for safety-critical trainborn real-time control software for Union Switch and Signal was mechanically proved.

*Department of Computer Sciences, University of Texas at Austin, Austin, TX 78712
†Department of Computer Sciences, University of Texas at Austin, Austin, TX 78712

A bit- and cycle-accurate model of a Motorola digital signal processor was mechanically proved to conform to a higher-level sequential view in which the pipeline was abstracted away – provided the microcode being executed was free of a well-defined set of hazards. Several microcoded DSP algorithms extracted from the ROM of that microprocessor were mechanically proved correct. These applications, and others, are described in [12]. All were modeled and verified using one theorem proving system, ACL2.

ACL2 [13] stands for "A Computational Logic for Applicative Common Lisp." It is a functional programming language, a first-order mathematical logic, and a mechanical theorem prover. ACL2 was written by Matt Kaufmann and J Strother Moore (an author of this paper) and is the successor of the Boyer-Moore theorem prover Nqthm [3, 5].

As a programming language, ACL2 is a version of Common Lisp. It provides the familiar Lisp data objects, including numbers, strings, symbols and lists, along with if-then-else and function application, including recursion. ACL2 is axiomatically described. For example, it is an axiom that `(IF x y z)` is `z` if `x` is `NIL`, and is `y` otherwise. Another axiom is that `(car (cons x y))` is `x`. Theorems about ACL2 functions can be proved in this logic, most often by case analysis, simplification, and mathematical induction. A mechanical tool has been built to help the human user construct proofs. This interactive computer program combines term rewriting, decision procedures and a wide variety of heuristic techniques to provide a symbolic manipulation system. The system has sophisticated automatic search strategies for finding certain kinds of proofs and those strategies can be informed and guided by advice from the user, most often in the form of key lemmas suggested by the user and proved by the system. For details, see [13] and the ACL2 Web site `http://www.cs.utexas.edu/users/moore/acl2`. In this paper we avoid ACL2 syntax and knowledge of ACL2 insofar as possible.

The techniques for modeling microprocessors and programming languages in such a logic have been developed over a long period of time in the Boyer-Moore community. A tour de force of the method is presented in the so-called CLI Stack (produced by Computational Logic, Inc.) [1, 8, 18] which is a hierarchy of verified components including a microprocessor, loader, linker, assembler, two compilers, an operating system and some applications programs, all quite simple but also actually fabricated and practical, and all of which have been formally specified and mechanically proved correct. Another example is the work of Yuan Yu, in which the Motorola 68020 microprocessor is formalized. Yu's work is sufficiently accurate that it is possible to compile 21 of the 22 programs in the Berkeley C String Library, using `gcc -o`, and run the resulting binaries on the formal model, computing the expected results. Furthermore, Yu formally specified what these 21 programs were supposed to do and used the Boyer-Moore theorem prover to prove mechanically that the binaries met the specifications [6]. For an introduction to the modeling and proof methods used in these projects, see [4]. We merely hint at the techniques as we briefly describe our model of the JVM.

Of particular historical importance to the present work is Rich Cohen's ACL2 model of a single-threaded JVM [7]. The so-called "defensive JVM" is an accurate and complete model of a subset of the JVM instruction set. As such, the machine is more complicated than the one discussed here, but does not support threads. The defensive JVM checks the dynamic conditions required to insure type safety and is an essential step toward the specification and verification of the Java bytecode verifier. Both Cohen's model and ours are based largely on the Sun Microsystems documentation for Java and the JVM [14, 9], informed by private conversations with experts and experience with Java and the JVM.

Also of special interest is the fact that the JEM1 microprocessor, the world's first silicon JVM, built by Rockwell Collins, was modeled formally with ACL2 [19, 11]. Some proofs were done with the model but its primary use was as a simulator. The ACL2 model executes at about 90% of the speed of a carefully-written C simulator for the same model. The issues involved in the efficient execution of ACL2 models are discussed in the article by Greve, Wilding, and Hardin (Chapter 8) of [12].

There is a large body of academic work on Java modeling but relatively little that is truly formal and still less that is supported by mechanized tools. A wonderful exception is the work by Nelson, Leino and others at

Compaq Systems Research Center on the "Extended Static Checker" for Java, which is formal, practical and mechanized. See `http://research.compaq.com/SRC/esc/`. The work of Borger and Schulte [2] on Java exceptions is quite formal and accurate, but not supported by mechanized proofs. Mechanically checked proofs about simple Java programs have been constructed with several theorem provers, including HOL, Isabelle, and PVS. See, for example, [17]. However, we are unaware of mechanically checked proofs (other than those reported here) of Java classes that use multi-threading. Our work is distinguished primarily by being cast in a formally defined operational (and executable) semantics. Because we formalize the semantics we can prove theorems about the model, not just about particular Java methods or classes. We know of no mechanically checked proofs (our included) of correctness properties of significant Java applications; the field is still in its infancy.

2 Specification of the JVM

In ACL2, machines are formalized by adopting an explicit representation of the states and then writing an interpreter for the machine language. Another way of putting it is this: to formalize a machine language, implement a simulator for it in functional Lisp. While this may seem to be a mere programming exercise, it is also a logic exercise if the simulator is written in an axiomatically described programming language like ACL2.

In our model of the JVM, a state consists of three components: the thread table, the heap, and the class table. We describe each in turn. When we use the word "table" here we generally mean a list of pairs in which "keys" (which might be thought of as constituting the left-hand column of the table) are paired with "values" (the right-hand column of the table). Such a table is a map from the keys to the corresponding values.

The thread table maps thread numbers to threads. Each thread consists of three components: a call stack, a flag indicating whether the thread is scheduled, and the heap address of an object of class `Thread` in the heap uniquely associated with this thread. We discuss the heap below.

The call stack is a list of frames treated as a stack (the first element of the list is the topmost frame). Each frame contains five components: a program counter and the bytecoded method body, a table associating variable names with values, a stack, and a synchronization flag indicating whether the method currently executing is synchronized. Unlike the JVM, the local variables of a method are referenced by symbolic names rather than positions.

The heap is a table associating heap addresses with instance objects. An instance object is a table. The keys of an instance object are the successive classes in the superclass chain of the object. The value of each such key is another table, mapping the immediate field names of the class to their values. The structure of heap addresses is unimportant but they can be distinguished from integers and other data types. In our model a heap address is a list of the form `(REF` *i*`)`, where *i* is a natural number. One point where our model differs from the JVM is that in our model `(NEW "`*class*`")` is completely responsible for the object's instantiation; all fields are initialized to `0`. Classes in our model do not have separate constructors.

Finally, the class table is a table mapping class names to class descriptions. A class description contains a list of its superclass names, a list of its immediate fields, and a list of its methods. We do not model syntactic typing in our machine, though we could. Thus, our list of fields is just a simple list of field names (strings) rather than, say, a table mapping field names to signatures. A method is a list containing a method name, the names of the formal parameters of the method, a synchronization status flag, and a list of bytecoded instructions. Our model omits signatures and the access modes of methods.

Bytecoded instructions are represented abstractly as lists consisting of a symbolic opcode name followed by zero or more operands. For example, `(LOAD X)` is the instruction that pushes the value of local variable `X` onto the stack in the current frame. `(ADD)` pops two items off the stack in the current frame and pushes their sum. (IFEQ 12) pops an item off the stack and if it is 0, increments the program counter by 12; otherwise it increments it by 1. The similarity of these instructions to certain JVM instructions should be obvious, as should be the differences: we ignore the different types of LOAD (e.g., ILOAD, DLOAD, etc.) and ADD instructions, we ignore the finite range of integer data, and we count program counter offsets in number of instructions rather than number of bytes. These and most of the other discrepancies between the current model and the JVM are matters of detail that would not change the basic structure of the model to fix and do not impact our ability to use the model to study proof techniques.

For those readers curious to see how we define the semantics of such operations in ACL2, here is our definition of the function execute-PUSH which we use to give semantics to the PUSH instruction. The instruction (PUSH 3) is comparable to ICONST_3 or BIPUSH 3 on the JVM.

```
(defun execute-PUSH (inst s th)
  (make-state
   (modify-tt th
              (push (make-frame (+ 1 (pc (top-frame s th)))
                                (locals (top-frame s th))
                                (push (arg1 inst)
                                      (stack (top-frame s th)))
                                (program (top-frame s th))
                                (sync-flg (top-frame s th)))
                    (pop (call-stack s th)))
              'SCHEDULED
              (thread-table s))
   (heap s)
   (class-table s)))
```

The function takes three arguments, named inst, s, and th. The first is the list expression denoting the instruction. The first element of inst will always be the symbol PUSH and the second is the constant that is to be pushed on the stack of the current frame. The second argument of execute-PUSH, s, is the JVM state, consisting of a thread table, a heap and a class table. The third argument, th, is the number of the thread that is to be "stepped." Execute-PUSH returns the state obtained by executing the PUSH instruction in the given thread of s. It creates that state with the function make-state, which takes three arguments: the thread table, the heap and the class table of the state to be returned. The last two components of the new state above are the same as those in s. The thread table is modified by replacing the entry for th by another entry. That entry's call stack is obtained by replacing the topmost frame of the current call stack (notice we push a frame onto a stack obtained by popping one off). In the new frame, the program counter is advanced by 1, the locals remain unchanged, the constant (extracted from inst using the function arg1) is pushed on the stack, and the method program and synchronization flag are unchanged.

The most complicated instruction formalized in our model is INVOKEVIRTUAL. An example INVOKEVIRTUAL instruction on our machine is represented by the list structure (INVOKEVIRTUAL "ColoredPoint" "move" 2). Note that in place of the JVM's signature we provide only the number of parameters, since we consistently ignore type issues in this model. We paraphrase the definition of execute-INVOKEVIRTUAL by describing the state it creates from an instruction of the form below, a state s, and a thread number th.

> (INVOKEVIRTUAL c *name* n): Let *ref* be the item n deep in the stack. This is expected to be a heap reference to an instance object, *obj*. Let *class* be the class of this object (the first key in the table, i.e., the name of the most specific class in the object's class hierarchy). Use the function lookup-method to determine from the class-table of s the closest method with name *name* in *class* or its superclass chain. Let *formals* and *body* be the formal parameters and bytecoded body of the closest method. Let *formals'* be *formals* with the new symbol THIS added to the front.

Create a new call stack, cs', from the call stack of thread th in s by replacing the topmost frame by a new frame in which the program counter has been incremented by one and $n+1$ items have been popped off the stack. Create another call stack, cs'', by pushing a new frame onto cs'. This new frame should have a program counter of 0 and an empty stack. The locals of the new frame should bind $formals'$ to the topmost $n+1$ items removed from the stack in s (above), the deepest of which is bound to `THIS`. The bytecoded body of the frame should be *body*. We will use cs' and cs'' in various cases below and we will not be interested in cs'' unless the closest method is non-native. Consider the following cases.

- The closest method is native: We support only two native methods, `"start"` and `"stop"` from the `"Object"` class. We describe only the first here. In this case, *obj* should include the class `"Thread"` in its superclass chain. The new state constructed by the `"start"` method has the same heap and class table as s. The thread table is changed in two ways. First, the call stack of th is replaced by cs' above (stepping over the `INVOKEVIRTUAL`). Second, the thread th' uniquely associated with *obj* is changed so that its scheduled flag is `SCHEDULED`.
- The closest method is a synchronized method: Fetch the contents of the `"monitor"` and `"mcount"` fields in the `"Object"` class of *obj*. If the mcount is 0 or the mcount is non-0 but the monitor is th, then we say *obj* is "available" to th. If *obj* is available to th, then the new state is obtained from s by replacing the call stack with cs'' after setting the `sync-flg` component of the top frame to `LOCKED`, and by replacing the heap of s with a heap in which the `"mcount"` field of the object at *ref* has been incremented by one and the `"monitor"` field has been set to th. If, on the other hand, *obj* is unavailable, then the "new" state is s itself. Thus, the thread hangs at the `INVOKEVIRTUAL` instruction until *obj* becomes available. We do not specify the scheduler; instead, our model allows all possible interleavings of thread executions and some thread states (as the one just described) make no change if stepped before progress is possible.
- Otherwise, the new state is obtained from s by replacing the call stack with cs'' after setting the `sync-flg` component of the top frame to `UNLOCKED`.

Given `execute-PUSH`, the reader can presumably imagine how this description is coded in ACL2.

We formalize a variety of instructions in this style, including `POP`, `LOAD`, `STORE`, `ADD`, `MUL`, `GOTO`, `IFEQ`, `IFGT`, `RETURN`, `XRETURN`, `NEW`, `GETFIELD`, `PUTFIELD`, `MONITORENTER`, and `MONITOREXIT`. For each such opcode *op* we define an ACL2 function `execute-`*op* that takes the instruction, current state, and thread number and returns the next state.

We then define `step` to be the function that takes a state and a thread number and executes the next instruction in the given thread, provided that thread exists and is `SCHEDULED`. `Step` is essentially a "big switch" on the opcode of the instruction indicated by the program counter and method body in the top frame of the call stack of the given thread.

Finally we define `run` to take a "schedule" and a state and return the result of stepping the state according to the given schedule. A schedule is just a list of numbers, indicating which thread is to be stepped next. That is, our model puts no constraints on the JVM thread scheduler; however stepping a non-existent, `UNSCHEDULED`, or otherwise blocked thread is a no-op. We find it convenient also to define `(runn n schedule s)` to run the first `n` steps of `schedule` starting in state `s`.

The complete ACL2 source text for our machine is available at `http://www.cs.utexas.edu/users/-moore/publications/m4/index.html`.

Our model omits many features of the JVM. Among the more glaring omissions are accurate support for the JVM primitive data types like ints, doubles, arrays, etc., support for syntactic typing both in the naming convention in the instruction set (e.g., `IADD` versus `DADD`) and field and method signatures, class loading and initialization, INVOKESTATIC (with the concomitant requirement that classes have representative instance objects in the heap upon which synchronization can be arranged), exception handling, and errors. Experience with other commercial microprocessor models leads us to believe that these features could be added to our model without fundamentally changing its basic structure. There is no doubt that they greatly complicate the model and would complicate proofs about programs that use the features in question. That is one of the reasons we left them out. Our model is adequate however as a vehicle for studying basic mechanized proof techniques for dealing with Java programs, including multi-threaded applications.

3 Some Examples of Execution

Because our model, `run`, is an ACL2 program, it can be executed on concrete data to produce concrete results. To run our bytecode we create a state, say s_0, containing the thread table, heap, and class table we have in mind. An expression constructing such a state is shown in Figure 1. The class `Alpha` contains a single instance method `fact` which is just a recursive factorial program written in our bytecode.[1] The bytecode is similar to that produced by compiling

```
public int fact(int n){
 if (n<=0) return 1;
   else return n*fact(n-1);}
```

except our arithmetic is not bounded. We show the JVM bytecode for this `fact` in Figure 1, in line-by-line correspondence with ours. The main program of the only thread in the thread table of s_0 creates a new instance object of class `Alpha` and invokes its `fact` method after pushing `5`. That is, it calls `fact` on `5`.

Call the state in Figure 1 s_0. To run s_0 we must provide a schedule. Since the main program is in thread `0` (the only thread) and creates no other threads, a suitable schedule is just a list of `0`'s.

The list constant shown in Figure 2 is the state created by evaluating `(run (repeat 58 0)` s_0`)`. Call that state s_{58}. Thus, s_{58} is the result of stepping thread 0 fifty-eight times starting with s_0. The code elided away is just the definition of our `fact` method. Inspection of thread `0` reveals that the program counter of the top frame is pointing to the last instruction, the `(XRETURN)`, and that `120` is on the top of the stack. Stepping once would pop the top frame and push the `120` onto the empty stack of the frame below. The new top frame is poised to execute the `(HALT)` instruction. So stepping s_0 sixty times halts the machine with `120` on top of the stack in the main frame. Since `120` is 5!, the `fact` method seems to have worked.

Also evident in Figure 2 is the representation of the instance object at heap address `0`, an object of class `Alpha` (which has no fields) with superclass `"Object"` (which has fields `"monitor"`, `"mcount"` and `"wait-set"`). Note also the class table, which, in addition to our `Alpha` class, contains two built in classes, `Object` and `Thread`. In our model, the `Object` class has only the three fields listed and two (native) methods, `"start"` and `"stop"` whose semantics are built into INVOKEVIRTUAL.

There are of course many schedules that run thread `0` in s_0 for sixty instructions. Any schedule containing sixty `0`'s would work, no matter how many other thread numbers are interspersed between them.

Because our model is expressed in a formal mathematical logic, it is possible to reason about it formally, using the ACL2 mechanical theorem prover. Rather than just test that the `fact` method works for `5` and a few other numbers, we can prove a theorem stating that the `fact` method computes the factorial of its argument.

[1] We have not modeled INVOKESTATIC in this machine, so we have chosen to make `fact` an instance method.

```
(make-state
 (make-tt                               ; Thread table
  (push                                 ;  call stack of thread 0
   (make-frame 0                        ;   frame:  pc
               nil                      ;           locals
               nil                      ;           stack
               '((NEW "Alpha")          ;           method body
                 (STORE OBJ)
                 (LOAD OBJ)
                 (PUSH 5)
                 (INVOKEVIRTUAL "Alpha" "fact" 1)
                 (HALT))
               'UNLOCKED)                           sync status
   nil))
 nil                                    ; Heap
 (make-class-def                        ; Class Table
  (list
   (make-class-decl
    "Alpha"                             ;  class name
    '("Object")                         ;  superclasses
    NIL                                 ;  fields
    '(("fact" (N) NIL                   ;  Method int fact(int)
       (LOAD N)                         ;   0 iload_1
       (IFGT 3)                         ;   1 ifgt 6
       (PUSH 1)                         ;   4 iconst_1
       (XRETURN)                        ;   5 ireturn
       (LOAD N)                         ;   6 iload_1
       (LOAD THIS)                      ;   7 aload_0
       (LOAD N)                         ;   8 iload_1
       (PUSH 1)                         ;   9 iconst_1
       (SUB)                            ;  10 isub
       (INVOKEVIRTUAL "Alpha" "fact" 1) ;  11 invokevirtual #8 <Method int fact(int)>
       (MUL)                            ;  14 imul
       (XRETURN)))))))                  ;  15 ireturn
```

Figure 1: A State for Computing Factorial

```
(((0                            ; Thread 0
   ((11                         ;  call stack, top frame: pc
     ((THIS . (REF 0)) (N . 5)) ;                         locals
     (120)                      ;                         stack (120 on top)
     ((LOAD N)                  ;                         method body
      (IFGT 3)
      (PUSH 1)
      (XRETURN)
      (LOAD N)
      (LOAD THIS)
      (LOAD N)
      (PUSH 1)
      (SUB)
      (INVOKEVIRTUAL "Alpha" "fact" 1)
      (MUL)
      (XRETURN))                ;                          (pc points here)
     UNLOCKED)                  ;                          sync status
    (5                          ;  next frame:             pc
     ((OBJ . (REF 0)))          ;                          locals
     ()                         ;                          stack (empty)
     ((NEW "Alpha")             ;                          method body
      (STORE OBJ)
      (LOAD OBJ)
      (PUSH 5)
      (INVOKEVIRTUAL "Alpha" "fact" 1)
      (HALT))                   ;                          (pc points here)
     UNLOCKED))                 ;                          sync status
   SCHEDULED NIL))
 ((0                            ; Heap address 0:
   ("Alpha")                    ;  Alpha fields:  none
   ("Object"                    ;  Object fields:
    ("monitor" . 0)
    ("mcount" . 0)
    ("wait-set" . 0))))
                                ; Class table
 (("Object"                     ;  Object class
   ()                           ;   superclasses
   ("monitor"                   ;   fields
    "mcount"
    "wait-set")
   (("start" () NIL NIL)        ;   methods
    ("stop"  () NIL NIL)))
  ("Thread"                     ;  Thread class
   ("Object")                   ;   superclasses
   ()                           ;   fields (none)
   (("run" () NIL (RETURN))))   ;   methods
  ("Alpha"                      ;  Alpha class
   ("Object")                   ;   superclasses
   ()                           ;   fields (none)
   (("fact" (N) NIL (LOAD N) ... (XRETURN)))))) ; methods
```

Figure 2: The Result of Stepping the Factorial State 58 Times

Theorem. `fact` *is correct.*

```
(implies (and (poised-to-invoke-fact s th n)
              (natp n))
         (equal (top
                 (stack
                  (top-frame
                   (run (fact-sched n th) s)
                   th)))
                (factorial n)))
```

The hypotheses of the theorem assume that `s` is a state poised (in thread `th`) to invoke our `fact` method on the natural number `n`. Because `fact` is an instance method, this requires inspecting the top two objects on the stack to make sure that the topmost is a natural number and that the resolution of the name `"fact"` in the class of the next item is our `fact` method. The conclusion is an equality stating that a certain expression is equal to `(factorial n)`. Here, `factorial` is the mathematical function of that name, defined in ACL2. The expression in question describes the top item on the stack in the top frame of thread `th` in the state obtained by executing state `s` a certain number of steps, as given by `(fact-sched n th)`. Thus, this theorem establishes that by running thread `th` a certain number of steps, it computes the factorial function.[2]

The proof of the factorial theorem can be constructed interactively with the ACL2 theorem prover and the theorem prover is entirely responsible for the correctness of the proof. In this case, the user provides an inductive argument and the machine carries out that argument, expanding definitions, applying axioms and basic theorems about the machine. For a discussion of such theorems see [15]. The proof that `fact` computes factorial takes about 30 seconds (on a 700 MHz machine).

In Figure 4 we show a more interesting state, modeled after the Java `Apprentice` code shown in Figure 3. In this state the main program creates an object of class `Container` and then loops forever creating and starting `Thread` objects of class `Job`. Each `Job` is in an infinite loop using the method `incr` to read, increment, and write into the `counter` field of the `Container` object. The critical section of the `incr` method is protected by `MONITORENTER` and `MONITOREXIT`.[3]

If we remove the `MONITORENTER` and `MONITOREXIT` (and the corresponding `LOAD`) instructions from the bytecode (i.e., remove the synchronization from the Java method) we can exhibit a schedule that makes the counter decrease: run the main thread until it has started two jobs, then run the first thread until it pushes the value of the counter (which at this point will be 0) onto its local stack, then run the other thread many cycles to increment the counter several times, and finally run the first job again so that it increments its 0 and writes a 1 into the counter field.

The ability to deal with schedules and states abstractly makes it easier to explore such issues. This illustrates the value of an executable abstract model.

However, there is no schedule that makes the state in Figure 4 decrease the counter. This cannot be demonstrated by testing. It can, however, be proved by analyzing our model. Here is a theorem proved with the ACL2 theorem prover about the state shown in Figure 4, here called `*a0*`.

Theorem. `Apprentice` *Monotonicity*

```
(implies (and (natp n)
              (natp m)
              (<= n m))
         (<= (counter (runn n any-schedule *a0*))
             (counter (runn m any-schedule *a0*))))
```

[2] Our machine has unbounded integer arithmetic. We could, of course, model Java's bounded arithmetic. The factorial theorem would have to be restated to reflect that.

[3] Our byte code for `"run"` exploits the fact that `"incr"` returns "`this`" and is slightly different than the compiled Java.

```
class Container {
    public int counter;
}
class Job extends Thread {
    Container objref;
    Object x;
    public Job incr () {
        synchronized(objref) {
            objref.counter = objref.counter + 1;
        }
        return this;
    }
    public void setref(Container o) {
        objref = o;
    }
    public void run() {
        for (;;) {
            incr();
        }
    }
}
class Apprentice {
    public static void main(String[] args){
        Container container = new Container();
        for (;;) {
        Job job = new Job();
        job.setref(container);
        job.start();
        }
    }
}
```

Figure 3: The Apprentice Class: Unbounded Parallelism

```
(make-state
 (make-tt                        ; Thread Table
  (push (make-frame 0            ;  call stack, top frame: pc
                    '((CONTAINER . NIL) (JOB . NIL))    ; locals (uninitialized)
                    ()                                  ; stack (empty)
                    '((NEW "Container")                 ; main method
                      (STORE CONTAINER)
                      (NEW "Job")
                      (STORE JOB)
                      (LOAD JOB)
                      (LOAD CONTAINER)
                      (INVOKEVIRTUAL "Job" "setref" 1)
                      (LOAD JOB)
                      (INVOKEVIRTUAL "Job" "start" 0)
                      (GOTO -7))
                    'UNLOCKED)
        nil))
 nil                             ; Heap
 (make-class-def                 ; Class Table
  (list (make-class-decl "Container"        ;   Container class
                         '("Object")        ;    superclasses
                         '("counter")       ;    fields
                         '())               ;    methods (none)
        (make-class-decl "Job"              ;   Job class
                         '("Thread" "Object");    superclasses
                         '("x" "objref")     ;    fields
                         '(("incr" () nil    ;    methods
                            (LOAD THIS)
                            (GETFIELD "Job" "objref")
                            (STORE TEMP)
                            (LOAD TEMP)
                            (MONITORENTER)
                            (LOAD THIS)
                            (GETFIELD "Job" "objref")
                            (LOAD THIS)
                            (GETFIELD "Job" "objref")
                            (GETFIELD "Container" "counter")
                            (PUSH 1)
                            (ADD)
                            (PUTFIELD "Container" "counter")
                            (LOAD TEMP)
                            (MONITOREXIT)
                            (LOAD THIS)
                            (XRETURN))
                           ("run" (N) nil
                            (LOAD THIS)
                            (INVOKEVIRTUAL "Job" "incr" 0)
                            (GOTO -1))
                           ("setref" (R) nil
                            (LOAD THIS)
                            (LOAD R)
                            (PUTFIELD "Job" "objref")
                            (RETURN))
                           )))))
```

Figure 4: Apprentice in Our Model

The theorem compares the values of the counter in two states, one obtained by running `*a0*` `n` steps and the other obtained by running `m` steps, both according to the same completely unconstrained schedule. If $n \leq m$, the counter in the former state is less than or equal to that in the latter state. This theorem is a statement about an unbounded number of parallel threads using the JVM synchronization primitives. The proof requires careful (and rather global) analysis of what is happening in the heap. (For example, all threads writing to the `Container` respect the monitor and no thread changes the `objref` field of a running thread.) See, for example, Praxis 56 in [10], where Haggar writes "Do not reassign the object reference of a locked object." For details of our proof see [16].

4 Conclusion

This paper is a first step at developing an executable abstract formal model of threading in the JVM. We have explained how such a model can be built, we have shown that the model can be executed on concrete data to test the behavior of methods and threads under various scheduling regimes, and we have illustrated that it is possible to prove theorems about all possible behaviors. These proofs can be checked mechanically by ACL2, a general-purpose theorem proving engine. This engine was not designed with JVM proofs in mind; indeed, all the engine "knows" about the model is its formal definition. Our proofs of the theorems cited were straightforward applications of general techniques understood well by the ACL2 community. Further investigation of such theorems would undoubtedly lead to the codification of JVM-specific proof techniques and formal metatheorems, such as that syntactically non-interfering threads can be analyzed separately.

The prover is sufficiently powerful to be of use in verifying microprocessor architectures and floating-point implementations. While our JVM model is currently quite simple compared to the implemented JVM, past experience with ACL2 supports the hope that ACL2 will be capable of handling significantly more realistic models of the JVM.

If our past experience is any guide, the formalization of proof techniques for a realistic model of the JVM will make it easier to reason formally about the JVM and Java. In addition, such an undertaking will most likely expose oversights or ambiguities in existing informal understandings of how to write correct and reliable Java programs.

5 Acknowledgments

Our machine, which we call M4 because it is the fourth machine in a series that approaches the JVM in complexity, owes much of its basic structure to Rich Cohen who used ACL2 to formalize a single-threaded version of the "defensive JVM" [7]. We are grateful to Rich for his pioneering effort into the JVM formalization, as well as to the entire ACL2 and Boyer-Moore community for their development of techniques to formalize and reason about such machines. We are also grateful to David Hardin and Pete Manolios, who have each made many valuable suggestions in the course of this work.

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, 1989.

[2] E. Borger and W. Schulte. Defining the java virtual machine as platform for provably correct java compilation. In *Proceedings of MFCS'98*, volume LNCS 1450, pages 17–35. Springer, 1998.

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

[4] R. S. Boyer and J S. Moore. Mechanized formal reasoning about programs and computing machines. In R. Veroff, editor, *Automated Reasoning and Its Applications: Essays in Honor of Larry Wos*, pages 147–176. MIT Press, 1996.

[5] R. S. Boyer and J S. Moore. *A Computational Logic Handbook, Second Edition.* Academic Press, New York, 1997.

[6] Robert S. Boyer and Yuan Yu. Automated proofs of object code for a widely used microprocessor. *Journal of the ACM*, 43(1):166–192, January 1996.

[7] R. M. Cohen. The defensive Java Virtual Machine specification, version 0.53. Technical report, Electronic Data Systems Corp, Austin Technical Services Center, 98 San Jacinto Blvd, Suite 500, Austin, TX 78701, 1997.

[8] A. D. Flatau. A verified implementation of an applicative language with dynamic storage allocation. Phd thesis, University of Texas at Austin, 1992.

[9] J. Gosling, B. Joy, and G. Steele. *The Java Language Specification.* Addison-Wesley, 1996.

[10] P. Haggar. *Practical Java Programming Language Guide.* Addison-Wesley, 2000.

[11] David Hardin, Matthew Wilding, and David Greve. Transforming the theorem prover into a digital design tool: From concept car to off-road vehicle. In Alan J. Hu and Moshe Y. Vardi, editors, *Computer-Aided Verification – CAV '98*, volume 1427 of *Lecture Notes in Computer Science.* Springer-Verlag, 1998. See URL `http://pobox.com/users/hokie/docs/concept.ps`.

[12] M. Kaufmann, P. Manolios, and J S. Moore, editors. *Computer-Aided Reasoning: ACL2 Case Studies.* Kluwer Academic Press, 2000.

[13] M. Kaufmann, P. Manolios, and J S. Moore. *Computer-Aided Reasoning: An Approach.* Kluwer Academic Press, 2000.

[14] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification (Second Edition).* Addison-Wesley, 1999.

[15] J S. Moore. Proving theorems about Java-like byte code. In E.-R. Olderog and B. Steffen, editors, *Correct System Design – Recent Insights and Advances*, pages 139–162. LNCS 1710, 1999.

[16] J S. Moore and G. Porter. Mechanized reasoning about Java threads via a JVM thread model. *(submitted for publication)*, 2000.

[17] J. van den Berg, M. Huisman, B. Jacobs, and E. Poll. A type-theoretic memory model for verification of sequential java programs. In *14th International Workshop on Algebraic Development Techniques (WADT'99)*, volume LNCS 1827, page (to appear). Springer, 2000.

[18] Matthew Wilding. A mechanically verified application for a mechanically verified environment. In Costas Courcoubetis, editor, *Computer-Aided Verification – CAV '93*, volume 697 of *Lecture Notes in Computer Science.* Springer-Verlag, 1993. See URL ftp://ftp.cs.utexas.edu/pub/boyer/nqthm/wilding-cav93.ps.

[19] Matthew Wilding, David Greve, and David Hardin. Efficient simulation of formal processor models. *Formal Methods in System Design*, to appear. Draft TR available as `http://pobox.com/users/-hokie/docs/efm.ps`.