The Apprentice Challenge

J STROTHER MOORE
University of Texas at Austin

and
GEORGE PORTER
University of Texas at Austin

We describe a mechanically checked proof of a property of a small system of Java programs involving an unbounded number of threads and synchronization via monitors. We adopt the output of the `javac` compiler as the semantics and verify the system at the bytecode level under an operational semantics for the JVM. We assume a sequentially consistent memory model and atomicity at the bytecode level. Our operational semantics is expressed in ACL2, a Lisp-based logic of recursive functions. Our proofs are checked with the ACL2 theorem prover. The proof involves reasoning about arithmetic, infinite loops, the creation and modification of instance objects in the heap, including threads, the inheritance of fields from superclasses, pointer chasing and smashing, the invocation of instance methods (and the concomitant dynamic method resolution), use of the `start` method on thread objects, the use of monitors to attain synchronization between threads, and consideration of all possible interleavings (at the bytecode level) over an unbounded number of threads. Readers familiar with monitor-based proofs of mutual exclusion will recognize our proof as fairly classical. The novelty here comes from (i) the complexity of the individual operations on the machine, (ii) the dependencies between Java threads, heap objects, and synchronization, (iii) bytecode-level interleaving, (iv) the unbounded number of threads, and (v) and proof engineering permitting "automatic" mechanical verification. We discuss these issues. The problem posed here is also put forth as a benchmark against which to measure other approaches to formally proving properties of multi-threaded Java programs.

Categories and Subject Descriptors: D.2.4 [**Software Engineering**]: Software/Program Verification; D.3.0 [**Programming Languages**]: General; F.4.0 [**Mathematical Logic and Formal Languages**]: General

General Terms: languages,verification

Additional Key Words and Phrases: Java, Java Virtual Machine, parallel and distributed computation, mutual exclusion, operational semantics, theorem proving

---

Author's address: J S. Moore, Department of Computer Sciences, University of Texas at Austin, Austin, TX 78712.
Author's address: G. Porter, Department of Electrical Engineering and Computer Sciences, University of California at Berkeley, Berkeley, CA 94720. When this work was done, the author was at the Department of Computer Sciences, University of Texas at Austin, Austin, TX 78712.

1. THE APPRENTICE SYSTEM IN JAVA

In this paper we will study the Java classes shown in Figure 1. Here, the `main` method in the `Apprentice` class builds an instance of a `Container` object and then begins creating and starting new threads of class `Job`, each of which finds the `Container` object in its `objref` field. The `run` method for the class `Job` is an infinite loop invoking the `incr` method on the job. The `incr` method obtains a lock on the `objref` of the job and then reads, increments, and writes the contents of its `counter` field. This is a simple example of unbounded parallelism implemented with Java threads. The name "Apprentice" is both allusion to the "Sorcerer's Apprentice" (because, under a fair schedule, more threads are continually being created) and a reminder that this is a beginner's exercise.

We will prove that under all possible scheduling regimes, the value of the counter in our model "increases weakly monotonically" in a sense that allows for Java's `int` arithmetic. More precisely, let c be the value of the `counter` in any state reachable from the initial state of the `Apprentice` class. Let c' be the value in any immediate successor state. Then one of the following is true: c is undefined, c' is c, or c' is the result of incrementing c by 1 in 32-bit twos-complement arithmetic. The first disjunct is only true if the main thread of the system has not been stepped sufficiently to create the `Container`. A simple corollary of the claim above is that once the counter is defined, it "stays defined."

We make three basic assumptions. First, the semantics of Java is given by the Java Virtual Machine (JVM) bytecode [Lindholm and Yellin 1999] generated by Sun Microsystem's `javac` compiler. Second, our formal operational model of JVM bytecode is accurate, at least for the opcodes in this example. Third, the JVM provides a sequentially consistent memory model, at least for "correctly synchronized" programs. That is, any execution of such a JVM program must be equivalent to some interleaved bytecode execution. The JVM memory model, which is described in Chapter 17 of [Lindholm and Yellin 1999], does not require this and probably will not require it for arbitrary programs. The memory model is under revision [Manson and Pugh 2001]. For details see `http://www.jcp.org/jsr/detail/133.jsp`.

Many readers may think the monotonicity claim is so trivial as not to deserve proof. We therefore start by demonstrating the contrary.

Readers unfamiliar with multi-threaded programming may not see why synchronization is necessary. After all, the only line of code writing to the counter is the assignment statement `objref.counter = objref.counter + 1;`. Such code cannot make the counter decrease even without synchronization. Right? Wrong. Imagine that two `Job`s have been started (with no synchronization block in the `incr` method). Suppose the first reads the value of `objref.counter`, obtains a `0`, increments it to `1` in the local memory of the thread and is then suspended before writing to the global `counter`. Suppose then that the second `Job` is run for many cycles and increments the counter to some large integer. Finally, suppose the scheduler suspends the second `Job` and runs the first again. That `Job` will write a `1` from its local memory to the global counter, decreasing the value that was already there. Hence, the synchronization is necessary.

Given the synchronization block, one might be tempted to argue that our theorem is trivial from a syntactic analysis of the `incr` method. After all, it locks out

```
class Container {
    public int counter;
}
class Job extends Thread {
    Container objref;
    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();
        }
    }
}
```

Fig. 1. The Apprentice Example in Java

all accesses to `objref` during its critical section. This argument, if taken literally, is specious because Java imposes no requirement on other threads to respect locks. At the very least we must amend the syntactic argument to include a scan of every line of code in the system to confirm that every write to the `counter` field of a `Container` is synchronized. This however, is inadequate. Below we describe a "slight" modification of the `main` method of the `Apprentice` class. This modification preserves the system-wide fact that the only write to the `counter` field of any `Container` is the one in the synchronized block of the `incr` method. But under some thread interleavings it is still possible for the counter to decrease.

To see how to do this, consider the fact that `objref` is a field of the self object, not a local variable. The synchronization block in the `incr` method is equivalent to the following under Java semantics:

```
synchronized(this.objref) {
    (this.objref).counter = (this.objref).counter + 1;
}
```

Thus, it is possible to synchronize on one `Container` (the value of `this.objref` at the time the block is entered) and then write to the `counter` field of another `Container`, if some line of code in the system changes the `objref` field of the self object of some thread, at just the right moment. (Such code would violate Praxis 56 in [Haggar 2000], where Haggar writes "Do not reassign the object reference of a locked object.")

Imagine therefore a different `main`. This one creates the `Container` and then creates and starts two `Jobs`, say job_1 and job_2, as above. Then it creates a second `Container` object, momentarily sets the `objref` field of job_1 to that second container and then sets it back to the first `Container`. Thereafter, `main` can terminate, spin, or create more `Jobs`. Everything else in this revised system is the same; in particular, the only write to the `counter` of any `Container` is from within the synchronized critical section of `incr`, the entire `Job` class is unchanged, and, except for `main`, every thread is running a `Job`. This modified system is only a few instructions different from the one shown. But the counter can decrease in it.

Here is a schedule that causes the counter in the first `Container` to decrease. Schedule `main` to create the two `Jobs` and to set the `objref` of job_1 to the second `Container`. Next, schedule job_1 so that it obtains a lock on the second `Container` and enters its critical section. Then schedule `main` again so that it resets the `objref` of job_1 to the first container. Schedule job_1 again so that it fetches the `0` in the `counter` field of the first container and increments it, but suspend job_1 before it writes the `1` back. At this point, job_1 is holding a lock on the second `Container` but is inside its critical section prepared to write a `1` to the `counter` of first `Container`. Now schedule job_2 to run many cycles, to increment the `counter` of the first `Container`. This is possible because job_1 is holding the lock on the other `Container`. Finally, schedule job_1 to perform its write. The `counter` of the first `Container` decreases even though the `Job` class is exactly the same as shown in Figure 1.

How can we ensure that no thread changes the `objref` field of the `Job` holding the lock on the `Container`? If we could ensure syntactically that the system contains no write to any `objref` of a `Job`, we would be safe. But the Apprentice system necessarily contains such a write in the `setref` method.

We hope this discussion makes it clear that it is nontrivial to establish that the code in Figure 1 increments the counter monotonically.

2. SEMANTIC MODEL

Mechanically checked proofs of program properties require the construction or adoption of a mechanical theorem prover of some sort. This also entails the choice of a mathematical logic in which the programming language semantics is formalized. We use the ACL2 logic and its theorem prover [Kaufmann et al. 2000b; 2000a]. The ACL2 logic is a general-purpose essentially quantifier-free first-order logic of recursive functions based on a functional subset of Common Lisp [Steele 1990]. ACL2 is the successor to the Boyer-Moore theorem prover Nqthm [Boyer and Moore 1997]. Formulas in this logic look like Lisp expressions.

Rather than formalize the semantics of Java we formalize that of the Java Virtual Machine [Lindholm and Yellin 1999].

We model the JVM operationally. That is, we adopt an explicit representation of JVM states and we write, in ACL2's Lisp subset, an interpreter for the JVM bytecode. One may view the model as a Lisp simulator for a subset of the JVM. Our model includes 138 byte codes. Our model ignores certain key aspects of the

JVM, including class loading and initialization and exception handling. We call our model M5, because it is the fifth machine in a sequence designed to teach formal modeling of the JVM to undergraduates at the University of Texas.

An M5 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 Lisp "association list," 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 the object 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 six components: a program counter (pc) and the bytecoded method body, a list positionally associating local variables with values, an operand stack, a synchronization flag indicating whether the method currently executing is synchronized, and the name of the class in the class table containing this method. Double word data types are supported.

The heap is a table associating heap addresses with instance objects. An instance object is a table whose keys are the successive classes in the superclass chain of the object and whose values are themselves tables mapping field names to values. A heap address is a list of the form `(REF` *i*`)`, where *i* is a natural number.

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 instance fields, a list of its static fields, its constant pool, a list of its methods, and the heap address of an object in the heap that represents the class. We do not model syntactic typing on M5. 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 flag, and a list of bytecoded instructions. M5 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. Here are three examples, `(IADD)`, `(NEW "Job")`, and `(PUTFIELD "Container" "counter")`. The first has no operands, the second has one, and the third has two. Corresponding to each opcode is a function in ACL2 that gives semantics to the opcode.

Here is our definition of the ACL2 function `execute-IADD` which we use to give semantics to the M5 `IADD` instruction. We call `execute-IADD` the *semantic function* for `IADD`. We paraphrase the definition below.

```
(defun execute-IADD (inst th s)
  (modify th s
          :pc (+ (inst-length inst) (pc (top-frame th s)))
          :stack (push (int-fix
                         (+ (top (pop (stack (top-frame th s))))
                            (top (stack (top-frame th s)))))
```

```
                              (pop (pop (stack (top-frame th s)))))))
```

Our function takes three arguments, named `inst`, `th`, and `s`. The first is the `IADD` instruction to be executed.[1] The second is a thread number. The third is a state. `Execute-IADD` returns the "next" state, produced by executing `inst` in thread `th` of state `s`. The `modify` expression above is the body of the semantic function. It constructs a new state by "modifying" certain components of `s`. Logically speaking, the function does not alter `s` but instead copies `s` with certain components changed. In `execute-IADD` the components changed are the program counter and the stack of the topmost frame of the call stack in thread `th` of state `s`. The program counter is incremented by the length (measured in bytes) of the `IADD` instruction. Two items are popped off the stack and their "sum" is pushed in their place. The two items are assumed to be Java 32-bit `int`s. Their "sum" is computed by adding the integers together and then converting the result to the corresponding integer in 32-bit twos-complement representation.

Of special relevance to Apprentice is the support for synchronization. Every object in the heap inherits from `java.lang.Object` the fields `monitor` and `mcount`. Roughly speaking, the former indicates which thread "owns" a lock on the object and the latter is the number of times the object has been locked. Our model supports reentrant locks but they are not used here. The `MONITORENTER` bytecode, when executed on behalf of some thread `th` on some object, tests whether the `mcount` of the object is `0`. If so, it sets the `monitor` of the object to `th`, sets the `mcount` to 1, and proceeds to the next instruction. We say that `th` then *owns the lock* on the object. If `MONITORENTER` finds that the `mcount` is non-`0` and the `monitor` is not `th`, it "blocks," which in our model means it is a no-op. Execution of that thread will not proceed until the thread can own the lock. `MONITOREXIT` unlocks the object appropriately.

We have formalized 138 bytecode instructions, following [Lindholm and Yellin 1999] as faithfully as we could with the exceptions noted below. For each such opcode *op* we define an ACL2 semantic function `execute-`*op*.

We define `step` to be the function that takes a thread number and a state and executes the next instruction in the given thread, provided that thread exists and is `SCHEDULED`.

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. Our model puts no *a priori* constraints on the JVM thread scheduler. Stepping a non-existent, `UNSCHEDULED`, or blocked thread is a no-op. By restricting the values of `sched` in the expression `(run sched s)` we can address ourselves to particular scheduling regimes.

```
(defun run (sched s)
  (if (endp sched)
      s
    (run (cdr sched) (step (car sched) s))))
```

[1]By convention, whenever `execute-IADD` is applied, the opcode of its `inst` will be `IADD` and the remaining elements of `inst` will be the operands of the instruction. In the case of `IADD` there are no other operands, so `inst` will be the constant `(IADD)`, but for many other opcodes, `inst` provides necessary operands.

Lisp programmers will recognize our `run` as a simulator for the machine we have in mind. However, unlike conventional simulators, ours is written in a functional (side-effect free) style. The complete ACL2 source text for our machine is available at `http://www.cs.utexas.edu/users/moore/publications/m5/m5.lisp`. For some additional discussion this style of formalizing the JVM, see [Moore and Porter 2001]

M5 omits support for syntactic typing, field and method access modes, class loading and initialization, exception handling, and errors. In addition, our semantics for threading is interleaved bytecode operations (and thus assumes sequential consistency).

3. THE APPRENTICE SYSTEM IN BYTECODE

Recall the Apprentice system given in Figure 1. Using the Sun Java compiler, `javac`, we converted Figure 1 to `class` files and then, using a tool written by the authors and called `jvm2acl2`, we converted those `class` files to an initial state for our JVM model. We define `*a0*` to be this state. We exhibit and discuss `*a0*` below. As with all our states, the value of `*a0*` is a triple consisting of a thread table, a heap, and a class table.

The initial thread table, shown in Figure 2, contains just one thread, numbered `0`. The call stack of the thread contains just one frame, poised to execute the bytecode for the `main` method of the `Apprentice` class. As the Apprentice system runs, more threads will be created, by the execution of the `(NEW "Job")` instruction at offset `11` below. That instruction allocates a new heap object of class `Job` and also constructs a new unscheduled thread because the class `Job` extends the class `Thread`. The newly created thread will become `SCHEDULED` when the `start` method (at `main` 25) is invoked on the `Job`.

The initial heap, shown in Figure 3, contains eight instance objects. Each represents one of the classes involved in this example (or a primitive class supported by our machine). Each object is of class `java.lang.Class`, from which it gets a `<name>` field, and each extends `java.lang.Object`, from which it gets the `monitor`, `mcount`, and `wait-set` fields. We omit most of the fields after the first object, since they all have the same structure. These fields of `Class` objects are used by synchronized static methods. We discuss synchronization below. The object at heap location `0` represents the `java.lang.Object` class itself. The heap address referring to this object is `(REF 0)`. As the `main` method executes, new objects in the heap will be created. A new `Container` is built by the execution of the `NEW` instruction `main 0`, and new `Job`s are built thereafter as the `main` program cycles through the infinite loop, `main 11`–`28`.

The initial class table contains eight class declarations. The first five are for the built-in classes `java.lang.Object`, `ARRAY`, `java.lang.Thread`, `java.lang.String`, and `java.lang.Class`.

Figure 4 presents the class declaration for the `Apprentice` class. We have omitted the bytecode for the `main` method, since it is shown in Figure 2.

Figure 5 presents the class declaration for the `Container` class. Note that it has one instance field, namely `counter`. It has only one method, the initialization

```
((0                              ; thread number
  (                              ; call stack (containing one frame)
   (0                            ;   program counter of frame
    nil                          ;   local variables of frame (none)
    nil                          ;   stack of frame (empty)
    (                            ;   bytecoded program of frame (main)
                                          ; byte offset from top
     (NEW "Container")                           ;  0
     (DUP)                                       ;  3
     (INVOKESPECIAL "Container" "<init>" 0)      ;  4
     (ASTORE_1)                                  ;  7
     (GOTO 3)                                    ;  8 (skip)
     (NEW "Job")                                 ; 11
     (DUP)                                       ; 14
     (INVOKESPECIAL "Job" "<init>" 0)            ; 15
     (ASTORE_2)                                  ; 18
     (ALOAD_2)                                   ; 19
     (ALOAD_1)                                   ; 20
     (INVOKEVIRTUAL "Job" "setref" 1)            ; 21
     (ALOAD_2)                                   ; 24
     (INVOKEVIRTUAL "java.lang.Thread" "start" 0)  ; 25
     (GOTO -17))                                 ; 28
    UNLOCKED                     ;   synchronization status of frame
    "Apprentice")                ;   class from which this method comes
   )                             ; end of call stack
   SCHEDULED                     ; scheduled/unscheduled status of thread
   nil))                         ; heap address of object representing this
                                 ;  thread (none)
```

Fig. 2. The Initial Thread Table

```
((0 ("java.lang.Class" ("<name>" . "java.lang.Object"))
    ("java.lang.Object" ("monitor" . 0)
                        ("mcount" . 0)
                        ("wait-set" . 0)))
 (1 ("java.lang.Class" ("<name>" . "ARRAY"))
    ("java.lang.Object" ...))
 (2 ("java.lang.Class" ("<name>" . "java.lang.Thread"))
    ("java.lang.Object" ...))
 (3 ("java.lang.Class" ("<name>" . "java.lang.String"))
    ("java.lang.Object" ...))
 (4 ("java.lang.Class" ("<name>" . "java.lang.Class"))
    ("java.lang.Object" ...))
 (5 ("java.lang.Class" ("<name>" . "Apprentice"))
    ("java.lang.Object" ...))
 (6 ("java.lang.Class" ("<name>" . "Container"))
    ("java.lang.Object" ...))
 (7 ("java.lang.Class" ("<name>" . "Job"))
    ("java.lang.Object" ...)))
```

Fig. 3. The Initial Heap

```
("Apprentice"                 ; class name
  ("java.lang.Object")        ; superclasses
  nil                         ; instance fields (none)
  nil                         ; static fields (none)
  nil                         ; constant pool (empty)
  (                           ; methods
   ("<init>"                  ;   initialization method name
    nil                       ;   parameters (none)
    nil                       ;   synchronization flag
    (ALOAD_0)                 ;   method body
    (INVOKESPECIAL "java.lang.Object" "<init>" 0)
    (RETURN))
   ("main"                    ;   main method name
    (|JAVA.LANG.STRING[]|)    ;   parameters (one)
    nil                       ;   synchronization flag
    (NEW "Container")         ;   method body
    ...
    (GOTO -17)))
   (REF 5))                   ; heap address of class representative
```

Fig. 4. The Apprentice Class Description

```
("Container"                  ; class name
 ("java.lang.Object")         ; superclasses
 ("counter")                  ; instance fields
 nil                          ; static fields (none)
 nil                          ; constant pool (empty)
 (("<init>"                   ; methods
   nil
   nil
   (ALOAD_0)
   (INVOKESPECIAL "java.lang.Object" "<init>" 0)
   (RETURN)))
 (REF 6))                     ; heap address of class representative
```

Fig. 5. The Container Class Description

method.

Finally, Figure 6 presents the class declaration for the `Job` class. It has one field, `objref`, into which the `Container` object will be stored. The class has an initialization method and the three user defined methods, `incr`, `setref`, and `run`. The instructions marked with `*` in the `incr` method are within the critical section of that method.

4. THE THEOREM AND ITS PROOF

The theorem we will prove is named Monotonicity and is stated formally in ACL2 below. It may be paraphrased as follows. Let `s1` be the state obtained by running an arbitrary schedule, `sched`, starting in the initial state of the Apprentice system, `*a0*`. Thus, by construction, `s1` is some arbitrary state reachable from `*a0*`. Let `s2` be the state obtained by stepping an arbitrary thread from `s1`. Thus, `s2` is any possible successor of `s1`. Suppose the value of the counter in `s1` is not `nil`. Then the

```
("Job"                          ; class name
 ("java.lang.Thread" "java.lang.Object") ; superclasses
 ("objref")                     ; instance fields
 nil                            ; static fields (none)
 nil                            ; constant pool (empty)
 (("<init>"                     ; methods
   nil
   nil
   (ALOAD_0)
   (INVOKESPECIAL "java.lang.Thread" "<init>" 0)
   (RETURN))
  ("incr"                       ;  incr method
   nil                          ;  parameters (none)
   nil                          ;  synchronization flag
   (ALOAD_0)                                        ;  0
   (GETFIELD "Job" "objref")                        ;  1
   (ASTORE_1)                                       ;  4
   (ALOAD_1)                                        ;  5
   (MONITORENTER)                                   ;  6
   (ALOAD_0)                                        ;  7 *
   (GETFIELD "Job" "objref")                        ;  8 *
   (ALOAD_0)                                        ; 11 *
   (GETFIELD "Job" "objref")                        ; 12 *
   (GETFIELD "Container" "counter")                 ; 15 *
   (ICONST_1)                                       ; 18 *
   (IADD)                                           ; 19 *
   (PUTFIELD "Container" "counter")                 ; 20 *
   (ALOAD_1)                                        ; 23 *
   (MONITOREXIT)                                    ; 24 *
   (GOTO 8)                                         ; 25
   (ASTORE_2)                                       ; 28
   (ALOAD_1)                                        ; 29
   (MONITOREXIT)                                    ; 30
   (ALOAD_2)                                        ; 31
   (ATHROW)                                         ; 32
   (ALOAD_0)                                        ; 33
   (ARETURN))                                       ; 34
  ("setref"                     ; setref method
   (CONTAINER)                  ; parameters
   nil                          ; synchronization flag
   (ALOAD_0)                                        ;  0
   (ALOAD_1)                                        ;  1
   (PUTFIELD "Job" "objref")                        ;  2
   (RETURN))                                        ;  5
  ("run"                        ; run method
   nil                          ; parameters
   nil                          ; synchronization flag
   (GOTO 3)                                         ;  0
   (ALOAD_0)                                        ;  3
   (INVOKEVIRTUAL "Job" "incr" 0)                   ;  4
   (POP)                                            ;  7
   (GOTO -5)))                                      ;  8
 (REF 7))                       ; heap address of class representative
```

Fig. 6. The Job Class Description

counter in s2 is either that in s1 or is one greater (in the 32-bit twos-complement sense of Java arithmetic).

Theorem: Monotonicity.

```
(let* ((s1 (run sched *a0*))
       (s2 (step th s1)))
  (implies (not (equal (counter s1) nil))
           (or (equal (counter s2)
                      (counter s1))
               (equal (counter s2)
                      (int-fix (+ 1 (counter s1)))))))
```

Our proof of the theorem is based on our definition of an invariant on states, named `good-state`. We prove three main lemmas. Lemma 1 says that `*a0*` is a `good-state`. Lemma 2 says that stepping an arbitrary thread, `th`, in a `good-state` produces a `good-state`. Lemma 3 says that if the counter is defined in a good state, then the desired relation holds after stepping an arbitrary thread.

From Lemmas 1 and 2 and induction it is easy to prove (Lemma 4) that every reachable state is good.

Proof of Monotonicity. From Lemma 4 we conclude that state `s1` of Monotonicity is a good state. Hence, from Lemma 3, we can conclude that the relation holds when the counter is defined. □

Lemma 1 is trivial to prove by computation, since `*a0*` is a constant and `good-state` is just a Lisp function we can evaluate.

Lemmas 2 and 3 are basically proved the same way, so we discuss only Lemma 2. We break the proof into three cases depending on the number, `th`, of the thread, being stepped. The first case (Lemma 2a) is when `th` is `0`; in this case, the `main` method is being stepped. The second (Lemma 2b) is when `th` is the number of some `SCHEDULED` thread other than `0`; such a thread will necessarily be running a `Job`. The third case (Lemma 2c) is when `th` is anything else; in this case, either `th` is not a thread number or indicates a still-`UNSCHEDULED` `Job` (one created by the `NEW` at `main 11` but not yet started by the `INVOKEVIRTUAL` at `main 25`). Stepping such a `th` is a no-op.

The complete ACL2 source text for our proof is available at `http://www.cs.-utexas.edu/users/moore/publications/m5/apprentice.lisp`.

5. THE GOOD STATE INVARIANT

Defining `good-state` is the crux of the proof. Roughly speaking, `good-state` characterizes the reachable states. The definition may be found in `http://www.-cs.utexas.edu/users/moore/publications/m5/apprentice.lisp`. We merely present the highlights here.

The formal definition of `good-state` is shown below. The variable `counter`, below, is bound to the value of the `counter` field of the `Container` at location `8` of the heap of `s`. We know that (in the good states) the `Container` created by the Apprentice system will be referenced by `(REF 8)`.

In addition, the variables `monitor` and `mcount` are bound, below, to the corresponding `java.lang.Object` fields of the `Container` object.

As the definition below makes clear, s is considered a good state provided it has a good class table, a good thread table, a good heap, and a certain condition holds on the counter, mcount, and monitor.

```
(defun good-state (s)
  (let ((counter (gf "Container" "counter" 8 (heap s)))
        (monitor (gf "java.lang.Object" "monitor" 8 (heap s)))
        (mcount (gf "java.lang.Object" "mcount" 8 (heap s))))
    (and (good-class-table (class-table s))
         (good-thread-table (thread-table s)
                            (- (len (heap s)) 1)
                            counter monitor mcount)
         (good-heap (thread-table s) (heap s))
         (or (equal (len (heap s)) 8)
             (and (integerp counter)
                  (if (equal mcount 0)
                      (equal monitor 0)
                    (and (equal mcount 1)
                         (< 0 monitor)
                         (< monitor (- (len (heap s)) 8)))))))))
```

The condition is that either the Container has not yet been allocated or else counter is an integer and either the mcount and monitor are 0 (meaning the Container is unlocked) or else the mcount is 1 and the monitor is the number of a Job thread in the Apprentice system.

The notions of "good" thread table and "good" heap are interdependent. Indeed, the two main aspects of the invariant concern both the thread table and the heap. A thread can be in its critical section only if it owns the monitor – a condition which requires inspecting the heap. Similarly, the object in the heap representing a particular Job must have its objref field set to the Container, unless main has not yet reached the setref for that Job – a condition which requires inspecting the thread table. Disentangling these two notions was one key to our success. Since we have to prove that good-state is invariant under step, it was important to make each conjunct above "as invariant" as possible.

Our disentangling is most apparent in the application of good-thread-table above. That predicate needs just four items from the heap: the heap address of the last object in the heap and the values of the counter, monitor, and mcount. Proving good-thread-table invariant is relatively easy for steps that do not change these quantities. The disentangling is not apparent in the application of good-heap above; but inside the definition of that predicate we apply an auxiliary predicate to the heap and pass it a flag that indicates whether the setref for the most recently created Job has been executed.

We now discuss the three main conjuncts above. The good-class-table predicate just recognizes the class table of the Apprentice system.

The good-thread-table predicate requires that thread 0 be running Apprentice main (and the methods it invokes) and that all other threads be Jobs appropriately spawned by main. By looking at the bytecoded programs and the class names of each frame of a call stack we can tell which methods are running; the program counter of the frame tells us where execution is. The call stack may be deep; for example the top frame may be running `java.lang.Thread.<init>`, in which case the frame below it must be suspended at instruction `1` of `Job.<init>` and the frame below that must be suspended at instruction `15` of `main`.

All the threads must be "good" in a sense that depends on which methods are running in them. To check these conditions it is necessary to know the heap address of the last object in the heap as well as the current `counter`, `mcount` and `monitor` values. The last object in the heap may or may not be a `Job` and, if a `Job`, may or may not be `SCHEDULED`, depending on where control is in `main`. If one of the `Job` threads is running the `incr` method and is in its critical section, then we must ensure that the corresponding thread owns the lock on the `Container` and that certain items (thought of by us as the value of the counter or derived from that value) in the thread's local state are accurate relative to the actual value of the `counter`.

We now turn to the `good-heap` predicate. One requirement on a "good" heap is that it start with the "standard prefix" which contains the eight previously mentioned objects representing the primitive and loaded classes of the Apprentice system. In addition, "good" heaps may subsequently contain the `Container` followed by a sequence of `Job`s, but whether these objects are present or not and their exact configurations depend on the program counter in the `main` program of thread `0`. The `Container` does not exist until the first instruction has been executed. Each new `Job` comes into being at instruction `11` but is not fully set up until the execution of the `setref` at instruction 21 is nearly completed. It is crucial that the `objref` field of a `Job` point to the `Container` when `incr` is invoked, but `Job`s do not come into existence satisfying that invariant.

In all, our `good-state` invariant involved the definition of 32 functions and predicates, consuming a total of 565 lines of pretty-printed ACL2. Many definitions could be eliminated at some cost to the perspicuity of the invariant and our function names are quite long, e.g., `Good-java.lang.Object.<init>-Frame`. Syntactic measures of complexity are misleading here. The best way to think of `good-state` is that it almost perfectly characterizes the reachable states. How does it fail?

Consider the call stack of thread `0` and, in particular, the frame in that stack running `main`. Is there a frame under that one? In fact, there is never a frame under the `main` frame in any reachable state. But it is not necessary to say so, because there is no `return` in the `main` frame. Our notion of `good-state` allows arbitrary frames under the `main` one. Similarly, we characterize the values of locals `1` and `2` of the `main` frame, but do not say there are no others.

The invariant was created manually, starting with the main idea: no two threads are in their critical section simultaneously. How is this said? If a `Job` is in the `incr` method and the program counter is between `7` and `24`, then the `monitor` of the `Container` is the thread number of the thread in question and the `mcount` of the `Container` is `1`. Working backward from there, we had to ensure that the object upon which `MONITORENTER` is called is indeed the `Container`, which is located at heap address `8`. That in turn forced us to require that the `objref` of the `Job` is the `Container`, etc.

We envision mechanized tools to fill in these simple but tedious aspects of the invariant; such tools are necessary if this method were to be used repeatedly on still larger Java programs. We have developed no such tools yet. Our interest in this exercise was in proving the invariant.

6. THE MECHANIZED PROOFS

Here is Lemma 2b. It says `good-state` is preserved when the thread being stepped, `th`, is running a `SCHEDULED` job. The first hypothesis establishes that `s` is a `good-state`. The next three establish that `th` is the number of a `Job` thread in such a state. The last hypothesis is largely redundant with the others. Essentially all it adds is that thread `th` is `SCHEDULED`. By explicitly saying the thread is "good" we make this lemma a little easier to prove.

Lemma 2b.
```
(implies (and (good-state s)
              (integerp th)
              (<= 1 th)
              (<= th (- (len (heap s)) 9))
              (good-thread th
                       'SCHEDULED
                       (assoc-equal th (thread-table s))
                       (gf "Container" "counter" 8 (heap s))
                       (gf "java.lang.Object" "monitor" 8 (heap s))
                       (gf "java.lang.Object" "mcount" 8 (heap s))))
         (good-state (step th s)))
```

The conclusion is that the result of stepping `th` is a `good-state`.

How should we go about proving this formula? Consider a case analysis on `s` based on the expansion of the `good-state` and `good-thread` hypotheses. Those hypotheses essentially describe all the possible states of thread `th`. For each of those states of thread `th`, we can determine the instruction that will be executed by the `step` expression in the conclusion. It is fairly obvious that one should not expand the `step` expression until such a determination is possible; the premature expansion of `step`, i.e., the expansion of `step` before the next instruction can be uniquely determined, results in an explosion of cases. In the worst case, it will result in the symbolic consideration of all possible next instructions, even those not occurring in class files in question. For this reason, we configure ACL2 so as not to expand `step` until the next instruction can be determined [Boyer and Moore 1996].

Even with this advice, if the theorem prover attacks Lemma 2b on its own, it expands both `good-state` expressions and the `good-thread` expression and considers all the cases. There are initially 28,944 of them, though many of them are contradictory and can be dismissed with further simplification. However, that was not the attack we wanted (indeed, we did not even consider letting the theorem prover try such a direct approach until we were assembling data for this paper).

The attack we used is: (i) expand the `good-state` and `good-thread` expressions in the hypothesis, break the formula into cases, and simplify maximally to eliminate impossible combinations, (ii) when no further simplification is possible, expand the `step` in the conclusion and simplify maximally, (iii) when no further simplification is possible, expand the `good-state` in the conclusion and simplify maximally. Phase (i) of this attack generates about 400 subgoals. Phase (ii) then expands the `step` expression in each of these. In most cases, phase (ii) expansion generates only one subgoal; but in about a quarter of the cases it generates up to 10 cases, most of which are dismissed by further simplification. In each of the remaining cases, phase (iii) expands the concluding `good-state`. Each of the resulting cases is proved.

In all, during the proof of Lemma 2b, approximately 2800 subgoals are considered and there are about 1000 tips of the proof tree. Each case requires about 150–200 lines (about 6000 bytes or 2–3 pages) of text to print, so the formulas are moderately large. The theorem prover generates a total of 19 MB of text to describe the proof, though of course we do not read it.[2] The theorem prover spends 2854 seconds (about 48 minutes) proving Lemma 2b on a 728 MHz Pentium III with 256 MB of RAM, running Allegro Common Lisp.

This same three-phase simplification strategy is provided as a hint in the proofs of Lemmas 2a (559 seconds), 3a (369 seconds), and 3b (1686 seconds). Lemmas 1, 2c, and 3c are trivial, requiring less than a tenth of a second each to prove.

In addition to the lemmas discussed in this paper, we had to prove approximately 75 helper lemmas. The need for these lemmas were discovered using "The Method" described in [Kaufmann et al. 2000b]. We paraphrase a few of the lemmas here, omitting many details.

—The length of a table (e.g., heap or thread-table) grows by one when a new entry is added.

—If the heap (thread-table) is good and a new object (thread) is allocated, the heap (thread-table) is still good.

—If the thread-table is good and a thread's status is switched from `UNSCHEDULED` to `SCHEDULED`, it is still good.

—The object representing thread i has heap reference `(REF` $i+8$`)`. (The arithmetic correspondence is unimportant; this theorem was a succinct way to establish that distinct threads have distinct representatives in the heap.)

—If the thread-table is good when the `Container` is not locked and a thread locks the `Container`, then the thread-table is good with the `Container` locked by that thread.

—If the thread-table is good for a certain value of the `counter` and the `Container` is locked by a given thread, then the thread-table is good for the next "highest" value of the `counter`.

Generally speaking, these are inductively proved lemmas that require only a few seconds each to prove.

A total of 85 theorems were stated by the user and proved by the system. ACL2 Version 2.6, running on a 728 MHz 256 MB Pentium III in Allegro Common Lisp requires 6869 seconds (about 115 minutes) to construct/check this proof. The proof would take approximately two-thirds as much time if Lemmas 2b and 3b were combined; most of the time for each of those is spent in phase (i) and (ii), which are identical for both lemmas. We proved them separately for pedagogical reasons.

[2]The theorem prover's output is most often read only when a case fails.

7. HUMAN EFFORT

How much time was spent developing this proof? That is a common question in connection with mechanized theorem proving projects. The question is hard to answer in this case because the project was done three times.

The first time, we used our simplest JVM thread model, named M4, which supports about 20 JVM bytecodes. We coded a version of Apprentice directly in M4 bytecode; the version had a `main` method that spawned exactly two `Jobs`. Our original motivation was simply to give us a multi-threaded example upon which we could test our then-new M4 model.

We decided to use the example as our first M4 proof exercise. Because the M4 model supports unbounded arithmetic, the `counter` in that system increases monotonically (no wrap-around). We proved this using the same approach described here: we defined a `good-state` predicate and proved its invariance and other properties using phased simplification. We invented phased simplification for that application and spent a day developing a complicated suite of rewrite rules and "destructor elimination" rules to "trick" ACL2 into performing phased simplification. The total amount of time spent on the 2-`Job` M4 version of Apprentice was one week, from testing of the M4 bytecode to the final "Q.E.D."

The proof was presented to the Austin ACL2 user's group meeting a few days later, on October 4, 2000. We had mentioned the problem as part of the round-table discussion the week before. At that time, Pete Manolios, a member of our group, had made some valuable comments about how to define the `good-state` invariant and had pursued his ideas on his own. At the October 4 meeting, Manolios also presented his proof of the 2-`Job` version. His proof was based on the observation that, except for the value of the counter, the 2-`Job` system is finite-state. Manolios' ACL2 proof involved a full state exploration in this abstract setting. He spent about two days on his proof.

Since our proof could be "easily" lifted to n `Jobs`, we changed the `main` method to create an unbounded number of `Jobs`. The `good-state` invariant was generalized appropriately and the theorem proved again (on October 8, 2000). In our 2-`Job` proof we had broken the invariance theorem into three lemmas, one for stepping thread `0`, one for stepping thread `1`, and one for stepping thread `2`. To do the n-`Job` case we invented the decomposition shown here. The total amount of time we spent lifting the 2-`Job` problem to the n-`Job` problem was 8 hours.

The next time we looked at the problem was in July, 2001. By then three things had changed: (a) we had M5 with its (bounded) `int` arithmetic and 138 realistically modeled JVM bytecodes, (b) we had `jvm2acl2` to convert class files to the M5 formalism mechanically, and (c) Moore had built support for phased simplification into the new release of ACL2.

We decided to use the Apprentice example as our first M5 proof exercise. We discarded the hand-coded M4 methods and used `jvm2acl2` to produce the `javac` bytecode shown here. The mechanically generated code was somewhat different than our original code, most significantly due to the use of `INVOKESPECIAL` to initialize each new object. In addition, the initial heap in the M5 model was different because of the existence of objects representing classes. This also changed the association of thread numbers to heap addresses, but in a linear way. The introduction of 32-bit arithmetic also required changes to the statement of Monotonicity. All these changes required a careful inspection and re-definition of the `good-state` invariant and the re-statement of some of the lemmas. We spent a total of about 20 hours porting the hand-coded M4 n-`Job` proof to the proof described here. About 6 of those hours were spent struggling with an impossible subgoal introduced by a careless mistake in our translation of a lemma. The mistake weakened the lemma so that it was provable but not useful.

8. RELATED WORK

Formally modeling computing machines operationally has a long tradition. McCarthy [McCarthy 1962] said "The meaning of a program is defined by its effect on the state vector." Mechanically analyzing programs with respect to a formally defined operational semantics also has a long tradition, especially in the Boyer-Moore community, where many techniques have been developed for it [Boyer and Moore 1996]. A good example of this is Yu's work [Boyer and Yu 1996] in which 21 of the 22 Berkeley C String Library subroutines were verified by mechanically analyzing the binary code produced by `gcc -o` for a Motorola 68020 model in Nqthm.

Turning to the modeling of Java and the JVM, the first mechanized formal model we know of was Cohen's "defensive JVM" [Cohen 1997] in ACL2. Cohen's machine includes type tags on all data objects so that type errors can be detected and signaled at runtime. It was designed for use in verifying the bytecode verifier. Cohen's machine does not include as many bytecodes as M5 nor does it include threads.

Our M5 is the fifth machine in a series of ACL2 models approaching the JVM. Our series was based on Cohen's machine and initially developed by Moore (with help from Cohen) to teach an undergraduate course at the University of Texas on modeling the JVM in ACL2. The sequential predecessor of M5 is discussed in [Moore 1999b], including how we use ACL2 to prove theorems about sequential bytecode programs.

ACL2 was used to model the Rockwell JEM1 microprocessor, the world's first silicon JVM, now marketed by Ajile Systems, Inc. The ACL2 model was used as the standard test bench on which Rockwell engineers tested the chip design against the requirements by executing compiled Java programs. The ACL2 model executed at approximately 90% of the speed of the previously used C model [Greve and Wilding 1998; Greve 1998]. In [Greve et al. 2000], Wilding and Greve describe how microprocessor models in ACL2 are made to execute fast. The model there executes at approximately 3 million simulated instructions per second on a 728 MHZ Pentium III host running Allegro Common Lisp.

The JEM1 model illustrates an advantage of executable formal models: they can be tested and compared to other models of the artifact. At Advanced Micro Devices, an executable ACL2 model of the RTL for the AMD Athlon™ floating-point square root was tested on 80 million floating-point vectors. The model computed the same answers as AMD's RTL simulator and this fact helped establish confidence in the formal model. ACL2 was then used to prove that the RTL for each elementary floating-point operation on the Athlon is compliant with the IEEE 754 floating-point standard [Russinoff 1998].

We now turn to related mechanized formal work other than that by the ACL2 community.

The Extended Static Checker (ESC) [Detlefs et al. 1998] is an example of a formal, practical and mechanized tool for establishing certain simple assertions about Java classes. ESC can check that no shared variable is accessed without holding a mutual exclusion lock on the variable ("simple locking discipline") which is also enforced by the dynamic checking tool Eraser [Savage et al. 1997]. While both ESC and Eraser can deal with dynamically allocated shared variables, they are incapable of dealing with general properties of systems.

The Java PathFinder [Brat et al. 2000; Visser et al. 2000] (JPF) is an explicit-state model-checker for programs written in Java. It can check certain kinds of invariants and deadlock. A tool with similar functionality is Bandera [Dwyer et al. 2001]. Both of these model-checking tools have been used to check properties of a Java version of the DEOS real-time operating system kernel, a program involving approximately 20 classes, 6 threads, 91 methods, 41 instance fields, and 51 static fields. The property was a complex time-partitioning requirement.

An anonymous referee of this paper used JPF to check a simplified version of the Apprentice problem and that experiment is quite illuminating as a comparison between model-checking and theorem proving.

To check the invariant, the referee added a new field, `prev`, to the `Container` and modified `incr` to store the old value of the `counter` into `prev` before incrementing the `counter` field. To cope with a limitation of the version of JPF being used (but fixed in more recent releases), the referee split the assignment statement in `incr` into two parts: (i) add `1` to the value of the `counter` and save the result into `temp`, and (ii) assign the value of `temp` to the `counter`. The referee then added an assertion at the end of the `incr` method to check `prev` $\leq$ `counter`. In addition, to keep the state-space "tractable" the referee modified `run` to call `incr` only three times and modified `main` to spawn only three `Jobs`. Let us call this the "finite Apprentice" example.

The referee then produced a "buggy version" of that problem by modifying `main` to generate a second `Container` and to reset temporarily the `objref` of a running `Job` so that the lock could be obtained on "the wrong" `Container`, as we described in Section 1. Both the buggy version and correct version of the finite Apprentice were submitted to JPF.

The referee writes "When carrying out the experiments above, it took me less than 5 minutes to set up the tool and start the model-check running. On the buggy version, the defect was found in less than a minute of run-time. For the correct version, the model-checker ran for 1 minute and 37 seconds (on a 450 MHz Pentium Xeon quad-processor running Linux)."

What is learned from this experiment? The fact that the buggy version was rejected with so little human effort is a powerful argument for the utility of such tools as JPF. As noted, we spent days working on our proof. Furthermore, less training is required to use a tool like JPF than to use ACL2 and hence more properties might be checked.

However, the property checked with JPF in this instance is much weaker than our property. The level of atomicity used by JPF in the experiment was at the "line of source code" level. To use that granularity and still detect the bug in the buggy version it is necessary to split the assignment statement into two lines. Had the referee not done that, the bug would not have been detected. JPF also supports bytecode-level interleaving but according to the JPF implementors [private communication] that is not often used because it tends to blow up the search space too much.

Another sense in which the property is weaker is that it is checked only when some thread is in the `incr` method, not "all the time." JPF can be used to check true invariance ([Havelund 1999] pp. 24) and the referee knew this but chose to weaken the property for search strategic reasons.

Somewhat more problematic is the use of a program variable, `prev`, to "capture" the property being checked. The experiment establishes that every time `incr` writes to the counter, `prev` $\leq$ `counter`. This does not mean the counter increases monotonically. The same invariant can be true for systems where the `counter` decreases, as long as `prev` is suitably set. The property checked "captures" the intuitive notion of "monotonicity" only if one understands how `prev` is being set elsewhere in the system. In our approach, the property being checked is stated outside of the target software.

Finally, because of Java's `int` arithmetic, the counter does not increase monotonically in the Apprentice problem even though it does so in the finite Apprentice problem. Given the limitations adopted to create finite Apprentice, `incr` is called at most nine times. Since the `counter` starts at `0`, it can only reach `9` and hence actually does increase every time it changes. But had the finite limitations allowed `incr` to be called more than 2^{31} times, the "invariant" `prev` $\leq$ `counter` would be found to be false. On a modern machine, the `counter` in Apprentice can be expected to decrease (wrap around) several times an hour. The invariant proved on finite Apprentice is invalid on the Apprentice problem.

None of this is meant to suggest that model-checking tools such as JPF are not extremely useful. The fact that the buggy finite Apprentice was detected with relatively little work is a demonstration of the value of such tools. But, whether we use model-checking or theorem proving, we must reflect carefully on the conclusions we draw from successful checks of simplified systems.

There are other theorem-proving based approaches to Java verification. One such tool is the LOOP tool [Berg et al. 2000; van den Berg and Jacobs 2001] which translates Java and JML (a specification language tailored to Java) classes into their semantics in higher order logic. As such, LOOP can be used as a front end for such theorem provers as PVS [Owre et al. 1992] and Isabelle [Nipkow and Paulson 1992]. However, LOOP currently deals only with sequential Java.

Other related work include [Pusch 1998; Barthe et al. 2001], where models of the JVM are formalized in Isabelle and Coq [Dowek et al. 1991]. In both efforts, the correctness of the bytecode verifier is addressed and the JVM models are largely concerned with type correctness rather than full functionality.

9. CONCLUSION

Readers familiar with monitor-based proofs of mutual exclusion, as well as users of ACL2 familiar with proving properties of operational models of microprocessors, will recognize our proof as fairly classical. The novelty here comes from (i) the complexity of the individual operations on the machine, (ii) the dependencies between threads, heap objects, and synchronization, (iii) bytecode-level interleaving, (iv) the unbounded number of threads, and (v) and proof engineering permitting "automatic" mechanical verification of the main code-related lemmas.

We are working on tools to help make proofs like this easier, including assistance with the formalization of invariants, visualization tools for symbolic simulation of state machine models, and standard simplification strategies.

We believe the Apprentice class is a good benchmark for formally based proof systems for Java. The Java expression of the class (Figure 1) is only about 30 lines long. The theorem can be stated in a few lines, given the definition of the bytecode semantics. The proof necessarily involves reasoning about the creation of instance objects and threads in the heap, the inheritance of fields from superclasses, pointer chasing and smashing, bounded arithmetic, infinite loops, the invocation of instance methods, dynamic method resolution, the starting of new threads, the reading and writing of fields in the heap, the use of monitors to attain synchronization between threads, and consideration of all possible interleavings or scheduling over an unbounded number of threads. Furthermore, Apprentice is not far from the "natural" domain of both model-checking and theorem proving technologies.

Despite the large size of our `good-state` predicate, the ACL2 theorem prover was up to the task of proving it invariant—with some key help from the user like the formulation of the necessary inductively proved lemmas and the suggestion to use phased simplification.

We are dismayed by the size of the `good-state` invariant even though we can imagine mechanically generating much of it.

It would be nice to be able to factor the proof so as to separate the activity of the `main` thread from that of the `Jobs`. We achieved a little such factoring in our decomposition of Lemma 2 into parts a, b, and c. However, there is still too much entanglement, as illustrated by the fact that the notion of whether the heap is "good" depends upon where the program counter is in thread `0`. One possibility is the transformation into a uniprocessor view as described in [Moore 1999a], though we do not see how to do this yet.

One reason we put this forward as a benchmark is so that we and others are challenged to find better solutions while preserving the challenging aspects of this formulation of the problem. Key aspects of the problem we solved include that threads and classes are objects in the heap, objects contain mutable pointers to other objects, objects inherit fields from superclasses, threads can be created, manipulated as objects, and started as processes, that synchronization is achieved through monitors that need not be respected by all code, and that an unbounded number of `Jobs` (objects and threads) are involved.

The problem is somewhat simpler if one assumes all the `Job` threads have been created, initialized, and started before monotonicity is considered. But realism compelled us to allow the `main` thread to continue to "boot up" new threads throughout the computation, preventing the system from ever achieving a "steady state" and forcing the invariant to deal with "ill-formed" (incompletely initialized) objects in the heap. Of particular note is the fact the invariant must allow the execution of code that writes to the `objref` field of a `Job`.

The problem is simpler still if one assumes only a small, fixed number of preexisting `Jobs`, as might actually be the case in some application areas such as embedded control. But, given trends in multi-threaded programming, it is worthwhile to develop techniques for dealing with large numbers of threads.

10. ACKNOWLEDGMENTS

We are grateful to David Hardin and Pete Manolios, each of whom made valuable suggestions in the course of this work. In addition, we thank the anonymous referee who model-checked the two versions of finite Apprentice with JPF. We deeply appreciate that contribution to this paper.

REFERENCES

BARTHE, G., DUFAY, G., JAKUBIEC, L., SERPETTE, B., AND DE SOUSA, S. M. 2001. A formal executable semantics of the JavaCard platform. In *ESOP 2001*, D. Sands, Ed. Vol. LNCS 2028. Springer-Verlag, 302–319.

BERG, J. V. D., HUISMAN, M., JACOBS, B., AND POLL, E. 2000. A type-theoretic memory model for verification of sequential Java programs. In *Recent Trends in Algebraic Development Techniques (WADT'99)*, D. Bert and C. Choppy, Eds. Number 1827 in LNCS. Springer, 1–21.

BOYER, R. S. AND MOORE, J. S. 1996. Mechanized formal reasoning about programs and computing machines. In *Automated Reasoning and Its Applications: Essays in Honor of Larry Wos*, R. Veroff, Ed. MIT Press, 147–176.

BOYER, R. S. AND MOORE, J. S. 1997. *A Computational Logic Handbook, Second Edition.* Academic Press, New York.

BOYER, R. S. AND YU, Y. 1996. Automated proofs of object code for a widely used microprocessor. *Journal of the ACM 43,* 1 (January), 166–192.

BRAT, G., HAVELUND, K., PARK, S., AND VISSER, W. 2000. Java PathFinder - a second generation of a Java model checker. In *Workshop on Advances in Verification.* `http://ase.arc.nasa.gov/jpf/wave00.ps.gz`.

COHEN, R. M. 1997. The defensive Java Virtual Machine specification, Version 0.53. Tech. rep., Electronic Data Systems Corp, Austin Technical Services Center, 98 San Jacinto Blvd, Suite 500, Austin, TX 78701.

DETLEFS, D. L., LEINO, K. R. M., NELSON, G., AND SAXE, J. B. 1998. Extended static checking. Tech. Rep. TR 159, Compaq Systems Research Center. Dec.

DOWEK, G., FELTY, A., HERBELIN, H., HUET, G., PAULIN, C., AND WERNER, B. 1991. The Coq proof assistant user's guide, Version 5.6. Tech. Rep. TR 134, INRIA. Dec.

DWYER, M., HATCLIFF, J., JOEHANES, R., LAUBACH, S., PASAREANU, C., VISSER, W., AND ZHENG, H. 2001. Tool-supported program abstraction for finite-state verification. In *Proceedings of the 23rd International Conference on Software Engineering.*

GREVE, D., WILDING, M., AND HARDIN, D. 2000. High-speed, analyzable simulators. See Kaufmann et al. [2000a], 113–136.

GREVE, D. A. 1998. Symbolic simulation of the JEM1 microprocessor. In *Formal Methods in Computer-Aided Design – FMCAD*, G. Gopalakrishnan and P. Windley, Eds. LNCS 1522. Springer-Verlag.

GREVE, D. A. AND WILDING, M. M. Jan. 12, 1998. Stack-based Java a back-to-future step. *Electronic Engineering Times*, 92.

HAGGAR, P. 2000. *Practical Java Programming Language Guide.* Addison-Wesley.

HAVELUND, K. 1999. Java PathFinder user guide. Tech. Rep. `http://ase.arc.nasa.gov/havelund/Publications/jpf-manual.ps.Z`, NASA Ames Research Center. August.

KAUFMANN, M., MANOLIOS, P., AND MOORE, J. S., Eds. 2000a. *Computer-Aided Reasoning: ACL2 Case Studies.* Kluwer Academic Press.

KAUFMANN, M., MANOLIOS, P., AND MOORE, J. S. 2000b. *Computer-Aided Reasoning: An Approach.* Kluwer Academic Press.

LINDHOLM, T. AND YELLIN, F. 1999. *The Java Virtual Machine Specification (Second Edition).* Addison-Wesley.

MANSON, J. AND PUGH, W. 2001. Semantics of multithreaded Java. Tech. Rep. TR 4215, Computer Science Department, University of Maryland. May. `http://www.cs.umd.edu/~pugh/java/-memoryModel/semantics.pdf`.

MCCARTHY, J. 1962. Towards a mathematical science of computation. In *Proceedings of IFIP Congress*. North-Holland, 21–28.

MOORE, J. S. 1999a. A mechanically checked proof of a multiprocessor result via a uniprocessor view. *Formal Methods in System Design 14,* 2 (March), 213–228.

MOORE, J. S. 1999b. Proving theorems about Java-like byte code. In *Correct System Design – Recent Insights and Advances*, E.-R. Olderog and B. Steffen, Eds. LNCS 1710, 139–162.

MOORE, J. S. AND PORTER, G. 2001. An executable formal JVM thread model. In *Java Virtual Machine Research and Technology Symposium (JVM '01)*. USENIX. `http://www.cs.utexas.edu/users/moore/publications/m4/model.ps.gz`.

NIPKOW, T. AND PAULSON, L. C. 1992. Isabelle-91. In *Proceedings of the 11th International Conference on Automated Deduction*, D. Kapur, Ed. Springer-Verlag LNAI 607, Saratoga Springs, NY, 673–676. System abstract.

OWRE, S., RUSHBY, J., AND SHANKAR, N. 1992. PVS: A prototype verification system. In *11th International Conference on Automated Deduction (CADE)*, D. Kapur, Ed. Lecture Notes in Artificial Intelligence, Vol 607, Springer-Verlag, 748–752.

PUSCH, C. 1998. Formalizing the Java virtual machine in Isabelle/HOL. Tech. Rep. TUM-I9816, Institut für Informatik, Technische Universiät München. See URL `http://www.in.tum.de/~pusch/`.

RUSSINOFF, D. 1998. A mechanically checked proof of IEEE compliance of a register-transfer-level specification of the AMD-K7 floating-point multiplication, division, and square root instructions. *London Mathematical Society Journal of Computation and Mathematics 1*, 148–200. http://www.onr.com/user/russ/david/k7-div-sqrt.html.

SAVAGE, S., BURROWS, M., NELSON, G., SOBALVARRO, P., AND ANDERSON, T. 1997. Eraser: a dynamic data race detector for multithreaded programs. *ACM Transactions on Computer Systems 15,* 4 (Nov).

STEELE, JR., G. L. 1990. *Common Lisp The Language, Second Edition.* Digital Press, 30 North Avenue, Burlington, MA 01803.

VAN DEN BERG, J. AND JACOBS, B. 2001. The LOOP compiler for Java and JML. In *TACAS 2001*. Vol. LNCS 2031. Springer, 299–313.

VISSER, W., HAVELUND, K., BRAT, G., AND PARK, S. 2000. Model checking programs. In *IEEE International Conference on Automated Software Engineering.* Grenoble, France.