Proving Properties of Java Threads

J Strother Moore*
George Porter†

November 12, 2000

Abstract

We describe a mechanically checked proof of a property of a system of small bytecoded programs involving an unbounded number of threads and synchronization via monitors. The bytecode programming language we study is modeled on that for the Java Virtual Machine. The formal semantics of our language is given by an operational model 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, conditionals, 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 or scheduling over an unbounded number of threads. This paper develops techniques for reasoning mechanically about multi-threaded Java programs. 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.

Keywords: parallel, distributed computation, mutual exclusion, operational semantics, verification, Java Virtual Machine

1 The Apprentice System in Java

The problem we consider is suggested by 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.

Our formalization of this problem is not sufficiently faithful to the semantics of Java for us to claim that we have verified the Java classes of Figure 1. For example, in our model, arithmetic is unbounded. We discuss other inaccuracies later. But we will exhibit a formalization of the Apprentice problem that shares a great many features with the Java system shown in Figure 1.

We will prove that under all possible scheduling regimes, the value of the counter in our model increases weakly monotonically. To state this formally requires formalizing a non-trivial amount of the semantics of

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

```
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 1: The Apprentice Example in Java

objects, threads and synchronization. The proof exercises a great deal of the formal machinery and illustrates what is involved in proving multi-threaded Java programs correct.

Many readers may think the monotonicity claim is so trivial as not to deserve proof. We therefore start by demonstrating the contrary. In our discussion we address ourselves to Java (rather than our model) but we ignore the finite bounds of Java `int` arithmetic.

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 if we did no 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` and is then suspended. Suppose that the second `Job` is then 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` to the 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 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. It is possible to modify the Apprentice system slightly so that there is exactly one line that writes to the `counter` field of a `Container`, namely, the one in the synchronized block of `incr`, but the counter can still 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, 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 [11], where Haggar writes "Do not reassign the object reference of a locked object.")

Imagine a different `main`. This one starts two `Jobs`, say job_1 and job_2, creates a second `Container` object, 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 insure that no thread changes the `objref` field of the `Job` holding the lock on the `Container`? If we could insure syntactically that the system contains no write to any `objref` of a `Job`, we would be safe. But the Apprentice system contains such a write in the `setref` method. The required analysis is not just syntactic.

We hope this discussion makes it clear that it is nontrivial to establish that the code in Figure 1 increments the counter monotonically. It is not enough to know that mutual exclusion is achieved. Certain objects in the heap must be immutable even though no locks are held on them.[1] Rather than formulate more theories on how the correctness of Apprentice might be "observed" we will prove it.

Our approach is to compile the Apprentice system into the bytecode language of a machine similar to the Java Virtual Machine (JVM) and then prove that under every schedule the counter increases weakly monotonically. One might argue that compiling the Java into bytecode is a refinement of the problem that introduces significant unnecessary complexity. We believe otherwise. The Java semantics of locking, heap allocation of threads and objects, and (even clean) pointer manipulation is complex and deserves attention to such detail. Furthermore, the JVM is such a good distillation of the semantics of Java that no significant new complexity is introduced. The complexity is already in the problem and is merely illuminated by the bytecode.

2 Background

The long range goal of this research is to develop a system in which properties of Java programs, including multi-threaded Java programs, can be verified mechanically. Our approach is to compile the Java programs into bytecode and verify the bytecode with respect to a formal operational semantics. This follows a long-established tradition in the Boyer-Moore school of program verification, a particularly convincing example of which is the use of a formal model of the Motorola 68020 to verify the Berkeley C String Library by verifying the binary produced by the gcc compiler [6].

Mechanically checked proofs of program properties requires the construction or adoption of a mechanical theorem prover of some sort. This also entails the choice of a formal mathematical logic in which the programming language is modeled, properties are specified, and proofs are constructed. We use the ACL2 logic and its theorem prover [14]. This is a general-purpose essentially quantifier-free first-order logic of recursive functions based on a functional subset of Common Lisp [20]. ACL2 is the successor to the Boyer-Moore theorem prover Nqthm [3, 5]. Formulas in this logic look like Lisp expressions.

Function definitions in ACL2 generally take the form `(defun` f `(`$v_1 \ldots v_n$`)` β`)`, where f is the symbol naming the function, the v_i are the formal parameters of the function, and β is the body of the function. Computationally, the value of an application of the function symbol to some actual expressions is obtained by evaluating the actual expressions, binding the v_i to the corresponding values, and then evaluating β. Logically, a definition adds the axiom `(`f $v_1 \ldots v_n$`)` $= \beta$. Restrictions are enforced to insure the preservation of the soundness of the logic.

Below is the ACL2 definition of the factorial function.

```
(defun factorial (n)
  (if (zp n)
      1
    (* n (factorial (- n 1)))))
```

`Factorial` is both a Lisp program and an axiomatically described mathematical function about which we can prove theorems.

[1] One could take the position that the Apprentice system is "bad code." It would be better to assign `this.objref` to some local variable, `obj`, upon entry to `incr`, and then lock, fetch, and update the `counter` field of `obj`. But our goal in this paper is not to find another way to implement a counter but to prove that the given code is correct.

Here is an ACL2 theorem about the function above, stating the fact that factorial grows monotonically

```
(defthm factorial-grows-monotonically
  (implies (and (natp n)
                (natp m)
                (< 1 m)
                (< n m))
           (< (factorial n) (factorial m))))
```

The formula above is an implication from a conjunction of four hypotheses or antecendents to a conclusion. The variable symbols `n` and `m` are free and can be thought of as implicitly universally quantified on the far outside of the formula. The hypotheses restrict `n` and `m` to natural numbers such that both `1` and `n` are less than `m`. The concusion is an inequality stating that `n` factorial is less than `m` factorial. This formula can be derived from the ACL2 axioms, the definition of `factorial`, mathematical induction, and other rules of inference.

We wish, ultimately, to formalize the semantics of Java by modeling the Java Virtual Machine. As noted, our model is not as faithful to the actual semantics as would be required for us to claim that our programs are JVM programs. To avoid confusion we refer to our model as "the M4 machine" (because it is the fourth in a sequence of machines approaching the JVM in complexity.)

From past experience with other formal models we believe our model can be elaborated while maintaining its basic structure and utility for verification. Evidence of this is available in [7, 8, 13] where appropriately elaborate models of commercial applications are discussed, including bit- and cycle-accurate floating point components for the AMD K5 and Athlon microprocessors, bit- and cycle-accurate models of a Motorola DSP chip, and verification of microcode extracted from the ROM of such a chip.

The model of the M4 is completely independent of the Apprentice example. It can be used to prove theorems about other M4 classes. Therefore, we next describe our formal model of M4. Then we show the formalization of the Apprentice system; this largely consists of exhibiting our bytecode for the methods in Figure 1. Following that, we state the goal theorem formally. Next, we familiarize the reader with some details of our M4 model by executing it on parts of the Apprentice problem. We then formally state an invariant maintained by the Apprentice system. We describe a mechanically checked proof that the property is, indeed, invariant. The proof was checked with the ACL2 theorem prover. Finally, we conclude with a discussion of our model and the challenges presented by the Apprentice problem.

3 Our Virtual Machine, M4

We model M4 operationally. That is, we adopt an explicit representation of M4 states and we write, in ACL2's Lisp subset, an interpreter for the desired bytecode. One may view the model as a Lisp simulator for M4. The design of M4 is based on a study of the Java Virtual Machine specification [15], but has been simplified. The JVM semantics of bytecode instructions mentioned below are specified in [15].

An M4 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 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 (pc) and the bytecoded method body, a table associating local variable names with values, an operand stack, and a synchronization flag indicating whether the method currently executing is synchronized. Unlike the JVM, the local variables of an M4 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 whose keys are the successive classes in the superclass chain of the object and whose values are tables mapping immediate field names to values.

The structure of heap addresses is unimportant but they can be distinguished from integers and other data types. On M4, a heap address is a list of the form `(REF` i`)`, where i is a natural number. If the value of `i` is i, then the value of the ACL2 expression `` `(REF ,i) `` is `(REF` i`)`. This is just the "backquote" convention of Common Lisp [20].

One difference between M4 and the JVM is that on M4 the `NEW` instruction, which allocates an object of a given class in the heap, is responsible for the object's instantiation; all fields are initialized to `0`. The JVM's `NEW` instruction leaves the fields uninitialized and they are initialized by invoking special constructor methods.

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 on M4. 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 sychronization status flag, and a list of bytecoded instructions. M4 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 pc 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`, `ALOAD`, etc) and `ADD` instructions, we ignore the finite range of arithmetic data, and we count pc offsets in number of instructions rather than number of bytes.

These and most of the other discrepancies between M4 and the JVM are matters of detail that (a) drastically affect the properties that hold of their respective programming languages but which (b) do not affect the basic structure of the model or our ability to use the model to study proof techniques.

Here is our definition of the ACL2 function `execute-PUSH` which we use to give semantics to the M4 `PUSH` instruction. An example `PUSH` instruction is `(PUSH 3)`. This instruction is comparable to `ICONST_3` or `BIPUSH 3` on the JVM: execution pushes `3` onto the operand stack.

```
(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)))
```

Our function takes three arguments, named `inst`, `s`, and `th`. The first is an instruction of the form `(PUSH` arg_1`)`. The second is the M4 state. The third is a thread number. `Execute-PUSH` returns the "next" state, produced by executing `(PUSH` arg_1`)` in thread `th` of state `s`. The next state is constructed, above, with the function `make-state`, which takes three arguments: the thread table, heap and class table of the desired state. Inspection of the `make-state` expression above reveals that the last two components of the next state are the same as those of `s`. The thread table is modified by replacing the entry for `th` by a new thread. The new thread'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 pc is advanced by `1`, the locals remain unchanged, the stack has arg_1 pushed on it, and the method program and synchronization flag are unchanged.

We formalize a variety of instructions in this style, including `POP`, `LOAD`, `STORE`, `ADD`, `MUL`, `GOTO`, `IFEQ`, `IFGT`, `INVOKEVIRTUAL`, `RETURN` (a "void" return), `XRETURN`, `NEW`, `GETFIELD`, `PUTFIELD`, `MONITORENTER`, and `MONITOREXIT`. For each such opcode *op* we define an ACL2 function `execute-`*op*.

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. 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 `sch` in the expression `(run sch s)` we can address ourselves to particular scheduling regimes.

Lisp programmers will recognize our M4 system as a simulator for the machine we have in mind. However, unlike conventional simulators, M4 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/m4/m4.lisp`. For some additional discussion of this model, see [19], where we explain how `INVOKEVIRTUAL` is specified. In the examples below where we list M4 bytecodes we list the corresponding JVM bytecodes as a means of documenting our instruction set, modulo the simplifications listed below. See [15].

The basic structure of M4 owes much to Rich Cohen's "defensive JVM" model [9]. Cohen's ACL2 structures for frames, the call stack, references, instance objects, the heap and the class table are more or less identical to ours. Cohen provides additional Java data types, explicitly type tagged as in [16] and accurately modeled with respect to the JVM [15]; unlike Cohen's machine, ours has provisions for multiple threads, `Thread` objects, and monitors. Historically, Cohen's machine came first (1996) and our sequence of machines, culminating in M4, evolved from 1997 to the present as part of an undergraduate course at the University of Texas on modeling the JVM.

The single-threaded predecessor to M4 is discussed in [18], with more background about proofs, including proofs of methods that modify inherited fields in objects and illustrate method resolution. The definitive paper for describing how to model machines in the Boyer-Moore tradition is [4], where Boyer and Moore describe their so-called "small machine" model, which formed the basis of much early work on verified microprocessors and the CLI Stack [1, 12, 10, 22, 16].

M4 omits many features of the JVM. Among the more glaring omissions are

- accurate support for the JVM primitive data types like ints, doubles, arrays, floats, etc.,

- support for syntactic typing both in the naming convention in the instruction set (e.g., `IADD` versus `DADD`) and field and method signatures,

- support for field and method access modes,
- provision of static fields,
- proper handling of field initialization,
- 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.

Nevertheless, we believe our model is adequate as a vehicle for studying basic mechanized proof techniques for dealing with Java programs, including multi-threaded applications.

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 is not supported by a mechanized proof system and is focused on only one aspect of Java semantics. Mechanically checked proofs about some aspects of simple Java programs have been constructed with several theorem provers, including HOL, Isabelle, and PVS. See, for example, [21].

Perhaps the most striking distinction between our model and other mechanically supported "Java-like" semantic models is our provision of a wide range of Java features rather than just one aspect (e.g., type checking, inheritance, exceptions, etc). Our work is also distinguished by being cast in a formally defined operational (and executable) semantics. Because we formalize the semantics (rather than merely provide a shallow embedding into some proof system) we can prove theorems about the model, not just about particular Java methods or classes.

We are unaware of mechanically checked proofs (other than those reported here) of Java classes that use multi-threading.

We know of no mechanically checked proofs (our included) of correctness properties of significant Java applications; the field is still in its infancy.

4 The Apprentice System in Bytecode

In Figure 2 we exhibit the M4 bytecode for the `main` method of the `Apprentice` class. To the right we list the M4 pc and the output of Sun's Java compiler for `main` as displayed by `javap`. The M4 code was hand-generated. Note that on M4, variables are named, so we write `(STORE CONTAINER)` and `(LOAD JOB)` rather than `astore_1` and `aload_2`. In addition, we are not concerned with type-checking on M4 so we have one load instruction instead of one for each type of data and we make no provision for signatures. Recall also that the M4 `new` instruction initializes all fields of the new object to `0`, whereas on the JVM new objects are initialized by a constructor method.

In Figure 3 we show the bytecode for the `incr` method in the M4 `Job` class description. No provision is made for exceptions on M4.

Note that the `incr` method first obtains the object in the `objref` field of its `THIS` object and stores that in the local variable named `TEMP`. Then (at pcs `3` and `4`) it obtains the lock on that object by loading `TEMP` and doing `MONITORENTER`. If execution arrives at pc `4` and the `mcount` field of the `Container` object is non-`0`

```
(NEW "Container")                        ; 0     new #2 <Class Container>
                                                 dup
                                                 invokespecial #7 <Method Container()>
(STORE CONTAINER)                        ; 1     astore_1
(NEW "Job")                              ; 2     new #3 <Class Job>
                                                 dup
                                                 invokespecial #8 <Method Job()>
(STORE JOB)                              ; 3     astore_2
(LOAD JOB)                               ; 4     aload_2
(LOAD CONTAINER)                         ; 5     aload_1
(INVOKEVIRTUAL "Job" "setref" 1)         ; 6     invokevirtual #10 <Method ...setref...>
(LOAD JOB)                               ; 7     aload_2
(INVOKEVIRTUAL "Job" "start" 0)          ; 8     invokevirtual #11 <Method ...start...>
(GOTO -7)                                ; 9     goto our pc 2
```

Figure 2: Bytecode for Method Apprentice main

```
(LOAD THIS)                              ; 0     aload_0
(GETFIELD "Job" "objref")                ; 1     getfield #8 <... objref>
(STORE TEMP)                             ; 2     astore_1
(LOAD TEMP)                              ; 3     aload_1
(MONITORENTER)                           ; 4     monitorenter
(LOAD THIS)                              ; 5     aload_0
(GETFIELD "Job" "objref")                ; 6     getfield #8 <... objref>
(LOAD THIS)                              ; 7     aload_0
(GETFIELD "Job" "objref")                ; 8     getfield #8 <... objref>
(GETFIELD "Container" "counter")         ; 9     getfield #6 <... counter>
(PUSH 1)                                 ;10     iconst_1
(ADD)                                    ;11     iadd
(PUTFIELD "Container" "counter")         ;12     putfield #6 <... counter>
(LOAD TEMP)                              ;13     aload_1
(MONITOREXIT)                            ;14     monitorexit
                                         ;       goto M4 pc 15
                                         ;       ... exeception code ...
(LOAD THIS)                              ;15     aload_0
(XRETURN)                                ;16     areturn
                                                 Exception table:
                                                  ...
```

Figure 3: Bytecode for Method Job incr

then our semantics of MONITORENTER makes it a no-op.[2] Thus, in our model, execution of the blocked thread is effectively suspended (when it is stepped nothing happens) until some other thread releases the lock.

Note that at instructions 13 and 14, the method releases the lock on TEMP. The region between the MONITORENTER and MONITOREXIT is the critical section of incr.

5 The Apprentice Monotonicity Theorem

In Figure 4 we show a complete M4 state containing the Apprentice example. In this state, called *s0*, there is one thread, containing the main method of Apprentice, poised to execute the instruction at pc 0. Note that the local variables, CONTAINER and JOB, are unitialized and the stack is empty. The heap in *s0* is empty. The class table contains a class declaration for Container (which extends Object) with a counter field and no methods. It also contains a declaration for Job, which extends Thread and Object, with fields x and objref and the methods incr, run and setref. Class declarations for Thread and Object are automatically provided by our function make-class-def.

We will prove the following theorem.

Theorem. Monotonicity of Apprentice:

```
(implies (and (integerp n)
              (<= 0 n)
              (integerp m)
              (<= n m))
         (<= (counter (runn n any-schedule *s0*))
             (counter (runn m any-schedule *s0*))))
```

We define (runn n schedule s) to run the first n steps of schedule starting in state s. The theorem says that if m is at least n, then the value of the counter after running m steps of any schedule is at least the value of the counter after running n steps of that schedule: the counter never decreases.

6 Some Details of Our Model

Before beginning the proof it is helpful to familiarize ourselves with the model by executing it.

Let (repeat n th) be a list of length n, each element of which is th. Thus, (repeat 3 0) is the list (0 0 0). The expression (run (repeat 2 0) *s0*) evaluates to a state, which we will call *s2*. This state is like *s0* but different in three ways. First, the pc of the topmost frame in the call stack of thread 0 of *s2* is 2, because the first two instructions of main, (NEW "Container") and (STORE CONTAINER), have been executed. Second, the local variable bindings of that *s2* frame are now equal to '((CONTAINER . (REF 0)) (JOB . NIL)) because the NEW allocated an object at heap address 0 and pushed a reference to that object onto the stack, and then the STORE popped that reference into local variable CONTAINER. Third, the heap of *s2* has the following instance object allocated at address 0.[3]

```
(("Container" ("counter" . 0))
 ("Object"    ("monitor" . 0) ("mcount" . 0) ("wait-set" . 0)))
```

This is a Container object with the field counter (and inheriting from Object the fields monitor, mcount, and wait-set). All the fields are initialized to 0.

[2] Technically, we should say "if the mcount field is non-0 and the monitor is some other thread," but in the Apprentice system we can show that whenever execution arrives at this MONITORENTER the former conjunct implies the latter.

[3] We do not think of the heap address as being where the JVM might allocate the object. It is an abstract name for the pointer. By choosing successive integers in chronological order of creation we make it easier to express certain invariants.

```
(make-state
 (make-tt                                                ;;; Thread Table
  (push                                                   ;;  call stack of thread 0
    (make-frame 0                                          ;   top frame, pc = 0
                '((CONTAINER . NIL) (JOB . NIL))           ;   local variables
                nil                                        ;   operand stack
                '((NEW "Container")                        ;   bytecode of
                  (STORE CONTAINER)                        ;   current method
                  (NEW "Job")                              ;   (main)
                  (STORE JOB)
                  (LOAD JOB)
                  (LOAD CONTAINER)
                  (INVOKEVIRTUAL "Job" "setref" 1)
                  (LOAD JOB)
                  (INVOKEVIRTUAL "Job" "start" 0)
                  (GOTO -7))
                'UNLOCKED)
    nil))
 nil                                                      ;;; Heap (empty)
 (make-class-def                                          ;;; Class Table
  (list (make-class-decl "Container"                       ;;  Container class
                         '("Object")                        ;   extends Object
                         '("counter")                       ;   fields
                         '())                               ;   methods
        (make-class-decl "Job"                             ;;  Job class
                         '("Thread" "Object")               ;   extends Thread (and Object)
                         '("x" "objref")                    ;   fields
                                                            ;   methods
                         '(("incr" () nil                   ;    method name, formals, sync flag
                            (LOAD THIS)                     ;    and bytecode
                            (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: The M4 State *s0*

Any schedule with exactly two 0's in it would do the same thing to `*s0*`, since there are no threads other than thread `0`. Stepping a thread other than `0` in `*s0*` or its immediate successor has no effect.

Running thread `0` further from `*s2*` creates a new `Job` object in the heap with heap address `1` and stores a reference to that object into the variable `JOB`. The program then pushes that reference onto the stack and pushes the `Container` reference on top of it. Then it invokes a one argument method named `setref`. Since the object two deep on the stack is of class `Job` and there is a `setref` method in `Job`, it is that method that is invoked. A new frame is pushed on the call stack of thread `0`, poised to execute the four bytecodes in that `setref` method. In that new frame, the variable `THIS` has the `Job` reference as its value and the variable `R` has the `Container` reference as its value. Execution of the `setref` method body deposits the `Container` reference into `objref` field of the `Job` object and returns no results to `main`. `(REF 1)` in the heap at this point refers to the `Job` object

```
(("Job" ("x" . 0) ("objref" . (REF 0)))
 ("Thread")
 ("Object" ("monitor" . 0) ("mcount" . 0) ("wait-set" . 0))).
```

Note that the `objref` field of `(REF 1)` contains `(REF 0)`; that is, the `objref` field of the `Job` just created now points to the `Container` object created earlier. This is crucial because it is through this field that the `incr` method will get the object upon which to synchronize and update.

The creation of the `Job` object by `NEW` also adds a new thread to the thread table (because `Job` extends `Thread`), poised to execute the `run` method of class `Job`. The thread will be called thread `1`. The thread table entry for thread `1` indicates that the corresponding `Job` object is at `(REF 1)` in the heap. We call this the "manifestation" of the thread as an object in the heap. Separating the `Job` object from the local state of the corresponding thread makes it easier to reason about threads, for otherwise the heap would be changing on every instruction.

The entry for thread `1` in the thread table is initially marked `UNSCHEDULED`. But when the `start` method is invoked on the `Job`, the thread entry is marked `SCHEDULED`. Thereafter, stepping either thread `0` or `1` will have a non-trivial effect on the state. The next time thread `0` is stepped, the `(GOTO -7)` is executed and the pc is repositioned at instruction 2, thereby setting up the creation of another thread.

In all, it takes twelve steps of thread `0` to create and start a new `Job`. The process changes the thread table and the heap, and writes to the `objref` field of a `Job` in the heap.

7 The Invariant

Our proof of the monotonicity theorem follows a classic mutual-exclusion invariant argument. The novelty is merely in the details imposed by Java's monitors and the pervasive use of the heap by all threads.

The key step in our proof is to define the function `good-state` so that we can prove the following lemma.

Lemma. `good-state` *invariant*

```
(implies (good-state s)
         (and (good-state (step s th))
              (<= (counter s) (counter (step s th)))))
```

The lemma says that if `s` is a good state, then stepping `s` on any thread, `th`, produces a good state whose counter value is at least that of `s`. Our main theorem follows from this Lemma by induction but we do not discuss that proof here. The definition of `good-state` is the crux of the proof and it is what we focus on in this paper.

We use the adjective "good" here, loosely, to mean reachable from `*s0*`. In fact, our good states include all the reachable states and more. We comment on this in the concluding remarks.

Here is the definition of `good-state`. It begins by letting the variables `counter`, `monitor`, and `mcount` be the values of the corresponding fields of the `Container` at heap address 0. When we speak of "the counter," "the monitor," or "the mcount" of a state, we mean the values of the corresponding `gf` ("get-field") expressions below; the previously mentioned function `counter` is defined in terms of `gf`.

```
(defun good-state (s)
  (let ((counter (gf "Container" "counter" 0 (heap s)))
        (monitor (gf "Object" "monitor" 0 (heap s)))
        (mcount (gf "Object" "mcount" 0 (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 (heap s) nil)
             (if (equal mcount 0)
                 (equal monitor 0)
               (and (equal mcount 1)
                    (< 0 monitor)
                    (< monitor (len (heap s)))))))))
```

After binding those three variables, `good-state` checks four conjuncts. The first, the `good-class-table` expression, checks that the class table of `s` is that of `*s0*`; class tables on M4 never change. We discuss `good-thread-table` and `good-heap` expressions below. The final conjunct says that either the heap is empty or else the mcount is `0` or `1`; in the former case, the monitor is `0` and in the latter the monitor lies strictly between `0` and the length of the heap. The monitor, which is in fact always an integral thread number of a scheduled thread is not so restricted by our invariant. We just need to know that thread 0 never holds the lock and that threads are never created holding the lock.

The thread table and the heap are closely connected. For example, if the `main` thread is inside the invocation of `setref`, then the `Job` whose `objref` field is to be set must be the most recently created `Job` object in the heap. (Otherwise, `setref` might be used to change the `objref` field of a running thread.) If a thread is in the critical section of `incr`, then certain obvious conditions must hold on the monitor and mcount to insure mutual exclusion. These observations show that to state invariants on the thread table we must have access to the heap. Conversely, invariants on the heap involve the thread table, e.g., every `Job` in the heap must be the manifestation of a thread in the thread table. More subtly, the `objref` of every `Job` in the heap must point to the `Container` at `(REF 0)` – except possibly the most recently created `Job`. The most recently created `Job` does not point to the `Container` until after the `main` thread has executed the appropriate `setref`.

Finally, note that there are an arbitrary number of threads and when one is stepped the others remain unchanged. But they must remain "good" with respect to the heap produced by stepping the active thread. How do we disentagle these two?

Our strategy is to check the thread table with respect to a few items of information from the heap: the most recently created `Job` and the counter, mcount and monitor. If an operation changes the heap but does not change those four aspects of it, we get the preservation of the other threads in the thread table "for free." Our `good-heap` predicate will take the entire thread table as an argument but not look (much) at the call stacks (the most actively changing component); the thread table is mainly used to check the correspondence of threads to `Job` objects. The `good-heap` predicate does have to look at the pc of the `main` program (thread `0`) to determine whether the `main` thread has or has not performed the `setref` on the most recently created `Job`.

This discussion is admittedly informal and imprecise but meant to be motivational. There is probably no better way to get this right than to be formal. So we provide the details, starting with `good-thread-table` as defined below.

```
(defun good-thread-table (tt i counter monitor mcount)
  (let* ((thread0 (first tt))
         (frame0 (frame0 (thread-call-stack thread0))))
    (and
     (alistp tt)
     (equal (thread-no thread0) 0)
     (good-thread0 thread0 i)
     (if (and (equal (program frame0) *main*)
              (<= (pc frame0) 1))
         (equal (cdr tt) nil)
         (good-threads 1 (cdr tt) counter monitor mcount
                       (or (equal (program frame0) *setref*)
                           (case (pc frame0)
                             (0 nil)
                             (1 nil)
                             (2 nil)
                             (3 t)
                             (4 t)
                             (5 t)
                             (6 t)
                             (7 t)
                             (8 t)
                             (t nil))))))))
```

When `good-thread-table` is used, `tt` is the thread table of a state, `i` is the heap address of the last item in the heap (one less than the length of the heap), and counter, monitor and mcount are the corresponding values from the heap. The definition starts by letting `thread0` be the first thread in the thread table (thread 0) and letting `frame0` be the topmost frame of the call stack of that thread. The body of `good-thread-table` is a conjunction of four conditions. The first two conjuncts are easy to paraphrase: the thread table is an association list and the thread number of the first thread in it is 0. The third conjunct checks that the thread `0` is running our `main` program, which we discuss further below. The last conjunct is a little complicated: if the pc of the thread `0` points into our `main` program at instructions `0` or `1`, then there are no other threads. Otherwise, the remaining threads satisfy the `good-threads` property. That predicate recognizes threads running `Job`s. The first argument to `good-threads` is the number of the first thread we expect to see, i.e., `1`. The final argument to `good-threads` is a flag, `t` or `nil`, indicating whether the pc in `main` implies that a `Job` has been created but not yet started. (We create a `Job` by executing the `(NEW "Job")` instruction at pc `2` and the corresponding thread should be `UNSCHEDULED` until `main` has executed the `start` method on it at pc `8`. To start a `Job` early risks, for example, that its `objref` has not been properly set to point to our container and might be reset later.)

We must be careful when speaking of "the pc" in a thread (meaning the pc of the topmost frame in its call stack). The pc alone is insufficient to tell us "where we are" in the execution; we must also know the program of the frame. We may intuitively think of thread `0`, say, as always running `main` in this example. But sometimes its top frame is running `setref`, invoked by `main` at `main`'s pc `6`. Thus, if the program in the top frame of thread `0` is `setref` (and thus "the pc of thread `0`" is `0`–`3`), then we know execution has reached pc `6` of `main`.

We describe `good-threads` at length below, but first, here is the definition of `good-thread0`. The variable `i` passed into it is the heap address of the most recently created object in the heap. `Thread` is a "good thread 0" with respect to `i` if the thread number is `0`, the status is `SCHEDULED`, and it has no manifestation in the heap (its `rref` is `nil`). The last conjunct checks the call stack of the thread. Its topmost frame must either be one running our `setref` method, in which case there must be a frame below that appropriately running our `main` method, or else the topmost frame is running our `main` method.

```
(defun good-thread0 (thread i)
  (let ((n (thread-no thread))
        (cs (thread-call-stack thread))
        (status (thread-status thread))
        (rref (thread-rref thread)))
    (and (equal n 0)
         (equal status 'SCHEDULED)
         (equal rref nil)
         (cond ((endp cs) nil)
               ((equal (program (frame0 cs)) *setref*)
                (and (good-setref-frame i (frame0 cs))
                     (not (endp (cdr cs)))
                     (good-main-frame i (frame1 cs) nil)))
               (t (good-main-frame i (frame0 cs) t))))))
```

Here is how we characterize a good frame running our `main` method. Again, `i` here is the heap address of the last object created. `Activep` indicates whether the given frame is on top of the call stack or is covered by another frame.

```
(defun good-main-frame (i frame activep)
  (let* ((pc        (pc frame))
         (locals    (locals frame))
         (stack     (stack frame))
         (program   (program frame))
         (flg       (sync-flg frame))
         (container (binding 'CONTAINER locals))
         (job       (binding 'JOB locals)))
    (and
     (equal program *main*)
     (equal flg 'UNLOCKED)
     (case pc
       (0 (and activep
               (equal stack nil)))
       (1 (and activep
               (equal stack '((REF 0)))))
       (2 (and activep
               (equal stack nil)
               (equal container '(REF 0))))
       (3 (and activep
               (equal stack `((REF ,i)))
               (equal container '(REF 0))))
       (4 (and activep
```

```
            (equal stack nil)
            (equal container '(REF 0))
            (equal job `(REF ,i))))
    (5 (and activep
            (equal stack `(,job))
            (equal container '(REF 0))
            (equal job `(REF ,i))))
    (6 (and activep
            (equal stack `((REF 0) ,job))
            (equal container '(REF 0))
            (equal job `(REF ,i))))
    (7 (and (equal stack nil)
            (equal container '(REF 0))
            (equal job `(REF ,i))))
    (8 (and activep
            (equal stack `(,job))
            (equal container '(REF 0))
            (equal job `(REF ,i))))
    (9 (and activep
            (equal stack nil)
            (equal container '(REF 0))))
    (t nil)))))
```

To be good, a `main` frame must contain our `main` program and be `UNLOCKED` (because our `main` is not a synchronized method). In addition, the pc must be a number between `0` and `9` (and, indeed, must be `7` if the frame is not active, since the `setref` method is invoked at pc `6` from our `main` (and the pc of the caller is incremented before the callee returns) and that is the only method invocation in our `main`). Depending on the pc, the stack and the bindings of the two local variables, `CONTAINER` and `JOB` are constrained. For example, if the pc is `6`, then the stack contains two items: on top is `(REF 0)` (a reference to the `Container` object), next is the value of the variable `JOB`; furthermore, the value of the variable `CONTAINER` must be `(REF 0)` and the value of `JOB` must be `(REF` i`)`, where i is the heap address of the most recently created object.

We similarly describe what a good `setref`, `run`, and `incr` frames look like. For completeness, we show all three below, but only discuss that for `incr` frames.

```
(defun good-setref-frame (i frame)
  (let ((pc      (pc frame))
        (locals  (locals frame))
        (stack   (stack frame))
        (program (program frame))
        (flg     (sync-flg frame)))
    (and
     (equal program *setref*)
     (equal locals `((THIS REF ,i) (R REF 0)))
     (equal flg 'UNLOCKED)
     (case pc
       (0 (equal stack nil))
       (1 (equal stack `((REF ,i))))
       (2 (equal stack `((REF 0) (REF ,i))))
```

```
       (3 t)
       (t nil)))))

(defun good-run-frame (th frame activep monitor mcount)
  (let ((pc (pc frame))
        (locals (locals frame))
        (stack (stack frame))
        (program  (program frame))
        (flg   (sync-flg frame)))
    (and
     (equal program *run*)
     (equal locals `((THIS . (REF ,th))))
     (equal flg 'UNLOCKED)
     (if activep
         (or (equal mcount 0)
             (not (equal monitor th)))
       t)
     (case pc
       (0 (and activep (equal stack nil)))
       (1 (and activep (equal stack `((REF ,th)))))
       (2 (if activep
              (equal stack `((REF ,th)))
            (equal stack nil)))
       (t nil)))))
```

In the `good-incr-frame` predicate below, `th` is the thread number of the thread running this alleged invocation of `incr`. A convenient aspect of the Apprentice system is that the thread number of such a thread is the same as the heap address of the corresponding `Job`, e.g., the self object for thread `3` is `(REF 3)`. The basic structure of this predicate is similar to that of `good-main-frame`. The frame should be running our `incr` method in an unsynchronized way. The case analysis on the pc of the thread is straightforward but bears scrutiny.

```
(defun good-incr-frame (th frame counter monitor mcount)
  (let ((pc (pc frame))
        (locals (locals frame))
        (stack (stack frame))
        (program  (program frame))
        (flg   (sync-flg frame))
        (self `(REF ,th)))
    (and
     (equal program *incr*)
     (equal flg 'UNLOCKED)
     (case pc
       (0 (and (equal locals `((THIS . ,self)))
               (or (equal mcount 0)
                   (not (equal monitor th)))
               (equal stack nil)))
       (1 (and (equal locals `((THIS . ,self)))
               (or (equal mcount 0)
```

```
            (not (equal monitor th)))
        (equal stack `(,self))))
(2 (and (equal locals `((THIS . ,self)))
        (equal stack '((REF 0)))
        (or (equal mcount 0)
            (not (equal monitor th)))))
(3 (and (equal locals
               `((THIS . ,self) (TEMP . (REF 0))))
        (equal stack nil)
        (or (equal mcount 0)
            (not (equal monitor th)))))
(4 (and (equal locals
               `((THIS . ,self) (TEMP . (REF 0))))
        (equal stack '((REF 0)))
        (or (equal mcount 0)
            (not (equal monitor th)))))
(5 (and (equal locals
               `((THIS . ,self) (TEMP . (REF 0))))
        (object-lockedp th monitor mcount)
        (equal stack nil)))
(6 (and (equal locals
               `((THIS . ,self) (TEMP . (REF 0))))
        (object-lockedp th monitor mcount)
        (equal stack `(,self))))
(7 (and (equal locals
               `((THIS . ,self) (TEMP . (REF 0))))
        (object-lockedp th monitor mcount)
        (equal stack '((REF 0)))))
(8 (and (equal locals
               `((THIS . ,self) (TEMP . (REF 0))))
        (object-lockedp th monitor mcount)
        (equal stack `(,self (REF 0)))))
(9 (and (equal locals
               `((THIS . ,self) (TEMP . (REF 0))))
        (object-lockedp th monitor mcount)
        (equal stack '((REF 0) (REF 0)))))
(10 (and (equal locals
                `((THIS . ,self) (TEMP . (REF 0))))
         (object-lockedp th monitor mcount)
         (equal stack `(,counter (REF 0)))))
(11 (and (equal locals
                `((THIS . ,self) (TEMP . (REF 0))))
         (object-lockedp th monitor mcount)
         (equal stack `(1 ,counter (REF 0)))))
(12 (and (equal locals
                `((THIS . ,self) (TEMP . (REF 0))))
         (object-lockedp th monitor mcount)
         (equal stack `(,(+ 1 counter) (REF 0)))))
```

```
    (13 (and (equal locals
                   `((THIS . ,self) (TEMP . (REF 0))))
             (object-lockedp th monitor mcount)
             (equal stack nil)))
    (14 (and (equal locals
                   `((THIS . ,self) (TEMP . (REF 0))))
             (object-lockedp th monitor mcount)
             (equal stack '((REF 0)))))
    (15 (and (equal locals
                   `((THIS . ,self) (TEMP . (REF 0))))
             (or (equal mcount 0)
                 (not (equal monitor th)))
             (equal stack nil)))
    (16 (and (equal locals
                   `((THIS . ,self) (TEMP . (REF 0))))
             (or (equal mcount 0)
                 (not (equal monitor th)))
             (equal stack `(,self))))
    (t nil)))))
```

Recall that `incr` (at pc `0`) pushes its self object on the stack, (`1`) fetches the `objref` field, (`2`) stores the resulting reference in `TEMP`, (`3`) pushes `TEMP` on the stack, and (`4`) enters its critical section by executing `MONITORENTER` on the referenced object. Initially the locals of the frame contain a binding of `THIS` to the self object. After the `STORE` to `TEMP` at pc `2`, two variables are bound in the frame. The stack is tracked similarly. More interestingly, note what is said about the mcount and monitor. Before we execute the `MONITORENTER` at pc `4`, we know that either the mcount is `0` (no thread owns the lock on the `Container`) or else a different thread owns the lock (i.e., `monitor` differs from `th`). After we execute the `MONITORENTER`, the `Container` is locked by us, which is checked by the predicate `object-lockedp`. Technically, for an object to be locked the mcount should be `1` or greater; we defined the predicate below to check equality to `1` simply because in this application we know the mcount is always `0` or `1`.

```
(defun object-lockedp (th monitor mcount)
  (and (equal mcount 1)
       (equal th monitor)))
```

Using `good-incr-frame` and `good-run-frame`, we define what it is to be a good thread other than our thread `0`.

```
(defun good-thread (th scheduled thread counter monitor mcount)
  (let ((n (car thread))
        (cs (cadr thread))
        (status (caddr thread))
        (rref (cadddr thread)))
    (and (equal n th)
         (equal status scheduled)
         (equal rref (list 'REF th))
         (cond ((equal scheduled 'UNSCHEDULED)
                (and (good-run-frame th (car cs) t monitor mcount)
                     (null (cdr cs))))
```

```
               ((endp cs) nil)
               ((equal (program (car cs)) *incr*)
                (and (good-incr-frame th (car cs) counter monitor mcount)
                     (consp (cdr cs))
                     (good-run-frame th (cadr cs) nil monitor mcount)))
               (t (good-run-frame th (car cs) t monitor mcount))))))
```

Here, `th` is the expected thread number, `thread` is the thread table entry we wish to check, `scheduled` is a flag indicating whether we require `thread` to be `SCHEDULED` or `UNSCHEDULED`, and the other three locals are the counter, monitor and mcount retrieved from the heap. To be good, `thread` must have the expected number and status and its manifestation as an instance object must be the object at heap address `th`. In addition, its call stack should either consist of a good `incr` frame covering a good but inactive `run` frame, or else be a good active `run` frame (and the latter must be the case if the thread is still unscheduled).

Finally, we define `good-threads` to check that all the threads in the thread table `tt` are good and all are scheduled except possibly the last.

```
(defun good-threads (i tt counter monitor mcount except-last-flg)
  (cond
   ((endp tt) t)
   (t (and (good-thread i
                        (if (and (endp (cdr tt))
                                 except-last-flg)
                            'unscheduled
                          'scheduled)
                        (car tt)
                        counter monitor mcount)
           (good-threads (+ 1 i)
                         (cdr tt)
                         counter monitor mcount except-last-flg)))))
```

This completes our sketch of the definitions involved with the thread table of a good state. Now we consider the heap.

Good heaps are characterized by the predicate below, which takes as its arguments a thread table `tt` and a heap, `heap`, of some state. We do not put any new restrictions on the thread table here, we are concerned only with the heap, but we need the thread table in order to insure that every `Job` in the heap corresponds to a thread. It helps, when reading this, to think of the thread table as already having been checked for all the good properties. In addition, we know the monitor and mcount (which actually reside in the heap) are always appropriately set. The key property we have to insure about the heap is that the `objref` of every `Job` points to the `Container`. Recall the bad effects that can occur should this be violated.

To be good, a heap must be an association list, i.e., a list of pairs (binding heap addresses to instance objects). Virtually everything else about the heap depends on where we are in the execution of thread `0`. Recall that if thread `0` is running `setref` then we know `main` has executed to pc `6`. If `main` is at pc `0`, the heap must be empty; if `main` is at pc `1`, the heap must contain one object, at address `0`; if `main` is further along, then there must be a correspondence between the threads (after the first) in the thread table and the objects (after the first) in the heap. Each thread must reference the corresponding object and each object must be a `Job` whose `objref` field points to the `Container`. The correspondence between the threads and the heap and the invariants on the objects in the heap are checked by the recursive function `good-objrefs`. The requirement that the `objref` fields point to the `Container` is relaxed for the last object in the heap, whose `objref` field may not have been set yet by the `setref` in `main`. Thus, we pass `good-objrefs` a last argument which indicates whether thread 0's state requires that the objref field of the last heap object be uninitialized. This allows us to preserve most of the good-heap invariant "for free" over steps that do not change this flag.

```
(defun good-heap (tt heap)
  (let* ((thread0 (first tt))
         (n0 (thread-no thread0))
         (frame0 (frame0 (thread-call-stack thread0))))
    (and
     (alistp heap)
     (equal n0 0)
     (cond
      ((equal (program frame0) *setref*)
       (and (consp heap)
            (equal (car (car heap)) 0)
            (consp (cdr heap))
            (good-objrefs (cdr tt) (cdr heap)
                          (not (equal (pc frame0) 3)))))
      ((equal (pc frame0) 0) (equal heap nil))
      (t (and (consp heap)
              (equal (car (car heap)) 0)
              (case (pc frame0)
                (1 (equal (cdr heap) nil))
                (2 (good-objrefs (cdr tt) (cdr heap) nil))
                (3 (and (not (endp (cdr heap)))
                        (good-objrefs (cdr tt) (cdr heap) t)))
                (4 (and (not (endp (cdr heap)))
                        (good-objrefs (cdr tt) (cdr heap) t)))
                (5 (and (not (endp (cdr heap)))
                        (good-objrefs (cdr tt) (cdr heap) t)))
                (6 (and (not (endp (cdr heap)))
                        (good-objrefs (cdr tt) (cdr heap) t)))
                (7 (and (not (endp (cdr heap)))
                        (good-objrefs (cdr tt) (cdr heap) nil)))
                (8 (and (not (endp (cdr heap)))
                        (good-objrefs (cdr tt) (cdr heap) nil)))
                (9 (and (not (endp (cdr heap)))
                        (good-objrefs (cdr tt) (cdr heap) nil)))
                (t nil))))))))
```

Finally, here is the definition of good-objrefs. This function is recursive, because there are an arbitrary number of threads and heap objects. Initially, threads is the list of threads after thread 0 in the thread table, and heap-pairs is the heap alist after the pair binding 0 to the Container. Good-objrefs compares corresponding elements of threads and heap-pairs. In particular, threadi is a thread, with thread number i, and obj is the corresponding instance object with heap address j. The predicate insists that i and j be the same (a convenient aspect of this system), that the thread point to obj as its manifestation in the heap, and that obj be a well-formed Job whose objref points to the (REF 0), the Container, unless the except-last-flg is on and the object is the last one in the heap.

```
(defun good-objrefs (threads heap-pairs except-last-flg)
  (cond
   ((endp heap-pairs) (endp threads))
   ((endp threads) nil)
   (t (let* ((threadi (car threads))
             (i (thread-no threadi))
             (rref (thread-rref threadi))
             (j (car (car heap-pairs)))
             (obj (cdr (car heap-pairs))))
        (and (equal i j)
             (equal rref `(REF ,j))
             (equal obj
                    (if (and (endp (cdr heap-pairs))
                             except-last-flg)
                        '(("Job" ("x" . 0) ("objref" . 0))
                          ("Thread")
                          ("Object" ("monitor" . 0)
                                    ("mcount" . 0)
                                    ("wait-set" . 0)))
                      '(("Job" ("x" . 0) ("objref" . (REF 0)))
                        ("Thread")
                        ("Object" ("monitor" . 0)
                                  ("mcount" . 0)
                                  ("wait-set" . 0)))))
             (good-objrefs (cdr threads)
                           (cdr heap-pairs)
                           except-last-flg))))))
```

Our apparent obsession with the most recently created `Job` object is important: it is the only one whose `objref` field may be changed.

This completes our summary of the `good-state` invariant. We discuss its complexity later, after we describe our proof at a high level.

8 The Proof

The property of `good-state` we wish to prove is

Lemma. `good-state` *invariant*

```
(implies (good-state s)
         (and (good-state (step s th))
              (<= (counter s) (counter (step s th))))).
```

The `good-state` predicate is sufficiently complicated that it is difficult to construct the proof of this lemma, even though it is mathematically straightforward. A good question at this point is: had we left something out would you have noticed? The final proof tree involves about two dozen inductively proved lemmas (mainly because we are dealing with an arbitrary number of threads) and has about 10,000 tips, each of which simplifies to true. Thus, mechanized proof was useful in increasing our confidence that we have got it right.

We describe our mechanically checked proof, not so much to convince the reader of the validity of the conjecture as to explain how we used ACL2 to prove it.

We break the proof into three cases on `th`: `th` is 0, `th` a `SCHEDULED` thread other than 0, and `th` is `UNSCHEDULED`. Each case was proved as a separate lemma and then we instructed ACL2 to use the three together to prove the `good-state` invariant.

The middle case is the most interesting. Our statement of the corresponding lemma is shown below.

```
(implies (and (good-state s)
              (integerp th)
              (<= 1 th)
              (< th (len (heap s)))
              (good-thread th
                           'scheduled
                           (assoc-equal th (thread-table s))
                           (gf "Container" "counter" 0 (heap s))
                           (gf "Object" "monitor" 0 (heap s))
                           (gf "Object" "mcount" 0 (heap s))))
         (and (good-state (step s th))
              (<= (counter s) (counter (step s th)))))
```

The statement is slightly subtle because most of the content of the `good-thread` hypothesis is implied by the `good-state` hypothesis and the conditions on `th`. The only thing the `good-thread` hypothesis adds is that thread `th` is scheduled. However, this formulation of the lemma is slightly easier to manipulate. The `good-state` hypothesis can be expanded to give us the basic properties of the class table, the heap, and thread `0`. The `good-threads` condition, which is buried within `good-state` and which implies most of the `good-thread` hypothesis above, can simply be ignored.

The proof of the lemma follows by expanding the `good-thread` term above, generaing two main cases: either the top frame of thread `th` is a good `incr` frame with a good `run` under it, or else it is a good `run` frame. Similarly expanding each case generates additional cases, e.g., the `incr` frame has pc `0`, or `1`, or 2, etc. In each case, additional restrictions are spelled out, i.e., that the binding `THIS` is `` `(REF ,th) `` and the mcount is either `0` or the monitor is different from `th`. The theorem prover generates this case split automatically by expanding the functions mentioned and normalizing the resulting `IF` expressions.

Once the particulars of a case are determined, we consider the `(step s th)` expression that occurs (twice) in the conclusion of the lemma. The particulars of the case allow us to expand the `step` quite specifically. For example, in the case that the top frame is an `incr` frame with pc `0`, we know that the next instruction is `(LOAD THIS)`, we know the binding of `THIS` (symbolically in terms of `th`), and we know that the stack is empty. The expansion of the `(step s th)` produces a symbolic state. For example, in the case described above, we know that `(step s th)` is a state whose top frame is an `incr` frame on top of the pre-existing `run` frame, the pc of the top frame is `1`, and the stack contains one item, `(REF th)`. That state is computed merely by expanding, symbolically, the definition of `step` and its subfunctions, including `execute-LOAD`.

The resulting symbolic state is used twice in the conclusion, once in the `good-state` conjunct and once in the `counter` expression in the conjunct comparing it with `(counter s)`.

The `good-state` conjunct can be reduced to true, usually by simple symbolic reasoning. For example, one can observe that the call stack is an `incr` frame pushed onto a `run` frame and that the `run` frame is known to satisfy the appropriate conditions by hypothesis. One can then determine that the `incr` frame has pc `1` and that the new symbolic stack is appropriate.

The handling of the `counter` comparison is also routine most of the time: `(counter (step s th))` is the same as `(counter s)` and so is weakly greater.

We are left with a handful of interesting cases. We discuss just one, when the active thread writes to the counter. In the following, let `counter`, `mcount`, and `monitor` be `(gf "Container" "counter" 0 (heap s))`, `(gf "Object" "mcount" 0 (heap s))`, and `(gf "Object" "monitor" 0 (heap s))`, respectively. Let `counter'`, `mcount'`, and `monitor'` be the corresponding expressions for `s` replaced by `(step s th)`.

Suppose the top frame of thread `th` is an `incr` frame with pc `12`. Then the next instruction is the `PUTFIELD` that writes to the counter. The `good-thread` hypothesis tells us that the stack contains `(+ 1 counter)` and `(REF 0)`. Furthermore, we know `(object-lockedp th monitor mcount)`, which tells us that `mcount` is `1` and `monitor` is `th`. `Execute-PUTFIELD` moves `(+ 1 counter)` into the `counter` field of the `Container` at `(REF 0)`. Thus `counter'` is `(+ 1 counter)` and `mcount'` and `monitor'` are `mcount` and `monitor`. From this we can conclude that the conclusion holds.

Notice that the argument just given does not involve mutual exclusion. If the value on top of the stack is one greater than the current value of the counter—as the invariant says it is when we are about to execute the `PUTFIELD`—then xecuting the `PUTFIELD` will increase the counter, whether the current thread has the lock or not!

Where is it important that only one thread has the lock? The mcount and monitor come into play in establishing that the other threads continue to satisfy their invariants when `th` is stepped. Suppose, contrary to mutual exclusion, that another thread is also at pc `12` in a good `incr` frame. Since it must satisfy the invariant, it too must have `(+ 1 counter)` on top of its stack. But this relationship cannot hold after stepping `th`, since the value of the counter changes but the value of the stack in the inactive thread does not. The invariant assures us that two threads cannot both be in their critical sections because for the pc to be `12` in the `incr` frame, mcount must be `1` and the monitor must be `th`; at most one thread can have this property.

Formally, this argument gives rise to a lemma which says that the `good-threads` property is preserved after the active thread changes the heap by incrementing the counter. This is proved inductively, since there are an arbitrary number of other threads.

As we made clear when discussing the ways the system could fail to have the monotonicity property, it is crucial to this proof that we maintain the invariant that the `objref` field of every scheduled `Job` contain `(REF 0)`. The impact of this in our proof is that every time we execute a step that changes the heap we have show that the `good-objrefs` property is preserved. This requires an inductive proof, since there are an arbitrary number of objects in the heap.

Such inductively proved lemmas had to be proved whenever we were establishing preservation of the "good" properties of all unchanged threads and heap objects, when the active thread changes the heap or the scheduled status of a thread. As suggested above, these lemmas all involved either the recursive function `good-threads` or the recursive function `good-objrefs`. The operations in question included the creation of a new thread, the execution of `setref`, the invocation of the `start` method on a thread, the execution of `MONITORENTER` and `MONITOREXIT`, and the writing to the `Container` by the `PUTFIELD` in the `incr` method.

To replay the entire proof takes about 45 minutes on a 750 MHz processor.

9 Conclusion

Our notion of `good-state` includes all the reachable states from `*s0*` and other states as well. For example, in `good-thread0` we require the call stack of thread `0` to contain either a `setref` frame pushed onto a `main` frame, or just a `main` frame. We say nothing about what is under the `main` frame. In fact, one could prove that in the Apprentice system the call stack in thread `0` is of length `1` or `2`. But since execution never returns from the `main` frame, it was not necessary for us to say there were no frames below it.

The Apprentice system has an infinite number of reachable states. In fact, it involves an unbounded number of threads, not to mention the fact that the counter in the heap is unbounded.

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 the complexity of the individual operations on the machine and the dependencies between threads, heap objects, and synchronization. We believe this complexity is inescapable, given the semantics of Java.

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 bytecode expression of the initial state (Figure 4) is less than twice as 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, arithmetic, conditionals, 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.

Our M4 is simpler than the 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 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, especially multi-threaded ones.

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.

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 the proof of the `good-state` invariant into the case where `th` is `0`, the case for other `SCHEDULED` threads, and the case for `UNSCHEDULED` threads. However, there is still too much entanglement, as shown by the reliance of the `good-objrefs` property on the `pc` of thread `0`. The problem cries out for a more elegant solution. One possibility is the transformation into a uniprocessor view as described in [17], 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. We would like to urge our colleagues, however, to preserve the semantics of M4 or, better still, increase the realism still more in the direction of the JVM. That is, we do not consider a solution "better" if the problem solved is less realistic. Key aspects of the problem we solved include that threads 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, and that synchronization is achieved through monitors that need not be respected by all code.

For example, 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. The problem is simpler still if one assumes only a fixed number of pre-existing `Jobs`, but again, we feel that this is an unrealistic assumption given trends in multi-threaded programming.

Finally, we urge others working on this problem not to quit until a mechanically checked proof has been produced, since mechanized checking or even automated verification is an important motivation in adopting formalism in the first place.

10 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 the "defensive JVM" [9]. 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. Pete Manolios has produced a more automatic ACL2 proof of a version of the problem in which there are exactly two pre-existing `Job` threads contending for the `Container`. Pete used our M4 semantics but changed the definition of the `main` method.

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] B. Brock, M. Kaufmann, and J S. Moore. ACL2 theorems about commercial microprocessors. In M. Srivas and A. Camilleri, editors, *Formal Methods in Computer-Aided Design (FMCAD'96)*, pages 275–293. Springer-Verlag, LNCS 1166, November 1996.

[8] Bishop Brock and Warren A. Hunt, Jr. Formally specifying and mechanically verifying programs for the Motorola complex arithmetic processor DSP. In *1997 IEEE International Conference on Computer Design*, pages 31–36. IEEE Computer Society, October 1997.

[9] 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.

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

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

[12] W.A. Hunt and B. Brock. A formal hdl and its use in the fm9001 verification. *Proceedings of the Royal Society*, 1992. to appear April 1992.

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

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

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

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

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

[18] 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.

[19] J S. Moore and G. Porter. An executable formal JVM thread model. *(submitted for publication)*, 2000.

[20] G. L. Steele, Jr. *Common Lisp The Language, Second Edition.* Digital Press, 30 North Avenue, Burlington, MA 01803, 1990.

[21] 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.

[22] 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.