A Commuting Diagram Relating Threaded and Non-threaded JVM Models

George M. Porter
Faculty Adviser: J Strother Moore, Ph.D.

April 16, 2001

Abstract

We establish a commuting diagram that relates two models of the Java Virtual Machine (JVM). The first model, M3, supports much of Java, including classes, objects, and dynamic method resolution. The second model, M4, builds upon M3 by adding threads, monitors, and synchronized methods. We describe a theorem, `Main`, that asserts that running certain "single-threaded" states on M4 is equivalent to transforming those states to the domain of M3, running the transformed state there, and translating the result back to the domain of M4. We define the criteria we use to determine if the resulting states are equivalent, and we define our notion of "single-threaded". We then discuss a few lessons learned during the development of `Main`

1 A Description of M3 and M4

M3 and M4 are both models of the Java Virtual Machine (JVM). They are respectively the third and fourth members of a series of machines approaching the JVM in complexity. (Sun's specification of the JVM can be found in [2]). M3 supports much of the functionality of the JVM, including many bytecodes (such as `ADD`, `IFEQ`, `MUL`, etc). M3 supports classes, with fields and methods. Setting and retrieving the fields of objects in the heap respect inheritance, as does method invocation.

M4 builds upon M3 by supporting multiple threads of execution in a way that is consistent with Sun's specification of the JVM found in [2]. Synchronization between threads is provided via synchronized methods and synchronized blocks. (For a description of Java's thread synchronization mechanisms, see [1]). M4 adds two new bytecodes: `MONITORENTER` and `MONITOREXIT`, which allow the JVM to access monitors located in every Java object in the heap. These monitors will be described in further detail below. For a complete description of M4, see [3].

In M4, a state consists of three components: the thread table, the heap, and the class table. We describe each in turn. When we use the word "table" here

we generally mean a list of pairs in which "keys" (which might be thought of as constituting the left-hand column of the table) are paired with "values" (the right-hand column of the table). Such a table is a map from the keys to the corresponding values.

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

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

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

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

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

```
(defun execute-PUSH (inst th s)
  (modify th s
          :pc (+ 1 (pc (top-frame s th)))
          :stack (push (arg1 inst)
                       (stack (top-frame s th)))))
```

Table 1: execute-PUSH

For those readers curious to see how we define the semantics of such operations in ACL2, see Table 1. It contains the definition of the function `execute-PUSH` which we use to give semantics to the `PUSH` instruction. The instruction `(PUSH 3)` is comparable to `ICONST_3` or `BIPUSH 3` on the JVM.

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

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

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

Create a new call stack, cs', from the call stack of thread th in s by replacing the topmost frame by a new frame in which the program counter has been

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

- The closest method is native: We support only two native methods, `"start"` and `"stop"` from the `"Object"` class. We describe only the first here. In this case, *obj* should include the class `"Thread"` in its superclass chain. The new state constructed by the `"start"` method has the same heap and class table as s. The thread table is changed in two ways. First, the call stack of *th* is replaced by cs' above (stepping over the `INVOKEVIRTUAL`). Second, the thread th' uniquely associated with *obj* is changed so that its scheduled flag is `SCHEDULED`.

- The closest method is a synchronized method: Fetch the contents of the `"monitor"` and `"mcount"` fields in the `"Object"` class of *obj*. If the mcount is 0 or the mcount is non-0 but the monitor is *th*, then we say *obj* is "available" to *th*. If *obj* is available to *th*, then the new state is obtained from s by replacing the call stack with cs'' after setting the `sync-flg` component of the top frame to `LOCKED`, and by replacing the heap of s with a heap in which the `"mcount"` field of the object at *ref* has been incremented by one and the `"monitor"` field has been set to *th*. If, on the other hand, *obj* is unavailable, then the "new" state is s itself. Thus, the thread hangs at the `INVOKEVIRTUAL` instruction until *obj* becomes available. We do not specify the scheduler; instead, our model allows all possible interleavings of thread executions and some thread states (as the one just described) make no change if stepped before progress is possible.

- Otherwise, the new state is obtained from s by replacing the call stack with cs'' after setting the `sync-flg` component of the top frame to `UNLOCKED`.

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

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

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

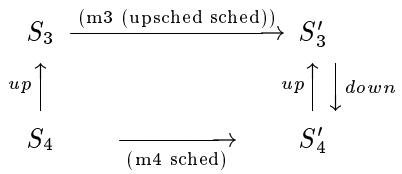
Finally we define `run` to take a "schedule" and a state and return the result of stepping the state according to the given schedule. A schedule is just a list of numbers, indicating which thread is to be stepped next. That is, our model puts no constraints on the JVM thread scheduler; however stepping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

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

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

2 A Commuting Diagram Between M3 and M4

Proving properties of a multi-threaded system is complicated by the fact that the threads can interact in numerous ways. The exact inter-leavings of the threads is not known before runtime, and so all possible interactions must be considered in proving its correctness. It would be beneficial to separate the threads, prove each of them correct independently of the other threads, and then conclude that the resulting multi-threaded state is correct. Often this is impossible, since the threads are intertwined and depend on each other. However, if it were known that the threads did not destructively interfere with each other, then each thread could be proved correct independently of the others. This is the driving force behind our commuting diagram `Main`, which is given below.



The formal expression of this diagram is presented at the end of this section, but for now consider a multi-threaded state S_4. Running the state according

to schedule `sched` results in a new state S_4'. We have defined a function `up` that transforms certain types of "single-threaded" M4 states into M3 states. A predicate `singp` determines if a state is "single-threaded". Currently, "single-threaded" states are those states in which no `Thread` objects have their `start` or `stop` methods invoked, only thread 0 is scheduled, and in which there are no synchronized or native methods. This definition is obviously restrictive, and our hope is that in time the `singp` predicate can be generalized to recognize other states that meet its criteria yet have multiple scheduled threads, for instance. Some of its restrictions have to do with the transformation into an M3 state, since M3 does not support native or synchronized methods, for example.

Given the `up` function, an M4 state is transformed (with loss of information about the non-scheduled threads) into an M3 state, which is then run via the `m3` machine and the component of the schedule that relates to thread 0 (`sched'`). The resulting state can be transformed in a straightforward way back into an M4 state via `down`. That resulting state is the same as S_4' in terms of thread 0, however information about the unscheduled threads is lost.

We now present the definitions of `up`, `down`, `singp`, `almost-equal`, and the commuting diagram `Main`.

```
(defun up (s)
  (m3::make-state (car (binding 0 (m4::thread-table s)))
                  (m4::heap s)
                  (m4::class-table s)))
```

`Up` transforms M4 states into M3 states. Note that the class-tables and heaps are the same in both cases, and that only thread 0 is lifted out of the thread table and set as M3's call-stack.

```
(defun down (s)
  (m4::make-state (bind 0 (list (m3::call-stack s)
                                'JVM::SCHEDULED
                                nil)
                        nil)
                  (m3::heap s)
                  (m3::class-table s)))
```

`Down` is naturally the opposite of `up`, taking M3's call-stack and setting it as the only element of M4's thread-table. The heap and class-table remain unchanged.

```
(defun singp (s)
  (and (at-most-thread0-scheduledp (thread-table s))
       (assoc-equal 0 (thread-table s))
       (equal (caddr (assoc-equal 0 (thread-table s)))
              'JVM::SCHEDULED)
       (no-starts-in-frames (car (binding 0 (thread-table s))))
       (no-starts-in-class-table (class-table s))
```

```
      (no-bytecodex-in-frames
        'JVM::MONITORENTER (car (binding 0 (thread-table s))))
      (no-bytecodex-in-class-table
        'JVM::MONITORENTER (class-table s))
      (no-bytecodex-in-frames
        'JVM::MONITOREXIT (car (binding 0 (thread-table s))))
      (no-bytecodex-in-class-table
        'JVM::MONITOREXIT (class-table s))
      (no-locked-frames-in-frames
       (car (binding 0 (thread-table s))))
      (no-locked-frames-in-class-table (class-table s))
      (no-other-native-methods-in-class-table (class-table s))))
```

`Singp` is the formal definition of a predicate that identifies "single-threaded" M4 states. Let us consider each of its conjuncts. The first three assert that the M4 state has exactly one scheduled thread, namely thread 0. The next two conjuncts ensure that there are no methods named `start` or `stop` invoked on any objects in the state. Following that are four conjuncts that check for the bytecodes `MONITORENTER` and `MONITOREXIT`. These bytecodes are not allowed since M3 does not support them. Following the four checks just described are two conjuncts that check that there are no synchronized methods. Again, since M3 does not support synchronized methods, we cannot allow our M4 state to contain synchronized methods. Lastly, `singp` asserts that there are no native methods in the M4 state, since again M3 does not support native methods.

There is one last definition needed before we can present the commutative diagram. Recall that `up` and `down` are not exact inverses, since `up` loses information (it discards all threads except thread 0). Thus we cannot say that `(down (up s)) = s`. It is the case that `(down (up s))` is the same as `s`, in the sense that the heaps are the same, the class-tables are the same, and thread 0 is the same. The only difference is that `s` may have many unscheduled threads, while `(down (up s))` has only one scheduled thread, thread 0. We formulate a predicate `almost-equal` that captures this meaning of "equal."

```
(defun almost-equal (s4 s4p)
  (and (equal (call-stack 0 s4p)
              (call-stack 0 s4))
       (equal (heap s4) (heap s4p))
       (equal (class-table s4) (class-table s4p))))
```

We now present the formal definition of our commutative diagram, `Main`:

```
(defthm main
  (implies (singp s)
           (almost-equal (down (m3::m3 (up s) (upsched sch)))
                         (m4 s sch)))
  :hints
  (("Goal" :in-theory (disable down up upsched
```

```
                              m3::m3 m4 almost-equal singp)
          :use ((:instance l2 (s (m4 s sch)))))))
```

2.1 Using the Diagram to Port a Theorem from M3 to M4

Imagine that you have a theorem about a property of an M3 state. How can you apply this theorem to an M4 state, given `Main`? In the short discussion to follow, we will describe the process of using `Main` to bring a given theorem over to the domain of M4. As a specific example, we will port a theorem about the Factorial function:

```
(defthm fact-is-correct
  (implies (poised-to-invoke-fact s0 n)
           (equal
            (m3 (fact-clock n) s0)
            (make-state
             (push (make-frame
                    (+ 1 (pc (top-frame s0)))
                    (locals (top-frame s0))
                    (push (acl2::factorial n)
                          (pop (pop (stack (top-frame s0)))))
                    (program (top-frame s0))
                    'JVM::UNLOCKED)
                   (pop (call-stack s0)))
             (heap s0)
             (class-table s0))))
  :hints (("Goal"
           :induct (fact-is-correct-hint s0 n))))
```

This theorem says that if a state `s0` is poised to invoke the fact instance method (recall we do not have `INVOKESTATIC`) on an integer `n`, the result is the same as if we had pushed $n!$ onto the stack (removing the instance object's reference) and incremented the pc. In other words, we are stating that the JVM bytecodes that comprise the `Fact` method correctly carry out the factorial function.

Bringing this theorem to the domain of M4 involves several steps, some of which relate specifically to our factorial example, while others are "generic" and apply to any theorems we might try to port to M4. There are two basic theorems that we must prove related to Factorial:

- First, we must show that the `(poised-to-invoke-fact s0 n)` given as a hypothesis in the above theorem is a result of establishing a similar property `(m4::poised-to-invoke-fact 0 s n)`. We do this via:

  ```
  (defthm condition1
  ```

```
(implies (m4::poised-to-invoke-fact 0 s n)
         (m3::poised-to-invoke-fact (up s) n))
 :hints (("Goal" :in-theory (enable m3::top))))
```

- Once that is established, we must show that the `(fact-clock n)` mentioned in `fact-is-correct` is equal to the schedule that we give our M4 state. This is done via our second condition:

```
(defthm condition2
  (equal (upsched (fact-sched 0 n)) (m3::fact-clock n))
  :hints (("Goal" :in-theory (enable m3::c+-revealed))))
```

Once these two conditions are satisfied, then a series of lemmas that are not specific to Factorial can establish the M4 version of our M3 theorem:

```
(defthm fact-is-correct
  (implies (and (singp s0)
                (poised-to-invoke-fact 0 s0 n))
           (almost-equal
            (m4 (fact-sched 0 n) s0)
            (make-state
             (modify-tt 0
                        (push (make-frame
                               (+ 1 (pc (top-frame 0 s0)))
                               (locals (top-frame 0 s0))
                               (push (acl2::factorial n)
                                 (pop
                                   (pop
                                    (stack (top-frame 0 s0)))))
                               (program (top-frame 0 s0))
                               'jvm::UNLOCKED)
                         (pop (call-stack 0 s0)))
                        'jvm::scheduled
                        (thread-table s0))
             (heap s0)
             (class-table s0))))
  :hints ...)
```

One of those lemmas relates `m3::top-frame` to `m4::top-frame` in such a way that we know that the top frame of an M3 state is equal to the top frame of the 0th thread of the M4 state obtained by using `up`. A similar theorem is made about `call-stack`.

3 The Proof of 'Main'

3.1 Three Lemmas

To prove the commuting diagram, we had to first prove three lemmas, `L1`, `L2`, and `L3`. The proofs of these lemmas will be described in detail below, but first we will present them and try to motivate their role in establishing `Main`. Then, we will prove `Main` given the lemmas. Finally, we will describe their proofs.

Lemma L1.
`(singp s)` $\Rightarrow$ `(m3 (up s) (upsched sch))` $=$ `(up (m4 s sch))`

In many respects `L1` is the hardest of the three lemmas to prove, since it relates M3 and M4 states. `L1` says that transforming a "single-threaded" M4 state `s` to the domain of M3 and running it with the appropriate schedule is the same as running `s` on M4, and then transforming it to the domain of M3.

`Upsched` is an ACL2 function that turns an M4 schedule into an M3 schedule, by stepping the machine the exact number of times `0` appears in `sch`.

Lemma L2.
`(singp s)` $\Rightarrow$ `(down (up s))` $\approx$ `s`

The most straight-forward of the three lemmas, `L2` relates `down` to `up`, in the sense of `almost-equal` ($\approx$).

Lemma L3.
`(singp s)` $\Rightarrow$ `(singp (m4 s sch))`

`L3` simply says that "single-threadedness" is preserved over the machine M4. This lemma is important since without it, we would be unable to reason about the machine after its first step.

3.2 The Derivation of "Main"

Recall the statement of `Main`:

Theorem Main.
`(singp s0)` $\Rightarrow$ `(down (m3 (up s0) (upsched sch)))` $\approx$ `(m4 s0 sch)`

Proof:

Assume

`[1]` `(singp S0)`

By L3 we have

`[2]` `(singp (m4 s0 sch))`

Thus, by L1 and `[1]` we have

```
[3]     (m3 (up s0) (upsched sch)) = (up (m4 s0 sch))
```

Applying `down` to both sides of `[3]` gives

```
[4]     (down (m3 (up s0) (upsched sch))
=
[5]     (down (up (m4 s0 sch)))
```

By L2 and `[2]` we get

```
[5]     (down (up (m4 s0 sch)))
≈
[6]     (m4 s0 sch)
```

Thus, `[4]` ≈ `[6]`.

Q.E.D.

Having derived 'Main', we now turn our attention to each of the three lemmas in turn.

3.3 L1

Recall the definition of lemma `L1`. `L1` asserts that if a state `s` is "single-threaded" according to `singp`, then running `s` on machine `m4` and transforming it to the domain of M3 is the same as transforming `s` to the domain of M3 and running it (with the appropriate schedule). To prove such a statement, we follow the design of the machines themselves. First, note that the machines `m3` and `m4` execute via repeated `step` operations. Thus, we prove a `step` version of lemma `L1`, and then generalize it to the machine via induction. The step lemma we prove is given by:

Lemma. L1-lemma2:

```
(implies (singp s)
         (equal (M3::step3 (up s))
                (up (step4 0 s))))
```

`Step3` and `step4` both act as large "switch" statements, fetching the next bytecode and executing the related `EXECUTE-`*op* function on the state, where *op* is the name of the bytecode. Thus to prove the theorem above, we define and prove lemmas for each bytecode. We then disable the definitions of the `EXECUTE-`*op* functions, and `L1-lemma2` follows from the fact that `step3` and `step4` open up into a case for each bytecode, and we have lemmas already proven for each of these cases. Let us consider the lemma related to the bytecode `ADD`.

Lemma. L1-lemma2-EXECUTE-ADD:

```
(implies (singp s)
         (equal (M3::EXECUTE-ADD inst (up s))
```

```
(up (M4::EXECUTE-ADD inst 0 s))))
```

Most of the bytecodes such as `ADD` and `PUSH` were proved with minimal difficulty. The notable exception was `INVOKEVIRTUAL`, which had to be treated differently, due in part to its sophistication, and in part to its role in the activation of threads in the JVM. Recall that in Java, threads are created via instantiating objects of the `Thread` class (or implementing the `Runnable` interface, although our model does not support interfaces). The threads created by instantiating those objects do not start out scheduled, they must explicitly be started via invoking their `start` methods. Invoking their `stop` methods causes them to become unscheduled. To preserve "single-threadedness", we must prevent threads from changing their scheduled status. Currently, we achieve this by preventing the invocation of any method named `start` or `stop`. Note this restriction in the statement of `L1-lemma-EXECUTE-INVOKEVIRTUAL`:

Lemma. L1-lemma-EXECUTE-INVOKEVIRTUAL:

```
(implies (and (singp s)
              (not (equal (caddr inst) "start"))
              (not (equal (caddr inst) "stop")))
         (equal (M3::EXECUTE-INVOKEVIRTUAL inst (up s))
                (up (M4::EXECUTE-INVOKEVIRTUAL inst 0 s))))
```

To prove the above lemma, we had to establish the fact that `singp` does indeed assert that no `start` or `stop` methods are invoked anywhere in the program (and thus the second and third hypotheses are satisfied). `Singp` uses recursive functions to check that `start` and `stop` methods are not invoked anywhere in the class-table or thread-table. To admit `l1-lemma-EXECUTE-INVOKEVIRTUAL`, we thus had to prove that those recursive functions established that any instruction that `INVOKEVIRTUAL` has the opportunity to execute is not `start` or `stop`. We did this by relating the recursive function `no-starts-in-class-table`, which appears as part of `singp`, to the M4 function `LOOKUP-METHOD-IN--SUPERCLASSES`. Once that relation was established, it was clear that when `singp` holds, `LOOKUP-METHOD-IN-SUPERCLASSES` will not return an instruction that invokes `start` or `stop` methods. Thus we showed that `singp` established hypotheses two and three of `L1-lemma-EXECUTE-INVOKEVIRTUAL`. Once that was completed, the lemma was admitted.

Having proved all of the bytecode-level lemmas, the step lemma given above was admitted easily, via a case analysis. Via induction over the step lemma, `L1-lemma2` was admitted to the theorem prover.

There is one more detail related to `L1` that bears inspection. The function `upsched` transforms an M4 schedule (which is a list of natural numbers) into a natural number that represents the number of times that the M3 state should be stepped. But what of non-zero elements of `sched`? `Upsched` discards them, and so we must show that they do not alter our "single-threaded" state. We admit another lemma stating this very fact:

Lemma. L1-lemma1:

```
(implies (and (singp s)
              (not (equal th 0)))
         (equal (step4 th s) s)))
```

We now turn our attention to the other two lemmas, `L2` and `L3`.

3.4 L2

In the last step of the derivation of `Main`, we took advantage of the fact that if a state `s` is "single-threaded", then it is "equal" (in the `almost-equal` sense) to `(down (up s))`. `L2` establishes this fact.

Lemma. L2:

```
(implies (singp s)
         (almost-equal (down (up s)) s))
```

The proof follows in a straight-forward manner from the definitions of `up`, `down`, and `almost-equal`.

3.5 L3

`L3` establishes the fact that "single-threadedness" is preserved over the machine `m4`. Since `L2` and `L3` have `singp` as a hypothesis, it is important that we establish that `singp` is preserved over `step4`. Otherwise, once we step the machine, we can no longer apply lemmas `L1` and `L2`.

The proof for `L3` is very similar to the proof of `L1`. First, we must prove the step-version of lemma:

Lemma. L3-lemma:

```
(implies (singp s)
         (singp (step4 th s)))
```

From `L3-lemma`, we use induction to establish `L3`. We prove the step-version of `L3` as we did with `L1`, namely we prove the property over each of the bytecodes. By then disabling the definitions of each of the `EXECUTE-`*op* functions, L3-lemma opens into a case for each bytecode, for which we have our bytecode-level proofs completed. An example of a straightforward bytecode that we prove is given by `ADD`:

Lemma. L3-lemma-EXECUTE-ADD:

```
(implies (singp s)
         (singp (execute-add inst 0 s))))
```

As before, we had to treat `INVOKEVIRTUAL` specially. In a manner similar to `L1`, we related the recursive functions given in `singp` to the method that `INVOKEVIRTUAL` is called to act on. After establishing that `singp` ensures

that INVOKEVIRTUAL only gets "well formed" methods, we were able to admit L3-lemma-EXECUTE-INVOKEVIRTUAL. Once this was accomplished, we could establish L3-lemma, and thus L3.

Proving the three lemmas led directly to the proof of Main, following the proof outline given above.

4 Discussion

The previous section presented the proof of Main. We now discuss several problems we encountered in trying to prove Main, and show how their solutions made their way into the final statement of Main and its proof. We will also include unexpected results that only came to light during the proof attempt.

4.1 Packages and Identifiers

One of the most surprising and interesting behaviors we discovered while proving Main relates to the way ACL2 handles packages. We defined the machines M3 and M4 in packages M3 and M4, respectively. As the proof of Main developed, we realized that certain identifiers in our states were not global, but rather grounded in a certain package. Consider for example identifiers such as LOAD, PUSH, THIS, and LOCKED. In reality, in the M4 model these are M4::LOAD, M4::PUSH, M4::THIS, and M4::LOCKED. But in the M3 model, the identifiers are represented by M3::LOAD, M3::PUSH, M3::THIS, and M3::LOCKED. When we tried to draw a correspondence between an M3 program and an M4 program that look identical, we realized that in fact they are not! Consider the following M3 program fragment:

```
(NEW "Alpha")
(PUSH 1)
(PUSH 2)
(ADD)
(STORE A)
(LOAD A)
```

Now the program is really represented by:

```
(M3::NEW "Alpha")
(M3::PUSH 1)
(M3::PUSH 2)
(M3::ADD)
(M3::STORE A)
(M3::LOAD A)
```

But in M4 each of the instructions are in the M4 package. To continue this idea, we realized that in a given program when we refer to THIS in the M3 package, we mean for that to be the same THIS in the M4 package. To correct

this problem, we define a new package `JVM`. We always refer to identifiers in the `JVM` package, so that from either M3 or M4 we can refer to the same logical object, for example `JVM::THIS`.

We had not considered this behavior before starting on the proof, and in fact did not realize it until deep into the proof attempt. This is one of the benefits to using formal proof techniques—they are often a good way to discover the unknown behaviors of a given formal system.

A perfect example of this fact relates to a careless mistake we made during the definition of M4. In M3, there is a `DUP` bytecode, which duplicates the item on the top of the stack of the topmost frame in the call-stack. By our omission, there was no such operation in M4, and so ACL2 failed on the proof attempt, trying to relate an M4 state to an M3 state in which the `DUP` bytecode had been executed. Since M4 had no `DUP` bytecode, this relation was doomed to failure. Once we added `DUP` to M4, the proof went through.

Some would suggest that if we had been more careful, this failure would have been avoided. They are right. However, attempting to live a life free of mistakes is hopeless, and so a technique like automatic proof checking is a valuable aid in finding and correcting mistakes.

4.2 Thread Scheduling

When we first started thinking about `Main`, we realized that our multithreaded states could have at most one scheduled thread. This restriction was the first to make up `singp`. It wasn't until we failed to prove `L1` that we realized that we had to make a stronger statement. To admit `Main`, we had to make sure that exactly one thread is scheduled in a given state. Why is this restriction necessary?

Recall that stepping an unscheduled thread in M4 has no effect (it is a "no-op"). Note that M3 behaves differently, in that each step in fact modifies the machine. Consider an M4 state in which *no* thread is scheduled. Stepping any of its threads leave the state unchanged, since `m4` will not step unscheduled threads. On the other hand, transforming the state to the domain of M3 and stepping it there *will* modify the state, since all scheduling information is lost during the translation.

So for this reason we had to tighten our definition of "single-threaded". This addition can be seen in the third conjunct of `singp`.

4.3 Omissions from singp

There were three additional refinements made to `singp` during the development of `Main`. Each of these refinements prevented features of M4 from interfering with the translation of multithreaded states to the domain of M3.

First, we had to prevent "single-threaded" states from containing native methods (other than `start` and `stop`, which are explicitly forbidden). M3 does not support native methods, and so it is impossible to translate states that depend on native methods to the domain of M3.

Secondly, we disallowed synchronized methods from our M4 states. Remember that when a synchronized method is invoked, the JVM automatically obtains locks on the instance object. M3 does not support monitors, and so it would be unable to obtain these locks. One idea would be to make synchronized method invocation behave just like non-synchronized method invocation on M3. If this were the case then invoking synchronized methods on M4 would change the heap (in terms of the monitor in the instance object), however there would be no analogous modification to the M3 heap. We feel that keeping the synchronized methods would thus be unsatisfactory, since the heap would be treated differently between M3 and M4.

The last omission is related to the second, namely we overlooked `MONITORENTER` and `MONITOREXIT`. Again, we could have made these bytecodes no-ops on M3, however we would be in a situation where these bytecodes modify the heap in M4, yet leave the heap unchanged in M3. In reality `MONITORENTER` and `MONITOREXIT` change the heaps in ways that the threads cannot detect. (In Java there is no way to determine the status of a given monitor–they are opaque objects). As mentioned in the following section, it might be beneficial to introduce a more powerful `up` that can modify M4 heaps into forms that work better with M3.

Each of these refinements tightened our notion of "single-threadedness", and once they were introduced, we were able to establish `Main`.

4.4 A Faulty call-stack-rref

During the development of `Main`, we decided to alter the order of arguments to M4. During this process, we overlooked `call-stack-rref`, a function used by M4. This function, given a thread and a thread-table, returns a reference to the thread's associated object in the heap. Interestingly, the faulty `call-stack-rref` did not prevent M4 or `Main` from certifying properly. The presence of the bug was only discovered during an attempt to find a state S_0 that satisfies both of the hypotheses of the theorem about Factorial. Before this attempt was made, the hypotheses were in fact unsatisfiable, and thus the theorem was vacuous! Lucky for us the faulty function did not cause other problems in the certification process, and once the bug was discovered and repaired we were able to recertify all of our books. We now know that there is at least one state (namely S_0) that satisfies the hypotheses of our theorem about Factorial.

5 Suggestions for future work

During the development of `Main`, we thought of several opportunities to improve our work. We present them in the hopes that our work will become more applicable to "real-world" applications involving multithreaded JVM programs.

An obvious place to begin improving our theorem is in the definition of `singp`. Recall that `singp` acts as a predicate that identifies "single-threaded" states. Currently, the methods used to make that determination are somewhat

crude. We prevent our program from invoking methods called `start` or `stop`. We do not allow synchronized methods. What `singp` is really looking for are states in which the threads do not interfere with each other. Currently, there are states that meet that criteria that `singp` rejects. For example, consider a state in which several threads are all scheduled, yet none of them modify the heap. It would be profitable to increase the sophistication in which `singp` decides if a given state is "single-threaded". This would allow more states to pass through `singp`.

Another avenue of improvement involves the translation of states between M3 and M4. Recall that currently, `up` and `down` do not modify the class-table or heap of the state they are transforming. Furthermore, they modify the thread-table only in that `up` lifts out thread 0, and `down` constructs a thread-table from an M3 call-stack. It would be advantageous to increase the sophistication of these two functions. Perhaps `up` could modify the heap in a way that removes all monitors from the objects. Since we already know that the state in question is "single-threaded", we would not have to worry about contention of locks. If we had such an `up`, then we would be able to add `MONITORENTER` and `MONITOREXIT` to M3, making them no-ops. Thus we could then remove `singp`'s restrictions about states that contain those bytecodes.

These two additions to our model and our proof would certainly improve the utility of `Main`. We hope that, in time, they will be added to `Main` and the JVM model.

6 Acknowledgments

I would like to thank those who have helped and supported me while learning ACL2, working on this project, and writing this paper. I am especially grateful to J Strother Moore, for spending countless hours with me on M4 and this proof, for developing the theorems that allow `Main` to port theorems from M3 to M4, and for supporting me in my educational goals. I would also like to thank Bob Boyer for his advice and aid, and for reading this paper. Finally, I would like to thank my family for their 22 years of constant love and support.

References

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

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

[3] J S. Moore and G. Porter. An executable formal JVM thread model. In *Proceedings of the USENIX Java Virtual Machine Research and Technology Symposium*, April 2001.

A M3

```
; Abstract Machine 3

; M3 is M2 except that we now have the heap, classes and invokevirtual.
; $Id: m3.lisp,v 1.13 2001/04/10 03:45:41 george Exp $

#|
(defpkg "JVM" '(nil t))

(defpkg "M3"
  (set-difference-equal
   (union-eq '(ASSOC-EQUAL LEN NTH ZP SYNTAXP
                           QUOTEP FIX NFIX E0-ORDINALP E0-ORD-<)
             (union-eq *acl2-exports*
                       *common-lisp-symbols-from-main-lisp-package*))
   '(PC PROGRAM PUSH POP REVERSE STEP ++)))

(certify-book "m3" 2)

|#

(in-package "M3")

; Utilities

; Stacks
(defun push (obj stack) (cons obj stack))
(defun top (stack) (car stack))
(defun pop (stack) (cdr stack))

(defthm stacks
  (and (equal (top (push x s)) x)
       (equal (pop (push x s)) s)))

(in-theory (disable push top pop))

; Alists
(defun bound? (x alist) (assoc-equal x alist))

(defun bind (x y alist)
  (cond ((endp alist) (list (cons x y)))
        ((equal x (car (car alist)))
         (cons (cons x y) (cdr alist)))
        (t (cons (car alist) (bind x y (cdr alist))))))
```

```
(defun binding (x alist) (cdr (assoc-equal x alist)))

; Instructions
(defun op-code (inst) (car inst))
(defun arg1 (inst) (car (cdr inst)))
(defun arg2 (inst) (car (cdr (cdr inst))))
(defun arg3 (inst) (car (cdr (cdr (cdr inst)))))

; M3 States
(defun make-state (call-stack heap class-table)
  (list call-stack heap class-table))

(defun call-stack (s) (nth 0 s))
(defun heap (s) (nth 1 s))
(defun class-table (s) (nth 2 s))

(defthm states
  (and (equal (call-stack (make-state cs h c)) cs)
       (equal (heap (make-state cs h c)) h)
       (equal (class-table (make-state cs h c)) c)))

(in-theory (disable make-state call-stack heap class-table))

; Frames
(defun top-frame (s) (top (call-stack s)))

(defun pc (frame) (nth 0 frame))
(defun locals (frame) (nth 1 frame))
(defun stack (frame) (nth 2 frame))
(defun program (frame) (nth 3 frame))
(defun sync-flg (frame) (nth 4 frame))

(defun make-frame (pc locals stack program sync-flg)
  (list pc locals stack program sync-flg))

(defthm frames
  (and
   (equal (pc (make-frame pc l s prog sync-flg)) pc)
   (equal (locals (make-frame pc l s prog sync-flg)) l)
   (equal (stack (make-frame pc l s prog sync-flg)) s)
   (equal (program (make-frame pc l s prog sync-flg)) prog)
   (equal (sync-flg (make-frame pc l s prog sync-flg)) sync-flg)))

(in-theory (disable make-frame pc locals stack program sync-flg))

(defun next-inst (s)
```

```
  (nth (pc (top-frame s)) (program (top-frame s))))

; Class Declarations
(defun make-class-decl (name superclasses fields methods)
  (list name superclasses fields methods))

(defun class-decl-name (dcl)
  (nth 0 dcl))
(defun class-decl-superclasses (dcl)
  (nth 1 dcl))
(defun class-decl-fields (dcl)
  (nth 2 dcl))
(defun class-decl-methods (dcl)
  (nth 3 dcl))

; This is a base set of classes that are 'built in' to M3 states
(defun base-class-def ()
   (list (make-class-decl "Object"
                          nil
                          '("monitor" "mcount" "wait-set")
                          nil)
         (make-class-decl "Thread"
                          '("Object")
                          nil
                          '(("run" () nil
                             (jvm::return))
                            ("start" () nil ())
                            ("stop" () nil ())))))

(defun make-class-def (list-of-class-decls)
   (append (base-class-def) list-of-class-decls))

(defun method-name (m)
  (nth 0 m))
(defun method-formals (m)
  (nth 1 m))
(defun method-sync (m)
  (nth 2 m))
(defun method-program (m)
  (cdddr m))

; The Standard Modify

(defun suppliedp (key args)
  (cond ((endp args) nil)
        ((equal key (car args)) t)
```

```
            (t (suppliedp key (cdr args)))))

    (defun actual (key args)
      (cond ((endp args) nil)
            ((equal key (car args)) (cadr args))
            (t (actual key (cdr args)))))

    (defmacro modify (s &rest args)
      (list 'make-state
            (cond ((suppliedp :call-stack args)
                   (actual :call-stack args))
                  ((or (suppliedp :pc args)
                       (suppliedp :locals args)
                       (suppliedp :stack args)
                       (suppliedp :program args)
                       (suppliedp :sync-flg args))
                   (list 'push
                         (list 'make-frame
                               (if (suppliedp :pc args)
                                   (actual :pc args)
                                 (list 'pc (list 'top-frame s)))
                               (if (suppliedp :locals args)
                                   (actual :locals args)
                                 (list 'locals (list 'top-frame s)))
                               (if (suppliedp :stack args)
                                   (actual :stack args)
                                 (list 'stack (list 'top-frame s)))
                               (if (suppliedp :program args)
                                   (actual :program args)
                                 (list 'program (list 'top-frame s)))
                               (if (suppliedp :sync-flg args)
                                   (actual :sync-flg args)
                                 (list 'sync-flg (list 'top-frame s))))
                         (list 'pop (list 'call-stack s))))
                  (t (list 'call-stack s)))
            (if (suppliedp :heap args)
                (actual :heap args)
              (list 'heap s))
            (if (suppliedp :class-table args)
                (actual :class-table args)
              (list 'class-table s))))

    ; (PUSH const)
    (defun execute-PUSH (inst s)
      (modify s
              :pc (+ 1 (pc (top-frame s)))
```

```
              :stack (push (arg1 inst) (stack (top-frame s)))))

; (POP)
(defun execute-POP (inst s)
  (declare (ignore inst))
  (modify s
          :pc (+ 1 (pc (top-frame s)))
          :stack (pop (stack (top-frame s)))))

; (LOAD var)
(defun execute-LOAD (inst s)
  (modify s
          :pc (+ 1 (pc (top-frame s)))
          :stack (push (binding (arg1 inst)
                                (locals (top-frame s)))
                       (stack (top-frame s)))))

; (STORE var)
(defun execute-STORE (inst s)
  (modify s
          :pc (+ 1 (pc (top-frame s)))
          :locals (bind (arg1 inst)
                        (top (stack (top-frame s)))
                        (locals (top-frame s)))
          :stack (pop (stack (top-frame s)))))

; (DUP)
(defun execute-DUP (inst s)
  (declare (ignore inst))
  (modify s
          :pc (+ 1 (pc (top-frame s)))
          :stack (push (top (stack (top-frame s))) (stack (top-frame s)))))

; (ADD)
(defun execute-ADD (inst s)
  (declare (ignore inst))
  (modify s
          :pc (+ 1 (pc (top-frame s)))
          :stack (push (+ (top (pop (stack (top-frame s))))
                          (top (stack (top-frame s))))
                       (pop (pop (stack (top-frame s)))))))

; (SUB)
(defun execute-SUB (inst s)
  (declare (ignore inst))
  (modify s
```

```
              :pc (+ 1 (pc (top-frame s)))
230           :stack (push (- (top (pop (stack (top-frame s))))
                              (top (stack (top-frame s))))
                            (pop (pop (stack (top-frame s)))))))

    ; (MUL)
235 (defun execute-MUL (inst s)
      (declare (ignore inst))
      (modify s
              :pc (+ 1 (pc (top-frame s)))
              :stack (push (* (top (pop (stack (top-frame s))))
240                           (top (stack (top-frame s))))
                            (pop (pop (stack (top-frame s)))))))

    ; (GOTO n)
    (defun execute-GOTO (inst s)
245   (modify s
              :pc (+ (arg1 inst) (pc (top-frame s)))))

    ; (IFEQ n)
    (defun execute-IFEQ (inst s)
250   (modify s
              :pc (if (equal (top (stack (top-frame s))) 0)
                      (+ (arg1 inst) (pc (top-frame s)))
                    (+ 1 (pc (top-frame s))))
              :stack (pop (stack (top-frame s)))))
255
    ; (IFNE n)
    (defun execute-IFNE (inst s)
      (modify s
              :pc (if (equal (top (stack (top-frame s))) 0)
260                   (+ 1 (pc (top-frame s)))
                    (+ (arg1 inst) (pc (top-frame s))))
              :stack (pop (stack (top-frame s)))))

    ; (IFGT n)
265 (defun execute-IFGT (inst s)
      (modify s
              :pc (if (> (top (stack (top-frame s))) 0)
                      (+ (arg1 inst) (pc (top-frame s)))
                    (+ 1 (pc (top-frame s))))
270           :stack (pop (stack (top-frame s)))))

    ; (IFLT n)
    (defun execute-IFLT (inst s)
      (modify s
```

```
            :pc (if (< (top (stack (top-frame s))) 0)
                    (+ (arg1 inst) (pc (top-frame s)))
                  (+ 1 (pc (top-frame s))))
            :stack (pop (stack (top-frame s)))))

; (NEW class)
(defun build-class-field-bindings (field-names)
  (if (endp field-names)
      nil
    (cons (cons (car field-names) 0)
          (build-class-field-bindings (cdr field-names)))))

(defun build-immediate-instance-data (class-name class-table)
  (cons class-name
        (build-class-field-bindings
         (class-decl-fields
          (bound? class-name class-table)))))

(defun build-an-instance (class-names class-table)
  (if (endp class-names)
      nil
    (cons (build-immediate-instance-data (car class-names) class-table)
          (build-an-instance (cdr class-names) class-table))))

(defun execute-NEW (inst s)
  (let* ((class-name (arg1 inst))
         (class-table (class-table s))
         (new-object (build-an-instance
                      (cons class-name
                            (class-decl-superclasses
                             (bound? class-name class-table)))
                      class-table))
         (new-address (len (heap s))))
    (modify s
            :pc (+ 1 (pc (top-frame s)))
            :stack (push (list 'JVM::REF new-address)
                         (stack (top-frame s)))
            :heap (bind new-address new-object (heap s)))))

; (GETFIELD class field)
(defun deref (ref heap)
  (binding (cadr ref) heap))

(defun field-value (class-name field-name instance)
  (binding field-name
           (binding class-name instance)))
```

```
(defun execute-GETFIELD (inst s)
  (let* ((class-name (arg1 inst))
         (field-name (arg2 inst))
         (instance (deref (top (stack (top-frame s))) (heap s)))
         (field-value (field-value class-name field-name instance)))
    (modify s
            :pc (+ 1 (pc (top-frame s)))
            :stack (push field-value
                         (pop (stack (top-frame s)))))))

; (PUTFIELD class field)
(defun set-instance-field (class-name field-name value instance)
  (bind class-name
        (bind field-name value
              (binding class-name instance))
        instance))

(defun execute-PUTFIELD (inst s)
  (let* ((class-name (arg1 inst))
         (field-name (arg2 inst))
         (value (top (stack (top-frame s))))
         (instance (deref (top (pop (stack (top-frame s)))) (heap s)))
         (address (cadr (top (pop (stack (top-frame s)))))))
    (modify s
            :pc (+ 1 (pc (top-frame s)))
            :stack (pop (pop (stack (top-frame s))))
            :heap (bind address
                        (set-instance-field class-name
                                            field-name
                                            value
                                            instance)
                        (heap s)))))

; (INVOKEVIRTUAL class method n)
(defun reverse (lst)
  (if (consp lst)
      (append (reverse (cdr lst)) (list (car lst)))
    nil))

(defun bind-formals (rformals stack)
  (if (endp rformals)
      nil
    (cons (cons (car rformals) (top stack))
          (bind-formals (cdr rformals) (pop stack)))))
```

```
(defun popn (n stack)
  (if (zp n)
      stack
    (popn (- n 1) (pop stack))))

(defun class-name-of-ref (ref heap)
  (car (car (deref ref heap))))

(defun lookup-method-in-superclasses (name classes class-table)
  (cond ((endp classes) nil)
        (t (let* ((class-name (car classes))
                  (class-decl (bound? class-name class-table))
                  (method (bound? name (class-decl-methods class-decl))))
             (if method
                 method
               (lookup-method-in-superclasses name (cdr classes)
                                              class-table))))))

(defun lookup-method (name class-name class-table)
  (lookup-method-in-superclasses name
                                 (cons class-name
                                       (class-decl-superclasses
                                        (bound? class-name class-table)))
                                 class-table))

(defun execute-INVOKEVIRTUAL (inst s)
  (let* ((method-name (arg2 inst))
         (nformals (arg3 inst))
         (obj-ref (top (popn nformals (stack (top-frame s)))))
         (obj-class-name (class-name-of-ref obj-ref (heap s)))
         (closest-method
          (lookup-method method-name
                         obj-class-name
                         (class-table s)))
         (vars (cons 'JVM::THIS (method-formals closest-method)))
         (prog (method-program closest-method))
         (s1 (modify s
                     :pc (+ 1 (pc (top-frame s)))
                     :stack (popn (len vars) (stack (top-frame s))))))
    (modify s1
            :call-stack
            (push (make-frame 0
                              (reverse
                               (bind-formals (reverse vars)
                                             (stack (top-frame s))))
                              nil
```

```
                                  prog
                                  'JVM::UNLOCKED)
415                    (call-stack s1)))))

     ; (XRETURN)
     (defun execute-XRETURN (inst s)
       (declare (ignore inst))
420    (let ((val (top (stack (top-frame s))))
             (s1 (modify s
                         :call-stack (pop (call-stack s)))))
         (modify s1
                 :stack (push val (stack (top-frame s1))))))
425
     ; (RETURN)
     (defun execute-RETURN (inst s)
       (declare (ignore inst))
       (modify s
430             :call-stack (pop (call-stack s))))

     ; The M3 Run Level
     (defun do-inst (inst s)
       (case (op-code inst)
435      (JVM::PUSH            (execute-PUSH          inst s))
         (JVM::POP             (execute-POP           inst s))
         (JVM::LOAD            (execute-LOAD          inst s))
         (JVM::STORE           (execute-STORE         inst s))
         (JVM::DUP             (execute-DUP           inst s))
440      (JVM::ADD             (execute-ADD           inst s))
         (JVM::SUB             (execute-SUB           inst s))
         (JVM::MUL             (execute-MUL           inst s))
         (JVM::GOTO            (execute-GOTO          inst s))
         (JVM::IFEQ            (execute-IFEQ          inst s))
445      (JVM::IFNE            (execute-IFNE          inst s))
         (JVM::IFGT            (execute-IFGT          inst s))
         (JVM::IFLT            (execute-IFLT          inst s))
         (JVM::NEW             (execute-NEW           inst s))
         (JVM::GETFIELD        (execute-GETFIELD      inst s))
450      (JVM::PUTFIELD        (execute-PUTFIELD      inst s))
         (JVM::INVOKEVIRTUAL   (execute-INVOKEVIRTUAL inst s))
         (JVM::XRETURN         (execute-XRETURN       inst s))
         (JVM::RETURN          (execute-RETURN        inst s))
         (otherwise       s)))
455
     (defun step3 (s)
       (do-inst (next-inst s) s))
```

```
(defun m3 (n s)
  (if (zp n)
      s
    (m3 (- n 1) (step3 s))))

; Compile it all.
; (comp t)

; The idea is that a JVM programmer will do something like
; (in-package "JVM")
; and then he will type a program that looks like:

; ("fact" (n) nil (load this) ...)

; and really he has created

; ("fact" (n) nil (jvm::load jvm::this) ...)
```

B M4

```
; M4.lisp
; J Strother Moore <moore@cs.utexas.edu>
; George Porter <george@cs.utexas.edu>
; $Id: m4.lisp,v 1.11 2001/04/10 03:45:41 george Exp $

#|
(defpkg "JVM" '(nil t))

(DEFPKG "M4"
  (set-difference-equal
   (union-eq '(ASSOC-EQUAL LEN NTH ZP SYNTAXP
                           QUOTEP FIX NFIX E0-ORDINALP E0-ORD-<)
             (union-eq *acl2-exports*
                       *common-lisp-symbols-from-main-lisp-package*))
   '(PC PROGRAM PUSH POP REVERSE STEP ++)))

(certify-book "m4" 2)

|#

; Notes:
; Do JVM objects have an mcount field?  Can the user set them with
; (putfield "Object" "mcount")?  This machine allows that, which
; can screw up monitors.


; -----------------------------------------------------------------------------
; Abstract Machine 4  - by George Porter and J Moore
; $Id: m4.lisp,v 1.11 2001/04/10 03:45:41 george Exp $

(in-package "M4")

; -----------------------------------------------------------------------------
; Utilities

(defun push (obj stack) (cons obj stack))
(defun top (stack) (car stack))
(defun pop (stack) (cdr stack))

#|
(defthm stacks
  (and (equal (top (push x s)) x)
       (equal (pop (push x s)) s)))
```

```
(in-theory (disable push top pop))

; Imported from ACL2.

(defun assoc-equal (x alist)
  (cond ((endp alist) nil)
        ((equal x (car (car alist)))
         (car alist))
        (t (assoc-equal x (cdr alist)))))
|#

(defun bound? (x alist) (assoc-equal x alist))

(defun bind (x y alist)
  (cond ((endp alist) (list (cons x y)))
        ((equal x (car (car alist)))
         (cons (cons x y) (cdr alist)))
        (t (cons (car alist) (bind x y (cdr alist))))))

(defun binding (x alist) (cdr (assoc-equal x alist)))

(defun op-code (inst) (car inst))
(defun arg1 (inst) (car (cdr inst)))
(defun arg2 (inst) (car (cdr (cdr inst))))
(defun arg3 (inst) (car (cdr (cdr (cdr inst)))))

; Imported from ACL2
#|
(defun nth (i lst)
  (if (zp i)
      (car lst)
    (nth (- i 1) (cdr lst))))

(defun zp (i)
  (if (integerp i) (<= i 0) t))
|#

(defun reverse (x)
  (if (consp x)
      (append (reverse (cdr x)) (list (car x)))
    nil))


; -----------------------------------------------------------------------------
; States

```

```
(defun make-state (thread-table heap class-table)
  (list thread-table heap class-table))
(defun thread-table (s) (nth 0 s))
(defun heap        (s) (nth 1 s))
(defun class-table (s) (nth 2 s))

(defthm states
  (and (equal (thread-table (make-state tt h c)) tt)
       (equal (heap (make-state tt h c)) h)
       (equal (class-table (make-state tt h c)) c)))

(defthm states2
  (and (equal (thread-table (list tt h c)) tt)
       (equal (heap (list tt h c)) h)
       (equal (class-table (list tt h c)) c)))

(in-theory (disable make-state thread-table heap class-table))

(defun call-stack (th s)
  (car (binding th (thread-table s))))

(defun call-stack-status (th s)
  (cadr (binding th (thread-table s))))

(defun call-stack-rref (th tt)
  (caddr (binding th tt)))


; ------------------------------------------------------------------------
; Class Declarations and the Class Table

; The class table of a state is an alist.  Each entry in a class table is
; a "class declaration" and is of the form

;   (class-name super-class-names fields defs)

; Note that the definition below of the Thread class includes a 'run' method,
;  which most applications will override.  The definition is consistent
;  with the default run method provided by the Thread class [O'reily page xxx]

(defun make-class-decl (name superclasses fields methods)
  (list name superclasses fields methods))

(defun class-decl-name (dcl)
  (nth 0 dcl))
(defun class-decl-superclasses (dcl)
```

```
  (nth 1 dcl))
(defun class-decl-fields (dcl)
  (nth 2 dcl))
(defun class-decl-methods (dcl)
  (nth 3 dcl))

(defun base-class-def ()
   (list (make-class-decl "Object"
                          nil
                          '("monitor" "mcount" "wait-set")
                          nil)
         (make-class-decl "Thread"
                          '("Object")
                          nil
                          '(("run" () nil
                             (JVM::RETURN))
                            ("start" () nil ())
                            ("stop" () nil ())))))

(defun make-class-def (list-of-class-decls)
   (append (base-class-def) list-of-class-decls))

; ------------------------------------------------------------------------
; Thread Tables
;
; A "thread table" might be used to represent threads in m4.  It consists of
;  a reference, a call stack, a flag to indicate whether its call-stack
;  should be stepped by the scheduler, and a ref to the original object
;  in the heap.
;
; Thread table:
; ((n   . (call-stack flag reverse-ref))
;  (n+1 . (call-stack flag reverse-ref)))
;
; The flags 'JMV::SCHEDULED and 'JVM::UNSCHEDULED coorespond to two of the four states
;  threads can be in (according to [O'Reily]).  For our model, this will
;  suffice.

(defun make-tt (call-stack)
  (bind 0 (list call-stack 'JVM::SCHEDULED nil) nil))

(defun modify-tt (th call-stack status tt)
  (bind th (list call-stack status (call-stack-rref th tt)) tt))

(defun addto-tt (call-stack status heapRef tt)
  (bind (len tt) (list call-stack status heapRef) tt))
```

```
(defun mod-thread-scheduling (th sched tt)
  (let* ((thrd (binding th tt))
         (oldcs  (car thrd))
         (oldhr  (caddr thrd))
         (newTH  (list oldcs sched oldhr)))
        (bind th newTH tt)))

(defun schedule-thread (th tt)
  (mod-thread-scheduling th 'JVM::SCHEDULED tt))

(defun unschedule-thread (th tt)
  (mod-thread-scheduling th 'JVM::UNSCHEDULED tt))

(defun rrefToThread (ref tt)
  (cond ((endp tt) nil)
        ((equal ref (cadddr (car tt))) (caar tt))
        (t (rrefToThread ref (cdr tt)))))

; ------------------------------------------------------------------------
; Helper function for determining if an object is a 'Thread' object

(defun in-list (item list)
  (cond ((endp list) nil)
        ((equal item (car list)) t)
        (t (in-list item (cdr list)))))

(defun isThreadObject? (class-name class-table)
  (let* ((class (bound? class-name class-table))
         (psupers (class-decl-superclasses class))
         (supers (cons class-name psupers)))
       (or (in-list "Thread" supers)
           (in-list "ThreadGroup" supers))))

; ------------------------------------------------------------------------
; Helper functions for locking and unlocking objects

; lock-object and unlock-object will obtain a lock on an instance
;  of an object, using th as the locking id (a thread owns a lock).  If th
;  already has a lock on an object, then the mcount of the object is
;  incremented.  Likewise if you unlock an object with mcount > 0, then
;  the lock will be decremented.  Note:  you must make sure that th can
;  and should get the lock, since this function will blindly go ahead and
;  get the lock

(defun lock-object (th obj-ref heap)
```

```
      (let* ((obj-ref-num (cadr obj-ref))
             (instance (binding (cadr obj-ref) heap))
             (obj-fields (binding "Object" instance))
             (new-mcount (+ 1 (binding "mcount" obj-fields)))
             (new-obj-fields
                (bind "monitor" th
                   (bind "mcount" new-mcount obj-fields)))
             (new-object (bind "Object" new-obj-fields instance)))
        (bind obj-ref-num new-object heap)))

    (defun unlock-object (th obj-ref heap)
      (let* ((obj-ref-num (cadr obj-ref))
             (instance (binding (cadr obj-ref) heap))
             (obj-fields (binding "Object" instance))
             (old-mcount (binding "mcount" obj-fields))
             (new-mcount (ACL2::max 0 (- old-mcount 1)))
             (new-monitor (if (zp new-mcount)
                              0
                              th))
             (new-obj-fields
                (bind "monitor" new-monitor
                   (bind "mcount" new-mcount obj-fields)))
             (new-object (bind "Object" new-obj-fields instance)))
        (bind obj-ref-num new-object heap)))

    ; objectLockable? is used to determine if th can unlock instance.  This
    ;  occurs when either mcount is zero (nobody has a lock), or mcount is
    ;  greater than zero, but monitor is equal to th.  This means that th
    ;  already has a lock on the object, and when the object is locked yet again,
    ;  monitor will remain the same, but mcount will be incremented.
    ;
    ; objectUnLockable? determins if a thread can unlock an object (ie if it
    ;  has a lock on that object)
    (defun objectLockable? (instance th)
      (let* ((obj-fields (binding "Object" instance))
             (monitor (binding "monitor" obj-fields))
             (mcount (binding "mcount" obj-fields)))
        (or (zp mcount)
            (equal monitor th))))

    (defun objectUnLockable? (instance th)
      (let* ((obj-fields (binding "Object" instance))
             (monitor (binding "monitor" obj-fields)))
        (equal monitor th)))

    ; ------------------------------------------------------------------------------
```

```
275   ; Frames

      (defun make-frame (pc locals stack program sync-flg)
        (list pc locals stack program sync-flg))

280   (defun top-frame (th s) (top (call-stack th s)))

      (defun pc      (frame) (nth 0 frame))
      (defun locals  (frame) (nth 1 frame))
      (defun stack   (frame) (nth 2 frame))
285   (defun program (frame) (nth 3 frame))
      (defun sync-flg (frame) (nth 4 frame))

      (defthm frames
        (and
290      (equal (pc (make-frame pc l s prog sync-flg)) pc)
         (equal (locals (make-frame pc l s prog sync-flg)) l)
         (equal (stack (make-frame pc l s prog sync-flg)) s)
         (equal (program (make-frame pc l s prog sync-flg)) prog)
         (equal (sync-flg (make-frame pc l s prog sync-flg)) sync-flg)))
295
      (in-theory (disable make-frame pc locals stack program sync-flg))

      ; ------------------------------------------------------------------------
      ; Method Declarations
300
      ; The methods component of a class declaration is a list of method definitions.
      ; A method definition is a list of the form

      ; (name formals sync-status . program)
305
      ; We never build these declarations but just enter list constants for them,

      ; Note the similarity to our old notion of a program definition.  We
      ; will use strings to name methods now.
310
      ; sync-status is 't' if the method is synchronized, 'nil' if not

      ; Method definitions will be constructed by expressions such as:
      ; (Note:  all of the symbols below are understood to be in the pkg "JVM".)
315
      ; ("move" (dx dy) nil
      ;   (load this)
      ;   (load this)
      ;   (getfield "Point" "x")
320   ;   (load dx)
```

```
;   (add)
;   (putfield "Point" "x")    ; this.x = this.x + dx;
;   (load :this)
;   (load :this)
;   (getfield "Point" "y")
;   (load dy)
;   (add)
;   (putfield "Point" "y")    ; this.y = this.y + dy;
;   (push 1)
;   (xreturn)))               ; return 1;

; Provided this method is defined in the class "Point" it can be invoked by

;   (invokevirtual "Point" "move" 2)

; This assumes that the stack, at the time of invocation, contains an
; reference to an object of type "Point" and two numbers, dx and dy.

; If a method declaration has an empty list for the program (ie- there are
; no bytecodes associated with the method), then the method is considered
; native.  Native methods are normally written in something like C or
; assembly language.  The JVM would normally ensure that the correct number
; and type of arguments are passed to the native method, and would then hand
; over control to C.  In our model, we simply "hardwire" invokevirtual to
; to handle these methods.
;  * Note that a method in Java will never have 0 bytecodes, since even if
;    it has no body, it will consist of at least the (xreturn) bytecode.

; The accessors for methods are:

(defun method-name (m)
  (nth 0 m))
(defun method-formals (m)
  (nth 1 m))
(defun method-sync (m)
  (nth 2 m))
(defun method-program (m)
  (cdddr m))
(defun method-isNative? (m)
  (equal '(NIL)
         (method-program m)))

; The Standard Modify

(defun suppliedp (key args)
  (cond ((endp args) nil)
```

```
          ((equal key (car args)) t)
          (t (suppliedp key (cdr args)))))

(defun actual (key args)
  (cond ((endp args) nil)
        ((equal key (car args)) (cadr args))
        (t (actual key (cdr args)))))

(defmacro modify (th s &rest args)
  (list 'make-state
        (cond
         ((or (suppliedp :call-stack args)
              (suppliedp :pc args)
              (suppliedp :locals args)
              (suppliedp :stack args)
              (suppliedp :program args)
              (suppliedp :sync-flg args)
              (suppliedp :status args))
          (list 'modify-tt
                th
                (cond ((suppliedp :call-stack args)
                       (actual :call-stack args))
                      ((and (suppliedp :status args)
                            (null (cddr args)))
                       (list 'call-stack th s))
                      (t
                       (list 'push
                             (list 'make-frame
                                   (if (suppliedp :pc args)
                                       (actual :pc args)
                                     (list 'pc (list 'top-frame th s)))
                                   (if (suppliedp :locals args)
                                       (actual :locals args)
                                     (list 'locals (list 'top-frame th s)))
                                   (if (suppliedp :stack args)
                                       (actual :stack args)
                                     (list 'stack (list 'top-frame th s)))
                                   (if (suppliedp :program args)
                                       (actual :program args)
                                     (list 'program (list 'top-frame th s)))
                                   (if (suppliedp :sync-flg args)
                                       (actual :sync-flg args)
                                     (list 'sync-flg (list 'top-frame th s))))
                             (list 'pop (list 'call-stack th s)))))
                (if (suppliedp :status args)
                    (actual :status args)
```

```
                    ''JVM::SCHEDULED)
                  (list 'thread-table s)))
415           ((suppliedp :thread-table args)
              (actual :thread-table args))
             (t (list 'thread-table s)))
            (if (suppliedp :heap args)
                (actual :heap args)
              (list 'heap s))
            (if (suppliedp :class-table args)
                (actual :class-table args)
              (list 'class-table s))))

; ------------------------------------------------------------------------
; (PUSH const) Instruction

(defun execute-PUSH (inst th s)
  (modify th s
          :pc (+ 1 (pc (top-frame th s)))
          :stack (push (arg1 inst)
                       (stack (top-frame th s)))))

; ------------------------------------------------------------------------
; (POP) Instruction

(defun execute-POP (inst th s)
  (declare (ignore inst))
  (modify th s
          :pc (+ 1 (pc (top-frame th s)))
          :stack (pop (stack (top-frame th s)))))

; ------------------------------------------------------------------------
; (LOAD var) Instruction

(defun execute-LOAD (inst th s)
  (modify th s
          :pc (+ 1 (pc (top-frame th s)))
          :stack (push (binding (arg1 inst)
                                (locals (top-frame th s)))
                       (stack (top-frame th s)))))

; ------------------------------------------------------------------------
; (STORE var) Instruction

(defun execute-STORE (inst th s)
  (modify th s
          :pc (+ 1 (pc (top-frame th s)))
```

```
              :locals (bind (arg1 inst)
                            (top (stack (top-frame th s)))
                            (locals (top-frame th s)))
              :stack (pop (stack (top-frame th s)))))

    ; ------------------------------------------------------------------------
    ; (DUP) Instruction

    (defun execute-DUP (inst th s)
      (declare (ignore inst))
      (modify th s
              :pc (+ 1 (pc (top-frame th s)))
              :stack (push (top (stack (top-frame th s)))
                           (stack (top-frame th s)))))

    ; ------------------------------------------------------------------------
    ; (ADD) Instruction

    (defun execute-ADD (inst th s)
      (declare (ignore inst))
      (modify th s
              :pc (+ 1 (pc (top-frame th s)))
              :stack (push (+ (top (pop (stack (top-frame th s))))
                              (top (stack (top-frame th s))))
                           (pop (pop (stack (top-frame th s)))))))

    ; ------------------------------------------------------------------------
    ; (SUB) Instruction

    (defun execute-SUB (inst th s)
      (declare (ignore inst))
      (modify th s
              :pc (+ 1 (pc (top-frame th s)))
              :stack (push (- (top (pop (stack (top-frame th s))))
                              (top (stack (top-frame th s))))
                           (pop (pop (stack (top-frame th s)))))))

    ; ------------------------------------------------------------------------
    ; (MUL) Instruction

    (defun execute-MUL (inst th s)
      (declare (ignore inst))
      (modify th s
              :pc (+ 1 (pc (top-frame th s)))
              :stack (push (* (top (pop (stack (top-frame th s))))
                              (top (stack (top-frame th s))))
```

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

; ------------------------------------------------------------------------
; (GOTO pc) Instruction

(defun execute-GOTO (inst th s)
  (modify th s
          :pc (+ (arg1 inst) (pc (top-frame th s)))))

; ------------------------------------------------------------------------
; (IFEQ pc) Instruction

(defun execute-IFEQ (inst th s)
  (modify th s
          :pc (if (equal (top (stack (top-frame th s))) 0)
                  (+ (arg1 inst) (pc (top-frame th s)))
                (+ 1 (pc (top-frame th s))))
          :stack (pop (stack (top-frame th s)))))

; ------------------------------------------------------------------------
; (IFNE pc) Instruction

(defun execute-IFNE (inst th s)
  (modify th s
          :pc (if (equal (top (stack (top-frame th s))) 0)
                  (+ 1 (pc (top-frame th s)))
                (+ (arg1 inst) (pc (top-frame th s))))
          :stack (pop (stack (top-frame th s)))))

; ------------------------------------------------------------------------
; (IFGT pc) Instruction

(defun execute-IFGT (inst th s)
  (modify th s
          :pc (if (> (top (stack (top-frame th s))) 0)
                  (+ (arg1 inst) (pc (top-frame th s)))
                (+ 1 (pc (top-frame th s))))
          :stack (pop (stack (top-frame th s)))))

; ------------------------------------------------------------------------
; (IFLT pc) Instruction

(defun execute-IFLT (inst th s)
  (modify th s
          :pc (if (< (top (stack (top-frame th s))) 0)
                  (+ (arg1 inst) (pc (top-frame th s)))
```

```
                (+ 1 (pc (top-frame th s))))
          :stack (pop (stack (top-frame th s)))))

; ------------------------------------------------------------------------
; (GETFIELD "class" "field") Instruction

(defun deref (ref heap)
  (binding (cadr ref) heap))

(defun field-value (class-name field-name instance)
  (binding field-name
           (binding class-name instance)))

(defun execute-GETFIELD (inst th s)
  (let* ((class-name (arg1 inst))
         (field-name (arg2 inst))
         (instance (deref (top (stack (top-frame th s))) (heap s)))
         (field-value (field-value class-name field-name instance)))
    (modify th s
            :pc (+ 1 (pc (top-frame th s)))
            :stack (push field-value
                         (pop (stack (top-frame th s)))))))

; ------------------------------------------------------------------------
; (PUTFIELD "class" "field") Instruction

(defun set-instance-field (class-name field-name value instance)
  (bind class-name
        (bind field-name value
              (binding class-name instance))
        instance))

(defun execute-PUTFIELD (inst th s)
  (let* ((class-name (arg1 inst))
         (field-name (arg2 inst))
         (value (top (stack (top-frame th s))))
         (instance (deref (top (pop (stack (top-frame th s)))) (heap s)))
         (address (cadr (top (pop (stack (top-frame th s))))))
        (modify th s
                :pc (+ 1 (pc (top-frame th s)))
                :stack (pop (pop (stack (top-frame th s))))
                :heap (bind address
                            (set-instance-field class-name
                                                field-name
                                                value
                                                instance)
```

```
                                (heap s)))))

; ------------------------------------------------------------------------------
; (INVOKEVIRTUAL "class" "name" n) Instruction

(defun bind-formals (rformals stack)
  (if (endp rformals)
      nil
    (cons (cons (car rformals) (top stack))
          (bind-formals (cdr rformals) (pop stack)))))

(defun popn (n stack)
  (if (zp n)
      stack
    (popn (- n 1) (pop stack))))

(defun class-name-of-ref (ref heap)
  (car (car (deref ref heap))))

(defun lookup-method-in-superclasses (name classes class-table)
  (cond ((endp classes) nil)
        (t (let* ((class-name (car classes))
                  (class-decl (bound? class-name class-table))
                  (method (bound? name (class-decl-methods class-decl))))
             (if method
                 method
                (lookup-method-in-superclasses name (cdr classes)
                                               class-table))))))

(defun lookup-method (name class-name class-table)
  (lookup-method-in-superclasses name
                                 (cons class-name
                                       (class-decl-superclasses
                                        (bound? class-name class-table)))
                                 class-table))

(defun execute-INVOKEVIRTUAL (inst th s)
  (let* ((method-name (arg2 inst))
         (nformals (arg3 inst))
         (obj-ref (top (popn nformals (stack (top-frame th s)))))
         (instance (deref obj-ref (heap s)))
         (obj-class-name (class-name-of-ref obj-ref (heap s)))
         (closest-method
          (lookup-method method-name
                         obj-class-name
                         (class-table s)))
```

```
            (vars (cons 'jvm::this (method-formals closest-method)))
            (prog (method-program closest-method))
            (s1 (modify th s
                        :pc (+ 1 (pc (top-frame th s)))
                        :stack (popn (len vars)
                                     (stack (top-frame th s)))))
            (tThread (rrefToThread obj-ref (thread-table s))))
       (cond
        ((method-isNative? closest-method)
         (cond ((equal method-name "start")
                (modify tThread s1
                        :status 'JVM::SCHEDULED))
               ((equal method-name "stop")
                (modify tThread s1
                        :status 'JVM::UNSCHEDULED))
               (t s)))
        ((and (method-sync closest-method)
              (objectLockable? instance th))
         (modify th s1
                 :call-stack
                 (push (make-frame 0
                                   (reverse
                                    (bind-formals (reverse vars)
                                                  (stack (top-frame th s))))
                                   nil
                                   prog
                                   'JVM::LOCKED)
                       (call-stack th s1))
                 :heap (lock-object th obj-ref (heap s))))
        ((method-sync closest-method)
         s)
        (t
         (modify th s1
                 :call-stack
                 (push (make-frame 0
                                   (reverse
                                    (bind-formals (reverse vars)
                                                  (stack (top-frame th s))))
                                   nil
                                   prog
                                   'JVM::UNLOCKED)
                       (call-stack th s1)))))))

; -----------------------------------------------------------------------------
; (NEW "class") Instruction
```

```
(defun build-class-field-bindings (field-names)
  (if (endp field-names)
      nil
    (cons (cons (car field-names) 0)
          (build-class-field-bindings (cdr field-names)))))

(defun build-class-object-field-bindings ()
  '(("monitor" . 0) ("monitor-count" . 0) ("wait-set" . nil)))

(defun build-immediate-instance-data (class-name class-table)
  (cons class-name
      (build-class-field-bindings
       (class-decl-fields
        (bound? class-name class-table)))))

(defun build-an-instance (class-names class-table)
  (if (endp class-names)
      nil
    (cons (build-immediate-instance-data (car class-names) class-table)
          (build-an-instance (cdr class-names) class-table))))

(defun execute-NEW (inst th s)
  (let* ((class-name (arg1 inst))
         (class-table (class-table s))
         (closest-method (lookup-method "run" class-name class-table))
         (prog (method-program closest-method))
         (new-object (build-an-instance
                      (cons class-name
                            (class-decl-superclasses
                             (bound? class-name class-table)))
                      class-table))
         (new-address (len (heap s)))
         (s1 (modify th s
                     :pc (+ 1 (pc (top-frame th s)))
                     :stack (push (list 'JVM::REF new-address)
                                  (stack (top-frame th s)))
                     :heap (bind new-address new-object (heap s)))))
    (if (isThreadObject? class-name class-table)
        (modify nil s1
                :thread-table
                (addto-tt
                 (push
                  (make-frame 0
                              (list (cons 'JVM::THIS (list 'JVM::REF new-address)))
                              nil
                              prog
```

```
                                    'JVM::UNLOCKED)
                     nil)
                    'JVM::UNSCHEDULED
                    (list 'JVM::REF new-address)
                    (thread-table s1)))
        s1)))


; ------------------------------------------------------------------------
; (RETURN) Instruction - Void Return

(defun execute-RETURN (inst th s)
  (declare (ignore inst))
  (let* ((obj-ref (binding 'JVM::THIS (locals (top-frame th s)))))
    (modify th s
            :call-stack (pop (call-stack th s))
            :heap (if (equal (sync-flg (top-frame th s)) 'JVM::LOCKED)
                      (unlock-object th obj-ref (heap s))
                    (heap s)))))

; ------------------------------------------------------------------------
; (XRETURN) Instruction - return 1 thing of arbitrary type

(defun execute-XRETURN (inst th s)
  (declare (ignore inst))
  (let* ((val (top (stack (top-frame th s))))
         (obj-ref (binding 'JVM::THIS (locals (top-frame th s))))
         (s1 (modify th s
                     :call-stack (pop (call-stack th s))
                     :heap (if (equal (sync-flg (top-frame th s)) 'JVM::LOCKED)
                               (unlock-object th obj-ref (heap s))
                             (heap s)))))
    (modify th s1
            :stack (push val (stack (top-frame th s1))))))


; ------------------------------------------------------------------------
; (MONITORENTER) Instruction

(defun execute-MONITORENTER (inst th s)
  (declare (ignore inst))
  (let* ((obj-ref (top (stack (top-frame th s))))
         (instance (deref obj-ref (heap s))))
    (cond
     ((objectLockable? instance th)
      (modify th s
```

```
                :pc (+ 1 (pc (top-frame th s)))
                :stack (pop (stack (top-frame th s)))
                :heap (lock-object th obj-ref (heap s))))
        (t s))))

; -----------------------------------------------------------------------------
; (MONITOREXIT) Instruction

(defun execute-MONITOREXIT (inst th s)
  (declare (ignore inst))
  (let* ((obj-ref (top (stack (top-frame th s))))
         (instance (deref obj-ref (heap s))))
    (cond
     ((objectUnLockable? instance th)
      (modify th s
              :pc (+ 1 (pc (top-frame th s)))
              :stack (pop (stack (top-frame th s)))
              :heap (unlock-object th obj-ref (heap s))))
     (t s))))

; -----------------------------------------------------------------------------
; Putting it all together

(defun next-inst (th s)
  (nth (pc (top-frame th s))
       (program (top-frame th s))))

(defun do-inst (inst th s)
  (case (op-code inst)
    (JVM::PUSH           (execute-PUSH inst th s))
    (JVM::POP            (execute-POP  inst th s))
    (JVM::LOAD           (execute-LOAD inst th s))
    (JVM::STORE          (execute-STORE inst th s))
    (JVM::DUP            (execute-DUP inst th s))
    (JVM::ADD            (execute-ADD inst th s))
    (JVM::SUB            (execute-SUB inst th s))
    (JVM::MUL            (execute-MUL inst th s))
    (JVM::GOTO           (execute-GOTO inst th s))
    (JVM::IFEQ           (execute-IFEQ inst th s))
    (JVM::IFNE           (execute-IFNE inst th s))
    (JVM::IFLT           (execute-IFLT inst th s))
    (JVM::IFGT           (execute-IFGT inst th s))
    (JVM::INVOKEVIRTUAL  (execute-INVOKEVIRTUAL inst th s))
    (JVM::RETURN         (execute-RETURN inst th s))
    (JVM::XRETURN        (execute-XRETURN inst th s))
    (JVM::NEW            (execute-NEW inst th s))
```

```
        (JVM::GETFIELD       (execute-GETFIELD inst th s))
        (JVM::PUTFIELD       (execute-PUTFIELD inst th s))
        (JVM::MONITORENTER   (execute-MONITORENTER inst th s))
        (JVM::MONITOREXIT    (execute-MONITOREXIT inst th s))
        (JVM::HALT           s)
        (otherwise s)))

(defun step4 (th s)
  (if (equal (call-stack-status th s) 'JVM::SCHEDULED)
    (do-inst (next-inst th s) th s)
    s))

(defun m4 (sched s)
  (if (endp sched)
      s
    (m4 (cdr sched) (step4 (car sched) s))))
```

C The proof script

```
;
; Commutative diagram between M3 and M4
; George Porter
;
; $Id: commute-diagram.lisp,v 1.12 2001/04/10 03:44:16 george Exp $

; up transforms an M4 state into an M3 state, with some loss of information
;

; (ld "commute-diagram.lisp" :ld-pre-eval-print t)

#|

(include-book "/v/hank/v113/george/src/thesis/m3")
(include-book "/v/hank/v113/george/src/thesis/m4")
(certify-book "commute-diagram" 2)

|#
(in-package "M4")

(defun up (s)
  (m3::make-state (car (binding 0 (m4::thread-table s)))
                  (m4::heap s)
                  (m4::class-table s)))

; down transforms an M3 state into an M4 state
;
(defun down (s)
  (m4::make-state (bind 0 (list (m3::call-stack s) 'JVM::SCHEDULED nil) nil)
                  (m3::heap s)
                  (m3::class-table s)))

; upsched transforms an M4 schedule into an M3 one
(defun upsched (sch)
  (if (endp sch)
      0
    (if (equal (car sch) 0)
        (+ 1 (upsched (cdr sch)))
      (upsched (cdr sch)))))

; almost-equal is our relation comparing M3 and M4 states, ignoring
;   unscheduled threads
;
(defun thread0-scheduled (tt)
```

```
45     (let* ((thd0 (binding 0 tt))
              (flag (cadr thd0)))
            (and (true-listp thd0)
                 (equal flag 'JVM::SCHEDULED))))

50   #|
     (defun no-threads-scheduled (tt)
       (cond ((endp tt) t)
             ((equal (caddr (car tt)) 'JVM::SCHEDULED) nil)
             (t (no-threads-scheduled (cdr tt)))))
55
     (defun only-thread0-scheduledp (tt)
       (and (thread0-scheduled tt)
            (no-threads-scheduled (cdr tt))))
     |#
60
     (defun at-most-thread0-scheduledp (tt)
       (cond ((endp tt) t)
             ((equal (caar tt) 0) (at-most-thread0-scheduledp (cdr tt)))
             (t (and (not (equal (caddr (car tt)) 'JVM::SCHEDULED))
65                   (at-most-thread0-scheduledp (cdr tt))))))

     (defun almost-equal (s4 s4p)
       (and (equal (call-stack 0 s4p)
                   (call-stack 0 s4))  ; call-stacks equal
70   ;       (at-most-thread0-scheduledp (thread-table s4))
     ;       (at-most-thread0-scheduledp (thread-table s4p))
             (equal (heap s4) (heap s4p))                 ; heaps equal
             (equal (class-table s4) (class-table s4p)))) ; class tables equal

75   ; singp is our predicate that determines if a state is single threaded
     ;

     ; no-starts-in-class-table helper functions
     ;
80   (defun check-bytecodes-in-method (bytecodes)
       (cond ((endp bytecodes) t)
             ((and (equal (caar bytecodes) 'JVM::INVOKEVIRTUAL)
                   (or (equal (caddar bytecodes) "start")
                       (equal (caddar bytecodes) "stop")))
85            nil)
             (t (check-bytecodes-in-method (cdr bytecodes)))))

     (defun check-methods-for-start (method-list)
       (if (endp method-list)
90         t
```

```
          (and (check-bytecodes-in-method (cdddr (car method-list)))
               (check-methods-for-start (cdr method-list)))))

(defun no-starts-in-class (class)
  (check-methods-for-start (cadddr class)))

(defun no-starts-in-class-table (ctable)
  (if (endp ctable)
      t
      (and (no-starts-in-class (car ctable))
           (no-starts-in-class-table (cdr ctable)))))

; no-bytecodex-in-frames helpers

(defun check-bytecodex-in-method (opcode bytecodes)
  (cond ((endp bytecodes) t)
        ((equal (car (car bytecodes)) opcode) nil)
        (t (check-bytecodex-in-method opcode (cdr bytecodes)))))

(defun check-methods-for-bytecodex (opcode method-list)
  (if (endp method-list)
      t
      (and (check-bytecodex-in-method opcode (cdddr (car method-list)))
           (check-methods-for-bytecodex opcode (cdr method-list)))))

(defun no-bytecodex-in-class (opcode class)
  (check-methods-for-bytecodex opcode (cadddr class)))

(defun no-bytecodex-in-class-table (opcode ctable)
  (if (endp ctable)
      t
      (and (no-bytecodex-in-class opcode (car ctable))
           (no-bytecodex-in-class-table opcode (cdr ctable)))))

; no-starts-in-frames helper functions
;

(defun no-starts-in-frames (frames)
  (if (endp frames)
      t
      (and (check-bytecodes-in-method (program (car frames)))
           (no-starts-in-frames (cdr frames)))))

(defun no-bytecodex-in-frames (opcode frames)
  (if (endp frames)
      t
```

```
          (and (check-bytecodex-in-method opcode (program (car frames)))
               (no-bytecodex-in-frames opcode (cdr frames)))))

; no-locked-frames helper functions

(defun no-locked-frames-in-frames (frames)
  (if (endp frames)
      t
      (and (not (equal (m4::sync-flg (car frames)) 'JVM::LOCKED))
           (no-locked-frames-in-frames (cdr frames)))))

(defun check-methods-for-locked-frames (method-list)
  (if (endp method-list)
      t
      (and (equal (method-sync (car method-list)) NIL)
           (check-methods-for-locked-frames (cdr method-list)))))

(defun no-locked-frames-in-class-table (ctable)
  (if (endp ctable)
      t
      (and (check-methods-for-locked-frames (cadddr (car ctable)))
           (no-locked-frames-in-class-table (cdr ctable)))))

; We now define the concept that the class table contains no native
; methods other than (possibly) "start" and "stop".

(defun check-other-native-methods (method-list)
  (if (endp method-list)
      t
   (if (or (equal (car (car method-list)) "start")
           (equal (car (car method-list)) "stop"))
       (check-other-native-methods (cdr method-list))
     (and (not (method-isnative? (car method-list)))
          (check-other-native-methods (cdr method-list))))))

(defun no-other-native-methods-in-class (class)
  (check-other-native-methods (cadddr class)))

(defun no-other-native-methods-in-class-table (ctable)
  (if (endp ctable)
      t
      (and (no-other-native-methods-in-class (car ctable))
           (no-other-native-methods-in-class-table (cdr ctable)))))

(defun singp (s)
  (and (at-most-thread0-scheduledp (thread-table s))
```

```
          (assoc-equal 0 (thread-table s))
          (equal (caddr (assoc-equal 0 (thread-table s))) 'JVM::SCHEDULED)
          (equal (cadddr (assoc-equal 0 (thread-table s))) nil)
          (no-starts-in-frames (car (binding 0 (thread-table s))))
          (no-starts-in-class-table (class-table s))
          (no-bytecodex-in-frames
               'JVM::MONITORENTER (car (binding 0 (thread-table s))))
          (no-bytecodex-in-class-table
               'JVM::MONITORENTER (class-table s))
          (no-bytecodex-in-frames
               'JVM::MONITOREXIT (car (binding 0 (thread-table s))))
          (no-bytecodex-in-class-table
               'JVM::MONITOREXIT (class-table s))
          (no-locked-frames-in-frames (car (binding 0 (thread-table s))))
          (no-locked-frames-in-class-table (class-table s))
          (no-other-native-methods-in-class-table (class-table s))))

; Theorems
; -----------------------------------------------------------------------------

; l2 shows that down and up are inverses, in an "almost-equal" sense
(defthm l2
  (implies (singp s)
           (almost-equal (down (up s)) s))
  :rule-classes nil)

; -----------------------------------------------------------------------------
; l3 - singp is preserved over stepping

(defthm assoc-equal-bind
  (equal (assoc-equal th1 (bind th2 x alist))
         (if (equal th1 th2)
             (cons th1 x)
           (assoc-equal th1 alist))))

(defthm at-most-thread0-scheduledp-bind
  (implies (at-most-thread0-scheduledp tt)
           (at-most-thread0-scheduledp (bind 0 entry tt))))

(defthm l3-lemma-EXECUTE-DUP
  (implies (singp s)
           (singp (execute-dup inst 0 s))))

(defthm l3-lemma-EXECUTE-ADD
  (implies (singp s)
           (singp (execute-add inst 0 s))))
```

```
(defthm l3-lemma-EXECUTE-PUSH
  (implies (singp s)
           (singp (EXECUTE-PUSH inst 0 s))))
(defthm l3-lemma-EXECUTE-POP
  (implies (singp s)
           (singp (EXECUTE-POP inst 0 s))))

(defthm l3-lemma-EXECUTE-LOAD
  (implies (singp s)
           (singp (EXECUTE-LOAD inst 0 s))))

(defthm l3-lemma-EXECUTE-STORE
  (implies (singp s)
           (singp (EXECUTE-STORE inst 0 s))))

(defthm l3-lemma-EXECUTE-SUB
  (implies (singp s)
           (singp (EXECUTE-SUB inst 0 s))))

(defthm l3-lemma-EXECUTE-MUL
  (implies (singp s)
           (singp (EXECUTE-MUL inst 0 s))))

(defthm l3-lemma-EXECUTE-GOTO
  (implies (singp s)
           (singp (EXECUTE-GOTO inst 0 s))))

(defthm l3-lemma-EXECUTE-IFEQ
  (implies (singp s)
           (singp (EXECUTE-IFEQ inst 0 s))))

(defthm l3-lemma-EXECUTE-IFNE
  (implies (singp s)
           (singp (EXECUTE-IFNE inst 0 s))))

(defthm l3-lemma-EXECUTE-IFLT
  (implies (singp s)
           (singp (EXECUTE-IFLT inst 0 s))))

(defthm l3-lemma-EXECUTE-IFGT
  (implies (singp s)
           (singp (EXECUTE-IFGT inst 0 s))))

; In the following, we were sometimes tempted to write (nth 2 ...) or (nth 3 ...)
; and other times tempted to write (caddr ...) or (cadddr ...).  We will just
```

```
; rewrite away all the NTHs of constants with this rule.

(defthm nth-opener
  (and (equal (nth 0 x) (car x))
       (implies (and (syntaxp (quotep i))
                     (integerp i)
                     (<= 0 i))
                (equal (nth (+ 1 i) x)
                       (nth i (cdr x))))))
(in-theory (disable nth))

(defthm at-most-thread0-scheduledp-bind-2
  (implies (and (at-most-thread0-scheduledp tt)
                (equal (cadr x) 'JVM::UNSCHEDULED))
           (at-most-thread0-scheduledp (bind th x tt))))

(defthm no-starts-in-class-table-implies-check-methods-for-start
  (implies (no-starts-in-class-table ct)
           (check-methods-for-start (cadddr (assoc-equal class ct)))))

(defthm check-methods-for-start-implies-check-bytecodes-in-method
  (implies (check-methods-for-start methods)
           (check-bytecodes-in-method (cdddr (assoc-equal method methods)))))

(defthm check-methods-for-start-implies-check-bytecodes-in-method-2
  (implies (no-starts-in-class-table ct)
           (check-bytecodes-in-method
            (cdddr
             (LOOKUP-METHOD-IN-SUPERCLASSES
              method
              classes
              ct)))))

; We repeat this for check-bytecodex-in-method.

(defthm no-bytecodex-in-class-table-implies-no-bytecodex-in-class
  (implies (no-bytecodex-in-class-table opcode ct)
           (check-methods-for-bytecodex opcode (cadddr (assoc-equal class ct)))))

(defthm check-methods-for-bytecodex-implies-check-bytecodex-in-method
  (implies (check-methods-for-bytecodex opcode methods)
           (check-bytecodex-in-method opcode (cdddr (assoc-equal method methods)))))

(defthm check-methods-for-bytecodex-implies-check-bytecodex-in-method-2
  (implies (no-bytecodex-in-class-table opcode ct)
           (check-bytecodex-in-method
```

```
              opcode
              (cdddr
               (LOOKUP-METHOD-IN-SUPERCLASSES
                method
                classes
                ct)))))
; ---

; We now have a similar argument to show that no invoked method is
; synchronized.

(defthm no-locked-frames-in-class-table-implies-not-method-sync
  (implies (NO-LOCKED-FRAMES-IN-CLASS-TABLE ct)
           (not (CADDR
                 (ASSOC-EQUAL method
                              (cadddr (ASSOC-EQUAL class ct)))))))

(defthm no-locked-frames-in-class-table-implies-not-method-sync-lookup-method
  (implies (NO-LOCKED-FRAMES-IN-CLASS-TABLE ct)
           (not (CADDR
                 (LOOKUP-METHOD-IN-SUPERCLASSES method superclasses ct)))))


(defthm l3-lemma-EXECUTE-INVOKEVIRTUAL
  (implies (and (singp s)
                (not (equal (caddr inst) "start"))
                (not (equal (caddr inst) "stop")))
           (singp (EXECUTE-INVOKEVIRTUAL inst 0 s))))

(defthm l3-lemma-EXECUTE-RETURN
  (implies (singp s)
           (singp (EXECUTE-RETURN inst 0 s))))

(defthm l3-lemma-EXECUTE-XRETURN
  (implies (singp s)
           (singp (EXECUTE-XRETURN inst 0 s))))

(defthm len-bind
  (equal (len (bind 0 v alist))
         (if (assoc-equal 0 alist) (len alist) (+ 1 (len alist)))))

(defthm assoc-equal-implies-non-0-len
  (implies (assoc-equal key alist)
           (not (equal 0 (len alist)))))

(defthm l3-lemma-EXECUTE-NEW
```

```
        (implies (singp s)
                 (singp (EXECUTE-NEW inst 0 s))))

(defthm l3-lemma-EXECUTE-GETFIELD
  (implies (singp s)
           (singp (EXECUTE-GETFIELD inst 0 s))))

(defthm l3-lemma-EXECUTE-PUTFIELD
  (implies (singp s)
           (singp (EXECUTE-PUTFIELD inst 0 s))))

; We don't really need these, because singp implies there are none of these
; instructions.  But in fact singp is preserved by them, so we prove these for
; future use.

(defthm l3-lemma-EXECUTE-MONITORENTER
  (implies (singp s)
           (singp (EXECUTE-MONITORENTER inst 0 s))))

(defthm l3-lemma-EXECUTE-MONITOREXIT
  (implies (singp s)
           (singp (EXECUTE-MONITOREXIT inst 0 s))))

(defthm only-thread-0-scheduled-lemma
  (IMPLIES (AND (AT-MOST-THREAD0-SCHEDULEDP tt)
                (EQUAL (CADDR (ASSOC-EQUAL TH tt))
                       'JVM::SCHEDULED))
           (EQUAL TH 0))
  :rule-classes nil)

(defthm only-thread-0-scheduled
  (implies (and (SINGP S)
                (EQUAL (CADDR (ASSOC-EQUAL TH (THREAD-TABLE S)))
                       'JVM::SCHEDULED))
           (equal th 0))
  :hints (("Goal" :use (:instance only-thread-0-scheduled-lemma
                                  (tt (thread-table s)))))
  :rule-classes nil)


; The next two lemmas are used to prove the lemma next-inst-not-start,
; which is needed in l3-lemma to prove the 2nd hyp of the l3-lemma
; invokevirtual case.

(defthm next-inst-not-start-lemma2-start
  (IMPLIES (and (CHECK-BYTECODES-IN-METHOD program)
```

```
                  (equal (car (NTH PC program)) 'JVM::INVOKEVIRTUAL))
             (NOT (EQUAL (CADDR (NTH PC program))
                         "start")))
    :hints (("Goal" :in-theory (enable nth))))

  (defthm next-inst-not-start-lemma2-stop
    (IMPLIES (and (CHECK-BYTECODES-IN-METHOD program)
                  (equal (car (NTH PC program)) 'JVM::INVOKEVIRTUAL))
             (NOT (EQUAL (CADDR (NTH PC program))
                         "stop")))
    :hints (("Goal" :in-theory (enable nth))))

  (defthm next-inst-not-start-lemma1-start
    (implies
     (and (NO-STARTS-IN-FRAMES cs)
          (equal (car (NTH pc (program (car cs)))) 'JVM::INVOKEVIRTUAL))
     (not (equal (caddr (NTH pc (program (car cs)))) "start")))
    :hints (("Goal" :in-theory (enable nth))))

  (defthm next-inst-not-start-lemma1-stop
    (implies
     (and (NO-STARTS-IN-FRAMES cs)
          (equal (car (NTH pc (program (car cs)))) 'JVM::INVOKEVIRTUAL))
     (not (equal (caddr (NTH pc (program (car cs)))) "stop")))
    :hints (("Goal" :in-theory (enable nth))))

  (defthm next-inst-not-start
    (implies
     (and (singp s)
          (equal (car (NTH pc (program
                               (CAADR (ASSOC-EQUAL 0 (THREAD-TABLE S))))))
                 'JVM::INVOKEVIRTUAL))
     (not (equal (caddr (NTH pc (program
                                 (CAADR (ASSOC-EQUAL 0 (THREAD-TABLE S))))))
                 "start"))))

  (defthm next-inst-not-stop
    (implies
     (and (singp s)
          (equal (car (NTH pc (program
                               (CAADR (ASSOC-EQUAL 0 (THREAD-TABLE S))))))
                 'JVM::INVOKEVIRTUAL))
     (not (equal (caddr (NTH pc (program
                                 (CAADR (ASSOC-EQUAL 0 (THREAD-TABLE S))))))
                 "stop"))))
```

```
    ; in order to relieve the 2nd hyp of the l3 INVOKVIRTUAL lemma.

    (defthm l3-lemma
      (implies (singp s)
               (singp (step4 th s)))
      :hints
      (("Goal"
        :use (:instance only-thread-0-scheduled
                        (s s)
                        (th th))
        :in-theory
        (disable EXECUTE-PUSH
                 EXECUTE-POP
                 EXECUTE-LOAD
                 EXECUTE-STORE
                 EXECUTE-DUP
                 EXECUTE-ADD
                 EXECUTE-SUB
                 EXECUTE-MUL
                 EXECUTE-GOTO
                 EXECUTE-IFEQ
                 EXECUTE-IFNE
                 EXECUTE-IFLT
                 EXECUTE-IFGT
                 EXECUTE-INVOKEVIRTUAL
                 EXECUTE-RETURN
                 EXECUTE-XRETURN
                 EXECUTE-NEW
                 EXECUTE-GETFIELD
                 EXECUTE-PUTFIELD
                 EXECUTE-MONITORENTER
                 EXECUTE-MONITOREXIT
                 singp))))

    (defthm l3
      (implies (singp s) (singp (m4 sched s)))
      :hints (("Goal" :in-theory (disable step4 singp))))

    (defthm state-decomposition-m4
      (iff (equal (M4::make-state tt1 h1 ct1)
                  (M4::make-state tt2 h2 ct2))
           (and (equal tt1 tt2)
                (equal h1 h2)
                (equal ct1 ct2)))
      :hints
      (("Goal"
```

```
505       :in-theory (enable M4::make-state))))

      (defthm state-decomposition-m3
        (iff (equal (M3::make-state cs1 h1 ct1)
                    (M3::make-state cs2 h2 ct2))
510           (and (equal cs1 cs2)
                   (equal h1 h2)
                   (equal ct1 ct2)))
        :hints
        (("Goal"
515       :in-theory (enable M3::make-state))))

      (defthm compare-m3-and-m4-make-frame
        (iff (equal (m3::make-frame pc1 locals1 stack1 program1 sync1)
                    (m4::make-frame pc2 locals2 stack2 program2 sync2))
520           (and (equal pc1 pc2)
                   (equal locals1 locals2)
                   (equal stack1 stack2)
                   (equal program1 program2)
                   (equal sync1 sync2)))
525     :hints
        (("Goal"
          :in-theory (enable m3::make-frame m4::make-frame))))

      (defthm m3-pc-is-m4-pc
530     (equal (m3::pc x) (m4::pc x))
        :hints
        (("Goal"
          :in-theory (enable m3::pc m4::pc))))

535   (defthm m3-locals-is-m4-locals
        (equal (m3::locals x) (m4::locals x))
        :hints
        (("Goal"
          :in-theory (enable m3::locals m4::locals))))
540
      (defthm m3-stack-is-m4-stack
        (equal (m3::stack x) (m4::stack x))
        :hints
        (("Goal"
545       :in-theory (enable m3::stack m4::stack))))

      (defthm m3-program-is-m4-program
        (equal (m3::program x) (m4::program x))
        :hints
550     (("Goal"
```

```
    :in-theory (enable m3::program m4::program))))

(defthm m3-sync-flg-is-m4-sync-flg
  (equal (m3::sync-flg x) (m4::sync-flg x))
  :hints
  (("Goal"
    :in-theory (enable m3::sync-flg m4::sync-flg))))

; L1

; L1-lemma1
(defthm l1-lemma1-not-0-is-not-sched
  (implies (and (at-most-thread0-scheduledp tt)
                (not (equal th 0)))
           (not (equal (caddr (assoc-equal th tt))
                       'JVM::SCHEDULED))))

(defthm l1-lemma1
  (implies (and (singp s)
                (not (equal th 0)))
           (equal (step4 th s) s)))

; L1-lemma2
(defthm l1-lemma2-EXECUTE-DUP
  (implies (singp s)
           (equal (M3::EXECUTE-DUP inst (up s))
                  (up (M4::EXECUTE-DUP inst 0 s))))
  :hints
  (("Goal"
    :in-theory (enable M3::pop M3::push M3::top))))

(defthm l1-lemma2-EXECUTE-ADD
  (implies (singp s)
           (equal (M3::EXECUTE-ADD inst (up s))
                  (up (M4::EXECUTE-ADD inst 0 s))))
  :hints
  (("Goal"
    :in-theory (enable M3::pop M3::push M3::top))))

(defthm l1-lemma2-EXECUTE-PUSH
  (implies (singp s)
           (equal (M3::EXECUTE-PUSH inst (up s))
                  (up (M4::EXECUTE-PUSH inst 0 s))))
  :hints
  (("Goal"
    :in-theory (enable M3::pop M3::push M3::top))))
```

```
(defthm l1-lemma2-EXECUTE-POP
  (implies (singp s)
           (equal (M3::EXECUTE-POP inst (up s))
                  (up (M4::EXECUTE-POP inst 0 s))))
  :hints
  (("Goal"
    :in-theory (enable M3::pop M3::push M3::top))))

(defthm l1-lemma2-EXECUTE-LOAD
  (implies (singp s)
           (equal (M3::EXECUTE-LOAD inst (up s))
                  (up (M4::EXECUTE-LOAD inst 0 s))))
  :hints
  (("Goal"
    :in-theory (enable M3::pop M3::push M3::top))))

(defthm m3-bind-is-m4-bind
  (equal (m3::bind x y list)
         (m4::bind x y list)))

(defthm l1-lemma2-EXECUTE-STORE
  (implies (singp s)
           (equal (M3::EXECUTE-STORE inst (up s))
                  (up (M4::EXECUTE-STORE inst 0 s))))
  :hints
  (("Goal"
    :in-theory (enable M3::pop M3::push M3::top))))

; (in-theory (disable m3-bind-is-m4-bind))

(defthm l1-lemma2-EXECUTE-SUB
  (implies (singp s)
           (equal (M3::EXECUTE-SUB inst (up s))
                  (up (M4::EXECUTE-SUB inst 0 s))))
  :hints
  (("Goal"
    :in-theory (enable M3::pop M3::push M3::top))))

(defthm l1-lemma2-EXECUTE-MUL
  (implies (singp s)
           (equal (M3::EXECUTE-MUL inst (up s))
                  (up (M4::EXECUTE-MUL inst 0 s))))
  :hints
  (("Goal"
    :in-theory (enable M3::pop M3::push M3::top))))
```

```
(defthm l1-lemma2-EXECUTE-GOTO
  (implies (singp s)
           (equal (M3::EXECUTE-GOTO inst (up s))
                  (up (M4::EXECUTE-GOTO inst 0 s))))
  :hints
  (("Goal"
    :in-theory (enable M3::pop M3::push M3::top))))

(defthm l1-lemma2-EXECUTE-IFEQ
  (implies (singp s)
           (equal (M3::EXECUTE-IFEQ inst (up s))
                  (up (M4::EXECUTE-IFEQ inst 0 s))))
  :hints
  (("Goal"
    :in-theory (enable M3::pop M3::push M3::top))))

(defthm l1-lemma2-EXECUTE-IFNE
  (implies (singp s)
           (equal (M3::EXECUTE-IFNE inst (up s))
                  (up (M4::EXECUTE-IFNE inst 0 s))))
  :hints
  (("Goal"
    :in-theory (enable M3::pop M3::push M3::top))))

(defthm l1-lemma2-EXECUTE-IFLT
  (implies (singp s)
           (equal (M3::EXECUTE-IFLT inst (up s))
                  (up (M4::EXECUTE-IFLT inst 0 s))))
  :hints
  (("Goal"
    :in-theory (enable M3::pop M3::push M3::top))))

(defthm l1-lemma2-EXECUTE-IFGT
  (implies (singp s)
           (equal (M3::EXECUTE-IFGT inst (up s))
                  (up (M4::EXECUTE-IFGT inst 0 s))))
  :hints
  (("Goal"
    :in-theory (enable M3::pop M3::push M3::top))))

(defthm l1-lemma2-EXECUTE-GETFIELD
  (implies (singp s)
           (equal (M3::EXECUTE-GETFIELD inst (up s))
                  (up (M4::EXECUTE-GETFIELD inst 0 s))))
  :hints
```

```
    (("Goal"
690   :in-theory (enable M3::pop M3::push M3::top))))

  (defthm l1-lemma2-EXECUTE-PUTFIELD
    (implies (singp s)
             (equal (M3::EXECUTE-PUTFIELD inst (up s))
695                 (up (M4::EXECUTE-PUTFIELD inst 0 s))))
    :hints
    (("Goal"
      :in-theory (enable M3::pop M3::push M3::top m3-bind-is-m4-bind))))

700 (defthm no-other-native-methods-in-class-table-implies-no-natives
    (implies (and (no-other-native-methods-in-class-table ct)
                  (not (equal method "start"))
                  (not (equal method "stop")))
             (NOT
705           (EQUAL
              '(NIL)
              (CDDDR
               (ASSOC-EQUAL
                method
710             (CADDDR (ASSOC-EQUAL class ct)))))))))

  (defthm no-other-native-methods-in-class-table-implies-no-natives-lookup-method
    (implies (and (no-other-native-methods-in-class-table ct)
                  (not (equal method "start"))
715               (not (equal method "stop")))
             (NOT
              (EQUAL
               '(NIL)
               (CDDDR
720             (LOOKUP-METHOD-IN-SUPERCLASSES
                method
                superclasses
                ct))))))

725 (defthm m3-popn-is-m4-popn
    (equal (m3::popn n stack)
           (m4::popn n stack))
    :hints (("Goal" :in-theory (enable m3::pop m4::pop))))

730 (defthm assoc-equal-modify-tt
    (equal (assoc-equal th (modify-tt th cs status tt))
           (list th cs status (call-stack-rref th tt))))
```

```
(defthm call-stack-make-state-modify-tt
  (equal (call-stack th (make-state (modify-tt th cs status tt) heap ct))
         cs)
  :hints (("Goal" :in-theory (enable make-state))))

(defthm m3-reverse-is-m4-reverse
  (equal (m3::reverse x)
         (m4::reverse x)))

(defthm m3-bind-formals-is-m4-bind-formals
  (equal (m3::bind-formals rformals stack)
         (m4::bind-formals rformals stack))
  :hints (("Goal" :in-theory (enable m3::pop m4::pop m3::top m4::top))))

(defthm M3-LOOKUP-METHOD-IN-SUPERCLASSES-is-M4-LOOKUP-METHOD-IN-SUPERCLASSES
  (equal (M3::LOOKUP-METHOD-IN-SUPERCLASSES method superclasses ct)
         (M4::LOOKUP-METHOD-IN-SUPERCLASSES method superclasses ct)))

(defthm l1-lemma-EXECUTE-INVOKEVIRTUAL
  (implies (and (singp s)
                (not (equal (caddr inst) "start"))
                (not (equal (caddr inst) "stop")))
           (equal (M3::EXECUTE-INVOKEVIRTUAL inst (up s))
                  (up (M4::EXECUTE-INVOKEVIRTUAL inst 0 s))))
  :hints
  (("Goal"
    :in-theory (enable M3::pop M3::push M3::top))))

(defthm l1-lemma2-EXECUTE-RETURN
  (implies (singp s)
           (equal (M3::EXECUTE-return inst (up s))
                  (up (M4::EXECUTE-return inst 0 s))))
  :hints
  (("Goal"
    :in-theory (enable M3::pop M3::push M3::top))))

(defthm l1-lemma2-EXECUTE-XRETURN
  (implies (singp s)
           (equal (M3::EXECUTE-xreturn inst (up s))
                  (up (M4::EXECUTE-xreturn inst 0 s))))
  :hints
  (("Goal"
    :in-theory (enable M3::pop M3::push M3::top))))


(defthm M3-BUILD-CLASS-FIELD-BINDINGS-IS-M4-BUILD-CLASS-FIELD-BINDINGS
```

```
    (equal (M3::BUILD-CLASS-FIELD-BINDINGS fields)
           (M4::BUILD-CLASS-FIELD-BINDINGS fields)))

(defthm M3-BUILD-AN-INSTANCE-is-M4-BUILD-AN-INSTANCE
  (equal (M3::BUILD-AN-INSTANCE class ct)
         (M4::BUILD-AN-INSTANCE class ct)))

(defthm l1-lemma2-EXECUTE-new
  (implies (singp s)
           (equal (M3::EXECUTE-new inst (up s))
                  (up (M4::EXECUTE-new inst 0 s))))
  :hints
  (("Goal"
    :in-theory (enable M3::pop M3::push M3::top))))

(defthm caar-up
  (equal (CAAR (UP S))
         (CAADR (ASSOC-EQUAL 0 (THREAD-TABLE S))))
  :hints (("Goal" :in-theory (enable M3::MAKE-STATE))))

(defthm check-bytecodex-in-method-implies-no-monitorenter
  (IMPLIES (CHECK-BYTECODEX-IN-METHOD 'JVM::MONITORENTER
                                      program)
           (NOT (EQUAL (CAR (NTH pc program))
                       'JVM::MONITORENTER)))
  :hints (("Goal" :in-theory (enable nth))))

(defthm check-bytecodex-in-method-implies-no-monitorexit
  (IMPLIES (CHECK-BYTECODEX-IN-METHOD 'JVM::MONITOREXIT
                                      program)
           (NOT (EQUAL (CAR (NTH pc program))
                       'JVM::MONITOREXIT)))
  :hints (("Goal" :in-theory (enable nth))))

(defthm check-bytecodex-in-method-implies-no-monitorenter-instr
  (implies (NO-BYTECODEX-IN-FRAMES 'JVM::MONITORENTER cs)
           (not (EQUAL (CAR (NTH (PC (CAr cs))
                                 (PROGRAM (CAr cs))))
                       'JVM::MONITORENTER))))

(defthm check-bytecodex-in-method-implies-no-monitorexit-instr
  (implies (NO-BYTECODEX-IN-FRAMES 'JVM::MONITOREXIT cs)
           (not (EQUAL (CAR (NTH (PC (CAr cs))
                                 (PROGRAM (CAr cs))))
                       'JVM::MONITOREXIT))))
(defthm l1-lemma2
```

```
        (implies (singp s)
                 (equal (M3::step3 (up s))
                        (up (step4 0 s))))
  :hints
  (("Goal"
    :in-theory
    (union-theories
     '(m3::top m3::call-stack)
     (disable M3::EXECUTE-PUSH
              M4::EXECUTE-PUSH
              M3::EXECUTE-POP
              M4::EXECUTE-POP
              M3::EXECUTE-LOAD
              M4::EXECUTE-LOAD
              M3::EXECUTE-STORE
              M4::EXECUTE-STORE
              M3::EXECUTE-DUP
              M4::EXECUTE-DUP
              M3::EXECUTE-ADD
              M4::EXECUTE-ADD
              M3::EXECUTE-SUB
              M4::EXECUTE-SUB
              M3::EXECUTE-MUL
              M4::EXECUTE-MUL
              M3::EXECUTE-GOTO
              M4::EXECUTE-GOTO
              M3::EXECUTE-IFEQ
              M4::EXECUTE-IFEQ
              M3::EXECUTE-IFNE
              M4::EXECUTE-IFNE
              M3::EXECUTE-IFLT
              M4::EXECUTE-IFLT
              M3::EXECUTE-IFGT
              M4::EXECUTE-IFGT
              M3::EXECUTE-INVOKEVIRTUAL
              M4::EXECUTE-INVOKEVIRTUAL
              M3::EXECUTE-RETURN
              M4::EXECUTE-RETURN
              M3::EXECUTE-XRETURN
              M4::EXECUTE-XRETURN
              M3::EXECUTE-NEW
              M4::EXECUTE-NEW
              M3::EXECUTE-GETFIELD
              M4::EXECUTE-GETFIELD
              M3::EXECUTE-PUTFIELD
              M4::EXECUTE-PUTFIELD
```

```
                  M4::EXECUTE-MONITORENTER
                  M4::EXECUTE-MONITOREXIT
                                          ;              singp
                  up
                  )))))

(include-book "/projects/acl2/v2-5/books/arithmetic/top-with-meta")

(defthm l1
  (implies (singp s)
           (equal (M3::m3 (upsched sch) (up s))
                  (up (m4 sch s))))
  :hints (("Goal" :in-theory (disable singp m3::step3 m4::step4 up))))

(defthm main
  (implies (singp s)
           (almost-equal (down (m3::m3 (upsched sch) (up s)))
                         (m4 sch s)))
  :hints
  (("Goal" :in-theory (disable down up upsched m3::m3 m4 almost-equal singp)
           :use ((:instance l2 (s (m4 sch s)))))))

; ----------------------------------------------------------------
; Below are application independent support lemmas and theorems
; that allow one to port M3 properties to M4.

(defthm condition3
  (equal (m3::top-frame (up s)) (top-frame 0 s))
  :hints (("Goal" :in-theory (enable m3::top))))

(defthm condition4
  (equal (m3::call-stack (up s)) (call-stack 0 s)))

(defthm almost-equal-bind-0
  (equal (almost-equal s0 (make-state (bind 0 thread-entry thread-table) heap class-table
         (almost-equal s0 (make-state (list (cons 0 thread-entry)) heap class-table))))

(defthm almost-equal-commutes
  (equal (almost-equal s0 s1)
         (almost-equal s1 s0)))

(defthm heap-and-class-table-up
  (and (equal (m3::heap (up s)) (heap s))
       (equal (m3::class-table (up s)) (class-table s))))

(defthm m3-states-again
```

```
      (and
       (implies (equal z (m3::make-state cs hp ct))
                (equal (m3::call-stack z) cs))
       (implies (equal z (m3::make-state cs hp ct))
                (equal (m3::heap z) hp))
       (implies (equal z (m3::make-state cs hp ct))
                (equal (m3::class-table z) ct))))

(defthm m3-push-and-pop
  (and (equal (m3::push x y) (push x y))
       (equal (m3::pop x) (pop x)))
  :hints (("Goal" :in-theory (enable m3::push m3::pop))))

(defthm m3-make-frame-to-m4
  (equal (m3::make-frame cs lc st pr fl)
         (make-frame cs lc st pr fl))
  :hints (("Goal" :in-theory (enable m3::make-frame make-frame))))

(defthm singp-implies-rref-nil
  (implies (singp s0)
           (equal (CADDDR (ASSOC-EQUAL 0 (THREAD-TABLE S0))) nil)))
```