Available online at www.sciencedirect.com

SCIENCE DIRECT®

Science of Computer Programming 57 (2005) 253–274

Science of Computer Programming

www.elsevier.com/locate/scico

Executable JVM model for analytical reasoning: A study

Hanbing Liu*, J. Strother Moore

Department of Computer Sciences, University of Texas at Austin, 1 University Station C0500, Austin, TX 78712-0233, USA

Received 3 December 2003; received in revised form 21 June 2004; accepted 27 July 2004
Available online 10 May 2005

Abstract

To study the properties of the Java Virtual Machine (JVM) and Java programs, our research group has produced a series of JVM models written in a functional subset of Common Lisp. In this paper, we present our most complete JVM model from this series, namely, M6, which is derived from a careful study of the J2ME KVM [Connected Limited Device Configuration (CLDC) and the K Virtual Machine, http://java.sun.com/products/cldc/] implementation.

On the one hand, our JVM model is a conventional machine emulator. M6 implements dynamic class loading, class initialization and synchronization via monitors. It executes most J2ME CLDC Java programs that do not use any I/O or floating point operations. Engineers may consider M6 an implementation of the JVM. The June 2003 version is implemented with around 10K lines of Lisp in 28 modules.

On the other hand, M6 is novel because it allows for analytical reasoning in addition to conventional testing. M6 is written in an applicative (side-effect free) subset of Common Lisp, for which we have given precise meaning in terms of axioms and inference rules. Properties of M6 and its bytecoded programs can be expressed as formulas and proved as theorems. Proofs are constructed interactively with a mechanical theorem prover. Its concreteness, completeness, executability and mechanized reasoning support make our model unique among JVM models.

We argue that our approach of building an executable model of the system with an axiomatically described functional language can bring benefits from both the testing and the formal reasoning worlds.

* Corresponding author.
E-mail addresses: hbl@cs.utexas.edu (H. Liu), moore@cs.utexas.edu (J.S. Moore).

0167-6423/$ - see front matter © 2005 Elsevier B.V. All rights reserved.
doi:10.1016/j.scico.2004.07.004

Keywords: JVM model; Bytecode verification; Simulator; Virtual machine; Formal methods; Program verification; Software specification

1. Introduction

Simulators have long been an important tool in the process of designing computer hardware systems. For example, SimpleScalar [3] has been widely used in computer architecture research labs to explore the design space of processors to achieve better performance. Verilog-XL and Synopsys VCS are widely used commercial simulators for hardware designs. Hardware simulators are often used for detecting flaws (in the design or the evolving software base to be run on it) before the physical artifact is built.

Such simulators are usually written with the intention of using them for extensive validation tests. Most of them are written in C to strive for the best simulation speed. Unfortunately, the efforts for speed come at a price. The state representations of those simulators are hard to characterize, because the state of a simulated artifact is mingled with the state of the machine on which the simulator is executed. The meaning of operations of the simulated artifacts can only be expressed as the side effects on the underlying machine state caused by executing the C programs that implement the operations. These make it almost impossible to reason about the simulator rigorously.

In this paper, we present our executable JVM model M6. Unlike most other simulators, M6 is not written in an imperative programming language like C. Instead, it is written in ACL2, a functional (side-effect free) subset of Common Lisp, which can also be cast as a mathematical logic [11].

We can execute the simulator as a Common Lisp program. It is derived from a careful study of the JVM specification. We also used the C implementation of the Java virtual machine KVM [5] as a reference for writing native methods and implementing complicated mechanisms described in the JVM specification. It supports almost all interesting features of the J2ME JVM [10], including class loading, class initialization, synchronization via monitors as well as exception handling. We are still working towards modeling the access permissions checking accurately.

In addition, we can also treat the JVM simulator as a set of formulas in the ACL2 logic and we can reason about the JVM model analytically using axioms and rules of inference. We use a computer aided reasoning tool, ACL2, to help us produce mechanically checked proofs of the properties of the model. For example, we have proved a property of the class loader in this model: if class A is loaded and class A is assignable to class B, then class B is also loaded. We have also proved that a simple Java program for computing factorials does compute factorials on this model, with appropriate consideration for the JVM's 32-bit `int` arithmetic. We prove "properties of Java" source code by compiling the Java to JVM bytecode with Sun's `javac` compiler and verifying the bytecode with respect to the M6 model.[1]

[1] In fact, the "properties" are properties of the actual bytecode programs.

In fact, our long term research project is more concerned with devising ways for reasoning about a complex software system effectively. Proving properties of the model and bytecoded programs for it is our group's main focus. Building a complete, accurate, and executable JVM model is needed as a firm base for the analysis. We expect to use our model to verify, formally and mechanically, the Java bytecode verifier and the class loader.[2] For example, in studying the Java bytecode verifier, we are particularly interested in issues of practical importance. That is whether the bytecode verification mechanism as described in the JVM specification can provide the safety guarantees for bytecode program executions. Such issues include how the current bytecode verifier (together with the JVM's linking mechanism) can provide adequate access control at runtime; how the bytecode verifier can correctly enforce the right restrictions on the kind of operations that can be applied to uninitialized objects and partially initialized objects, i.e. how a JVM can prevent bytecode programs breaking the proper constructor abstraction. In short, we hope to use the model to explore interactions between those ostensibly "meta"-level facilities of the JVM and the high level language semantics. We also expect our work to form the basis of a Java source-code verification system.

To pursue work on the bytecode verifier we have introduced a "defensive" [4] version of M6 that is manifestly safe by virtue of performing runtime checks to insure the legality of every operation. We are working on proofs that relate this machine to M6 under the hypothesis that the bytecode verifier approves of the bytecode. Our expectation that an analyzable semantics for JVM bytecode will enable the verification of Java source code is based on the experience of Yu [2], who used the analogous approach to verify 21 of 22 routines in the Berkeley C String Library. He compiled the C source code with `gcc` and verified the binary machine code formally and mechanically with respect to an operational semantics for the Motorola MC68020. His MC68020 model is analogous to our M6. He used the predecessor of the ACL2 theorem prover. The abstractions of Java are so well reflected in JVM bytecode – compared to the abstractions of C versus MC68020 machine code – that we believe Java verification tools can be built on top of M6. However, this is work in progress and this paper focuses primarily on the M6 model itself.

Finally, by developing an *executable* formal semantics we provide a semantics that can be subjected to conventional validation testing on substantial Java test suites. Experience with integrating formal proofs into the commercial hardware design process indicates that it is advantageous to subject formal specifications to thorough testing. For example, Advanced Micro Devices, Inc., uses ACL2 to verify properties of microprocessor components. All the elementary floating point operations on the AMD Athlon™ have been verified with ACL2 to be compliant with the IEEE floating point standard [21]. Of course, the proofs are actually about *formal models* of the circuits; these models are mechanically derived from the actual RTL code for the circuits. Before attempting to prove the models

[2] It is more interesting to prove properties of the JVM specification and properties of Java programs independently of particular JVM implementations. However the JVM specification is difficult to formalize directly and correctly. It is easier to formalize a concrete JVM realization that implements the mechanisms described by the JVM specification. One can still study properties of the JVM specification by studying properties of one concrete JVM implementation. This is the approach that we have adopted in this work.

correct, AMD tested the models against their in-house RTL simulator. Eighty-million floating point test vectors were run through the ACL2 model (of the floating point square root module) and the AMD simulator. The outputs were equivalent, bit by bit. Having thus established the credibility of the RTL-to-ACL2 translation process, AMD undertook the verification of each elementary floating point module and found four flaws (not detected by the test suite) that caused them to change the designs. The final designs were mechanically proved before the Athlon was first fabricated.

As for our executable JVM model, we hope to achieve a simulation speed of millions of instructions/second based on Greve and Wilding's experience using ACL2 to model the Rockwell Collins JEM1, the world's first Java direct-execution microprocessor [9,8,22]. Their optimized simulator of the JEM1 chip in ACL2 runs almost as fast as the original C simulator (at 90% of C speed) at Rockwell Collins. They replaced the C simulator back-end in their program development environment with the ACL2 model and users of the program-debugging front-end were unaware of the change. Similar optimization of M6 has not yet been done but we are confident that ACL2 supports the production of high speed analyzable models.

The organization of the paper is as follows. Section 2 describes our approach for implementing the JVM simulator. Section 3 presents some highlights of our JVM model as a simulator. Space does not permit a thorough description of the model so we focus on just one aspect in detail, namely class initialization. We also discuss briefly a simulation run of a multi-threaded program as an example. Section 4 describes some properties we have established analytically and some we are exploring. We conclude after a discussion of related work.

2. Approach

To study the properties of the JVM and Java programs, our research group has developed a series of JVM models at different abstraction levels. They are written in the ACL2 specification language to allow both executability of the model and the suitability for reasoning with the ACL2 system. Unlike our previous JVM models in the series [18,16,17], the emphasis of the one presented here is on the completeness of the model. We expect to run a suitable subset of some conformance test suite at some point. We intend it to pass a "comprehensive" Java test suite. It executes the complete set of instructions of a JVM, except for floating point arithmetic operations. We also provide the implementation of a subset of Java native methods that appears in the Java runtime library to allow execution of a wide range of realistic Java programs. Complicated aspects such as exception handling, dynamic class loading and class initialization are also carefully modeled. The simulator is implemented in 10K lines of Lisp code in 28 modules and the source code is online at [13].

In the rest of this section, we sketch the simulator implementation. We also comment on our experience in writing a JVM in a functional programming language.

The JVM interpreter loop is modeled with a Lisp function, `run`, which takes as its input a "schedule" and a Lisp representation of the JVM state and returns the state obtained by stepping individual threads as specified by the schedule. The semantics of each instruction

is given by a corresponding state transition function. Primitives for class resolution, class loading, exception propagation as well as Java native methods are also modeled with respective Lisp functions. We did not model the complexity involved in the Java memory model. We assume memory read and write primitives are atomic.

2.1. State representation

Because our simulator is written in a functional programming language, it is quite different from most simulators written in C. All aspects of the machine state are encoded explicitly in one logical object denoted by a term.

A JVM state in our simulator is a seven-tuple consisting of a global program counter, a current thread register, a heap, a thread table, an internal class table that records the runtime representations of the loaded classes, an environment that represents the source from which classes are to be loaded, and a fatal error flag used by the interpreter to indicate an unrecoverable error.

The thread table is a table containing one entry per thread. Each entry has a slot for a saved copy of the global program counter, which points to the next instruction to be executed when this thread is scheduled next time. Among other things, the entry also records the method invocation stack (or "call stack") of the thread. The call stack is a stack of frames. Each frame specifies the method being executed, a return pc, a list of local variables, an operand stack, and possibly a reference to a Java object on which this invocation is synchronized.

The heap is a map from addresses to instance objects. The internal class table is a map from class names to descriptions of various aspects of each class, including its direct superclass, implemented interfaces, fields, methods, access flags, and the bytecode for each method.

All of this state information is represented as a single Lisp object composed of lists, symbols, strings, and numbers. Operations on state components, including determination of the next instruction, object creation, and method resolution, are all defined as Lisp functions on these Lisp objects.

Below we describe the representations of selected state components in some detail.

Objects. Each Java object in the heap has three components:

- A *common-info* section that contains a hash code for this object, a monitor for the JVM and Java programs to synchronize on and the type of this object.
- A *specific-info* section to indicate whether this object is a regular object or an object that represents a class, and, if so, which class it represents.
- A *java visible* section to record the information directly visible to a bytecode program and that can be accessed with instructions such as `getfield`, `putfield`.

For example, the following is an object of type `java.lang.String`:

```
(OBJECT
  (COMMON-INFO 0 (MONITOR ...)
                 "java.lang.String")
  (SPECIFIC-INFO STRING)
```

```
(("java.lang.String" ("value" . 89)
                     ("offset" . 0)
                     ("count" . 4))
 ("java.lang.Object")))
```

The hash code is 0. The "java visible" part is a list of immediate fields from `java.lang.String` and the immediate fields from its superclasses, in this case, `java.lang.Object`.

We need the information stored in these three components to implement primitives for "putfield", "getfield" and monitor enter/exit operations. The information is also needed to implement the native methods such as the `getClass` method of `java.lang.Object`,

Thread table entry. We have described the structure of the thread table informally at the beginning of this section. Each thread table entry has slots for recording a thread id, a pc, a call stack, a thread state, a reference to the monitor, the number of times the thread has entered the monitor, and a reference to the Java object representation of the thread in the heap.

As a concrete example, the following entry is taken from an actual thread table when we use our model to execute a multi-threaded program for computing factorials. A semicolon (;) begins a comment extending to the end of the line.

```
(THREAD 0                  ; thread id is 0
 (SAVED-PC . 0)            ; slot for saved pc
 (CALL-STACK
  (FRAME (RETURN_PC . 7)   ; pc to return to
         (OPERAND-STACK)   ; empty operant stack
         (LOCALS 104)
         (METHOD-PTR "FactHelper" "<init>" ...)
         (SYNC-OBJ-REF . -1))
  (FRAME (RETURN_PC . 18)
         (OPERAND-STACK 104)
         (LOCALS 102)
         (METHOD-PTR "FactHelper" "compute"...)
         (SYNC-OBJ-REF . -1))
   ...)
 (STATUS THREAD_ACTIVE)    ; thread state
 (MONITOR . -1)            ; lock
 (MDEPTH . 0)              ; entering count
 (THREAD-OBJ . 55))        ; object rep in heap
```

Method representation. We have developed a tool, *jvm2acl2*, which takes Java class files and converts them into a format to be used with our model. Each class file is converted into a Lisp constant. The environment component of a JVM state contains an external class table, which is composed of a list of such Lisp constants. The class loader in our JVM model reads from the external class table and constructs a runtime representation of the class in the internal class table. The following is an example to

illustrate how the Java method is represented in the internal class table of our state representation.

```
public static String valueOf(char data[]) {
       return new String(data);
}
```

is represented as

```
(METHOD
  "java.lang.String"
  "valueOf"
  (PARAMETERS (ARRAY CHAR))
  (RETURNTYPE . "java.lang.String")
  (ACCESSFLAGS *CLASS* *PUBLIC* *STATIC*)
  (CODE (MAX_STACK . 3)
        (MAX_LOCAL . 1)
        (CODE_LENGTH . 9)
        (PARSEDCODE
         (0 (NEW (CLASS "java.lang.String")))
         (3 (DUP))
         (4 (ALOAD_0))
         (5 (INVOKESPECIAL
              (METHODCP "<init>"
                        "java.lang.String"
                        ((ARRAY CHAR)) VOID)))
         (8 (ARETURN)
         (ENDOFCODE 9))
        (EXCEPTIONS)
        (STACKMAP)))
```

2.2. State manipulation primitives

Our JVM state is represented explicitly as a Lisp object. We define a set of primitives for manipulating the object. Many of those primitives correspond to the operations and procedures described in the JVM specification [12]. Some of them correspond to the native methods in the Java runtime library that are implementation dependent. Others are utilities that we introduce for implementing our simulator.

For example, to find the next instruction for execution in state `s`, we use the function `next-inst` defined below. It takes `s` as its only formal parameter. It then binds three local variables. The variable `ip` is bound to `(pc s)`. In Lisp, `(pc s)` denotes the value of the Lisp function `pc` when applied to `s`, i.e., $pc(s)$. Next the variable `method-ptr` is bound to the method pointer of the top frame of the call stack of the current thread, as determined by the function `current-method-ptr`. Finally, the variable `method-rep` is bound to the method referred to by `method-ptr`; we use `deref-method` to look up the method in the instance class table. Having bound the three local variables, we then use

`inst-by-offset` to determine the instruction at offset `ip` in the method `method-rep` and return the instruction.

```
(defun next-inst (s)
  (let* ((ip  (pc s))
         (method-ptr (current-method-ptr s))
         (method-rep
            (deref-method method-ptr
                        (instance-class-table s)))
       (inst-by-offset ip method-rep)))
```

As another example, the following function is used in the descriptions of the JVM instructions that invoke methods.

```
 (defun call_method_general
       (this-ref method s0 size)
   (let ((accessflags
          (method-accessflags method))
         (s1
          (state-set-pc (+ (pc s0) size) s0)))
     (cond
      ((mem '*native* accessflags)
       (invokeNativeFunction method s1))
      ((mem '*abstract* accessflags)
       (fatalError "abstract_method invoked" s0))
      (t
       (let ((s2 (pushFrameWithPop ... s1 ...)))
         (if (mem '*synchronized* accessflags)
             (mv-let
              (mstatus s3)
              (monitorEnter this-ref s2)
              (set-curframe-sync-obj this-ref s3))
           s2))))))
```

We read this roughly as follows. The function has four formals: `this-ref`, the current "self"-reference; `method`, the method to invoke; `s0`, the current state; and `size`, the length in bytes of the invocation instruction. Let `accessflags` be the access flags of the method and let `s1` be a state like `s0` but with the program counter incremented by `size`. If `accessflags` indicates that the method is native or abstract, we handle the call with special-purpose functions. Otherwise, let `s2` be a state obtained from `s1` by popping the proper number of values off the current operand stack, and pushing a suitable call frame on top of the current call frame. We construct this state with `pushFrameWithPop`. If `accessflags` indicates the invoked method is synchronized, we call `monitorEnter` to obtain two results, `mstatus` (which we ignore) and a state `s3`, in which the current thread holds the monitor on `this-ref`; we return `s3` after updating the new current frame to indicate that the monitor on `this-ref` must be released upon exit from the method. If the invoked method is not synchronized, we return `s2`.

Notice that monitorEnter mentioned above is just another state manipulating primitive that takes a reference and a state and returns a status flag and a state. There are two kinds of states that can be returned by monitorEnter. If monitorEnter succeeds, the state returned is representing a state in which the current thread is holding the monitor and the current thread is still active. If monitorEnter does not succeed — because some other thread is holding the monitor — then the state returned represents a state in which the current thread is suspended and moved to the waiting queue for the monitor.

As our last example in this section, we introduce the function implementing the native method currentThread of java.lang.Thread. It takes a state as its input and pushes on the operand stack of the current frame a reference to the Java object that represents the current thread.

```
(defun Java_java_lang_Thread_currentThread (s)
 (let*
  ((tid (current-thread s))
   (thread-rep
    (thread-by-id tid (thread-table s)))
   (thread-ref (thread-ref thread-rep)))
  (pushStack thread-ref s)))
```

2.3. State transition function

We model the semantics of the JVM instructions operationally, in terms of such primitives as are discussed above. The meaning of executing a JVM instruction is given by a state transition function on JVM states. Here is the state transition function for the IDIV instruction.

```
(defun execute-IDIV (inst s)
 (let ((v2 (topStack s))
       (v1 (secondStack s)))
  (if (equal v2 0)
      (raise-exception
          "java.lang.ArithmeticException" s)
   (ADVANCE-PC
     (pushStack (int-fix (truncate v1 v2))
                (popStack (popStack s)))))))
```

Here, inst is understood to be a parsed IDIV instruction. ADVANCE-PC is a Lisp macro for advancing the global program counter by the size of the instruction. PushStack pushes a value on the operand stack of the *current frame* (the top call frame of the current thread) and returns the resulting state. When the item on the top of the operand stack of the current frame is zero, the output of execute-IDIV is a state obtained from s by raising an exception of type java.lang.ArithmeticException. If the top item is not zero, the resulting state is obtained by changing the operand stack in the current frame and advancing the program counter. The operand stack is changed by pushing a certain value (described below) onto the result of popping two items off the initial operand stack. The value pushed

is the twos-complement integer represented by the low order 32 bits of the integer quotient of the second item on the initial operand stack divided by the first item on it.

For a more complicated instruction, here is our semantics for `invokestatic`.

```
(defun execute-invokestatic (inst s)
 (let* ((cp    (arg inst))
        (method-ptr (methodCP-to-method-ptr cp)))
  (mv-let
   (method new-s)
   (resolveMethodReference method-ptr t s)
   (if method
       (let*
        ((class
          (static-method-class-rep method new-s))
         (cname
          (classname class))
         (cref
          (class-ref class)))
        (cond
         ((class-rep-in-error-state? class)
          (fatalError "class in error state!"
                      new-s))
         ((not (class-initialized? cname  new-s))
          (initializeClass cname new-s))
         (t
          (call_static_method cref method new-s))))
     (fatalSlotError cp new-s)))))
```

To invoke a static method, we first get the constant pool entry and obtain a symbolic `method-ptr`. Then we call the method resolution procedure `resolveMethodReference` to find the `method` to which the `method-ptr` resolves. Notice that method resolution may cause class loading so the procedure returns a pair `(method new-s)` as declared in the `mv-let`. The fatal error flag is set when the method is `nil`, i.e., the method resolution failed. If the class to which the method belongs has not been initialized, we choose not to advance the program counter, instead returning a state prepared for starting the class initialization procedure using `(initializeClass cname new-s)`. Otherwise, we use `call_static_method` to return a state properly prepared for the interpreter to start executing the resolved method. The definition of `call_static_method` (not shown here) uses the previously shown `call_method_general` to construct that state.

3. A simulator

Not only have we modeled the semantics of most JVM instructions, we have also provided implementations for native methods from the CLDC runtime library. As a result, we can use the formal model as a simulator to execute realistic JVM programs.

We have translated the entire Sun CLDC library implementation into our representation with 672 methods in 87 classes [5]. We provide implementations for 21 out of 41 native methods that appear in Sun's CLDC library. The remaining ones are mostly related to the I/O of the JVM, which we do not model in our current simulator.

We have written several test programs to run on this model to exercise various aspects of the simulator such as exception handling, synchronization, and class initialization. One of the test programs we run is a multi-threaded Java program that implements an impractical but illuminating parallel factorial algorithm.

The program takes a command line parameter represented in a `java.lang.String`. It calls the method `parseInt` of `java.lang.Integer`, which in turn invokes a dozen Java functions, to parse the string and return an integer. Then it spawns a specified number of `FactHelper` threads (five in the current version). Those threads share an instance of `FactJob`, in which the intermediate result is stored. Those `FactHelpers` repeatedly compete for the monitor on the `FactJob`, compute one iteration and then release the monitor by executing `monitorexit`. The main thread prints the result and quits when it is awakened by a `notify` call from any of the `FactHelpers` indicating that the computation has terminated.

```
class FactJob {
    int value = 1;
    int n;
    ... };

class FactHelper implements Runnable {
    FactJob myJob;
    public void run() {
      ... wait for notifyAll on myJob.
      for (;;) {
        synchronized(myJob) {
         if (myJob.n<=0) {
             myJob.notify();
             return; // done
         } else {
             myJob.value = myJob.value*myJob.n;
             myJob.n = myJob.n - 1;
         }}}}
    ...};

public class Fact {
    static int HELPER_COUNT = 5;

    public static int fact(int n) {
        FactJob theJob = new FactJob(n);
        ... spawn HELPER_COUNTER FactHelpers.
        ... send notifyAll to FactHelpers
```

```
         try{
          synchronized (theJob){ theJob.wait();};
          //wait for at least one Helper finishes.
         } catch ...
        return theJob.value;};
}
```

The main method is:

```
public static void main(String[] args) {
    int n = Integer.parseInt(args[0]);
    System.out.println(""+Fact.fact(n));
}
```

To compute 10 factorial, the simulator executes 1748 steps, most of which are spent in the class initialization, parsing an integer from a string, string copying and printing the result character by character. In the process, 5 threads are spawned, 118 heap objects are created, and 18 classes are loaded and initialized. A transcript of this execution is available online.[3]

In the rest of this section, we pick some aspects of our simulator and explain them in some more detail.

3.1. Interpreter loop

As mentioned previously, our JVM model takes a "schedule" (a list of thread ids) and a state as the input and repeatedly executes the next instruction from the thread as indicated in the schedule, until the schedule is exhausted.

```
(defun run (sched s)
 (cond
  ((endp sched)
   s)
  (t
   (let ((nid (car sched))
         (cid (current-thread s)))
     (cond
      ((equal cid nid)
       (run (cdr sched) (step s)))
      (t
       (run
        (cdr sched)
        (loadExecutionEnvironment
         nid
         (storeExecutionEnvironment s)))))))))
```

[3] See `pftrace.lisp` in [13].

The scheduling policy is thus left unspecified. Any schedule can be simulated. However to use the model as a JVM simulator without providing a schedule explicitly, we have implemented some simple scheduling policies. One of them is a not very realistic round-robin scheduling algorithm, which does a rescheduling after executing each instruction.

3.2. Class initialization

Class initialization is a very complicated aspect of the JVM. Multi-threading adds to the complexity. To initialize a class, the JVM needs to execute a special class initialization method as part of the process; thus the initialization of a class cannot be made atomic. The process needs careful synchronization between threads that may try to initialize the same class at the same time. We explain this aspect of our simulator in the most detail.

Modeled after the KVM implementation [5], our simulator follows the 11-step algorithm described in the JVM specification Sec. 2.17.5. [12] and completes a class initialization in six stages. Recall the state transition function that describes `invokestatic`. If the class to which the resolved method belongs has not been initialized the function does not advance the pc; instead it returns a state by calling `(initializeClass classname s)`. Roughly, the state returned is a special state recognized by the interpreter loop as indicating the start of the class initialization process. Each thread maintains a simple finite state machine to keep track of the class initialization stage that the thread is currently in. A series of functions, with names of the form `runCliniti`, implement the transitions of this machine. The details are in `jvm-static-initializer.lisp` of [13].

```
(defun runClinit1 (classname s)
 (let ((cid (current-thread s)))
  (mv-let (mstatus new-s)
          (classMonitorEnter classname s)
          (if (not (equal mstatus
                          'MonitorStatusOwn))
              (set-cinit-stage  cid 2 new-s)
            (runClinit2 classname new-s))))))
```

To initialize a class, the current thread first needs to acquire the monitor associated with the class. This is necessary because we need to cope with the situation that multiple threads may be trying to initialize the same class at the same time. If the acquisition attempt fails, i.e. the `mstatus` is not equal to `MonitorStatusOwn`, we return the following state to the interpreter loop: the original current thread is suspended and the class initialization process stage of the thread is set to 2 with `(set-cinit-stage cid 2 new-s)`. The `set-cinit-stage` call is needed so that when the thread is resumed after JVM grants the monitor, the top level interpreter loop can resume from the second stage of the class initialization process. Otherwise, we directly advance to stage 2.

```
(defun runClinit2 (classname s)
 (let*
  ((class-rep  (class-by-name classname ...))
```

```
 (initThread (init-thread-id class-rep))
 (cid        (current-thread s)))
(cond
 ((and (not (equal initThread -1))
       (not (equal initThread
                   (current-thread s))))
  (let
    ((new-s (classMonitorWaitX classname s)))
   (set-cinit-stage  cid 2 new-s)))
 ((or (equal initThread (current-thread s))
      (class-initialized? classname s))
  ....)
 (t
  (let
   ((sinit
     (setClassInitialThread classname cid s)))
   (mv-let (mstatus exception-name s-new)
           (classMonitorExitX classname sinit)
           (runClinit3 classname s-new)))))))
```

In stage 2, we first check whether some other thread is initializing the class. If some other thread is initializing the class, we return a state in which the current thread is suspended to wait for a notify signal from the monitor and the class initialization stage remains at 2. If no other thread is initializing the class, we check whether the class has already been initialized. If it has not been initialized, we mark the state to indicate that the current thread is initializing the class and return the resulting state to the interpreter loop.

```
(defun runClinit3 (classname s)
  (let ((class-rep (class-by-name classname ...))
        (cid (current-thread s)))
    (if (not (isInterface class-rep))
        (if (and (super-exists class-rep)
                 (not (class-initialized?
                        (super class-rep) s)))
            (initializeClass
                    (super class-rep)
                    (set-cinit-stage cid 4 s))
          (runClinit4 classname s))
      (runClinit4 classname s))))
```

In stage 3 of class initialization, we first check whether the superclass has been initialized. If the superclass is not initialized yet, we use `initializeClass` to prepare a state so that the interpreter loop can recognize and start the initialization process for the superclass. We also properly mark the state, so that when class initialization for the superclass is completed, the process of initializing the current class enters stage 4.

```
(defun runClinit4 (classname s)
  (let* ((clinit-ptr (clinit-ptr classname))
         (thisMethod
            (getSpecialMethod clinit-ptr s))
         (cid (current-thread s)))
    (if thisMethod
        (pushFrame clinit-ptr nil
                   (set-cinit-stage cid 5 s))
      (runClinit5 classname s))))
```

Now we are in class initialization stage 4. To complete the class initialization, the interpreter must execute the class initialization method of the class. We achieve this by pushing a call frame for the class initialization method to the call stack. We also mark the state so that when the interpreter is done with the invocation of the class initialization method, it can recognize that it needs to resume the class initialization process from stage 5. In stage 5, we try to acquire the monitor associated with the class. And in stage 6, the class initialization process is completed; we send the `notifyAll` signal, so that threads that are waiting in stage 2 can proceed.

4. Formal analysis

The focus of our research has been developing a practical methodology for reasoning about complex hardware and software artifacts, including models of the complexity of the one described here. We have shown that our executable JVM model includes enough detail to permit its use as a simulator. But it has dual use as a formal semantics of the JVM and as such permits us to reason about the JVM and its bytecoded methods. It is easy to formulate simpler, less complete models of the JVM (we have a series of less complete models ourselves). Whether there is a simpler semantics that is as complete as this one is an open question as far as we are concerned. Even should a simpler useful semantics be developed, this model is an excellent stepping-stone toward the verification of an implementation of the JVM (e.g., the J2ME KVM [5]) and is worthy of study in that context.

We study two broad categories of the properties using this accurate JVM model. We use our JVM model to study the properties of the JVM specification and the J2ME KVM implementation. We also use the formal model to study the properties of Java programs that run on "real" (practically efficient) JVMs. In the rest of the section, we present some work we have done in these two categories.

4.1. Properties of the model

We are interested in studying dynamic class loading in the JVM. We have formally proved an invariant of dynamic class loading in our JVM model. The invariant says that if class A is loaded, and if a value of type class A is assignable to a slot of type class B, then class B must be already correctly loaded. As a relevant note, the concept of *assignable to* is rather complicated. For a value of type class A to be assignable to a slot of type class B,

B must be a direct superclass of A, one of the direct superinterfaces of A, or there must be a class C, where values of type class C are assignable to a slot of type class B, and class C must be a direct superclass of class A or one of the direct superinterfaces of class A.

The JVM relies on this invariant about loaded classes to behave correctly in field resolution and method dispatching. In fact, JVM operations that involve examining the class hierarchy by searching through the superclass chain rely on this property to operate correctly. It is useful to be sure that this alleged "invariant" is in fact preserved by the class loader in the JVM.

We have proved that this invariant is preserved by the class loader in this formal model of the JVM. Our proof has two main steps. First, we proved a different formulation of the invariant, which roughly says that all classes reachable from any loaded class through its superclass chain and superinterface chains are also loaded. The reformulated property is named `load-inv` and its definition is built up in several levels. We first defined a Lisp function `collect-assignableToName` that crawls along the class hierarchy and collects all classes reachable from a given one. We use that concept to express the idea, formalized as `loader-inv-helper1`, that all classes reachable from a given one are correctly loaded.

```
(defun loader-inv-helper1
   (class-rep class-table env-class-table)
   (let* ((classname (classname class-rep))
          (supers (collect-assignableToName
                    classname env-class-table)))
     (all-correctly-loaded?
          supers class-table env-class-table)))
```

Finally, we defined the reformulated alleged invariant property, `loader-inv`, to check `loader-inv-helper1` for every class in the class table. To establish that `load-inv` is invariant[4] we proved the following theorem.

```
(defthm loader-inv-is-inv-respect-to-loader
  (implies (loader-inv s)
           (loader-inv (load_class classname s)))
```

This theorem has one hypothesis and one conclusion. The hypothesis says that `s` satisfies the `loader-inv` property. The conclusion says that the state produced by `load_class` satisfies it.

The second step of our proof is to show that the above reformulated property implies our original one.

```
(defthm inv-and-isAssignableTo-env
  (implies
    (and (loader-inv s)
         (no-fatal-error? s)
```

[4] To prove that `load-inv` is an invariant of any execution, we need to show that it is preserved not only by `load_class` primitive, but also by all machine steps. We have not done that yet.

```
    (isAssignableTo-env A B
              (env-class-table (env s))))
(implies (correctly-loaded? A
              (instance-class-table s)
              (env-class-table (env s)))
         (correctly-loaded? B
              (instance-class-table s)
              (env-class-table (env s)))))
```

A more ambitious project that uses this formal JVM model is studying the adequacy of the current CLDC bytecode verification algorithm as described in JSR-139 for J2ME JVMs [6]. We want to show that bytecode passed by CLDC's bytecode verification algorithm will behave as expected.

Our approach to the ultimate goal of verifying a bytecode verifier is taken from Cohen's original work [4]: we introduce a "defensive" version of M6 that checks for "all possible" runtime anomalies. We will use ACL2's "guard verification" facility to insure that the defensive machine respects the pre-conditions of all primitive operations. When the defensive machine detects an unacceptable state it stops and "waves a red flag".[5] Our goal is to prove two things. First, the defensive machine is equivalent to M6 when the red flag is not waved. Second, if the defensive machine is running CLDC bytecode verified code, the red flag is not waved. These properties are much more complicated than indicated here and involve the characterization of "tagged" states (where data objects carry information about their types), abstract "signature states" consisting primarily of the tags alone (akin to those states visited during bytecode verification), the mappings between tagged and untagged states, the commutativity of these mappings with the respective step functions, and the characterization of the "consistent" states of the defensive machine.

We already have some partial results about the correctness of the bytecode verification algorithm on one of our simpler JVM models, M3. For example, we have proved that executing any program on a regular JVM (M3) computes the "same" result as executing it on a corresponding defensive machine, when the red flag is not waved. We also proved that any execution of the defensive JVM preserves a *consistent-state* predicate, which implies that starting from a consistent initial state, a defensive JVM always has well-formed heaps, appropriate stacks, pointers only to valid objects, etc. We also proved that the original iterative bytecode verification algorithm terminates on all bytecode programs.

4.2. Properties of bytecode programs

Besides reasoning about the properties of the JVM model itself, we use the model to reason analytically about the programs that run on it. This gives us opportunities for finding program flaws beyond those likely to be uncovered by simulation of executions.

For example, we have proved that a single-threaded program for computing factorials is correct on M6. The property is stated with the following ACL2 formula. The ACL2

[5] We use such a colloquial expression to avoid confusing this situation with JVM errors conditions, exceptions, etc.

system mechanically checked our proof of the theorem. We note that this is only to illustrate one style of proofs. This particular style uses an explicit clock function. This proof methodology works well for proving total correctness for single-threaded programs. However one is not limited to this particular style. Ray and Moore recently showed that one can use the different styles in verifying different parts of programs. They present a mechanical process for converting a proof in one style that uses clock function to a proof in another style that needs no clock function, and vice versa [20].

```
(defthm factorial-is-correct
 (implies
   (and (poised-to-execute-fact s)
        (integerp n)
        (<= 0 n)
        (intp n)
        (equal n (topStack s)))
   (equal (simple-run s (fact-clock n))
          (state-set-pc
           (+ 3 (pc s))
           (pushStack (int-fix (fact n))
                      (popStack s))))))
```

This theorem has five hypotheses. The first, `(poised-to- execute-fact s)`, is a rather complicated predicate whose definition is not shown here. It says that the state `s` is well formed and that the next instruction will invoke a particular bytecoded factorial method produced by compiling with `javac` a recursive `int`-valued Java definition of factorials. The other hypotheses say that the value, `n`, on top of the operand stack is a non-negative `int`. The conclusion is an equality between two expressions denoting M6 states. The first expression denotes the state produced by starting in `s` and stepping the single thread in question a certain number of steps, namely, the number produced by `(fact-clock n)`. The second expression denotes the state obtained from `s` by incrementing the `pc` by 3 (the size of the invoke instruction), popping one item (namely `n`) from the operand stack, and pushing the `int` denoted by the low order 32 bits of the mathematically precise `n`! (written in ACL2 as `(fact n)`).

In formal methods nomenclature, "partial" correctness specifies that a method returns an acceptable answer *if* it terminates; "total" correctness adds to that the proof requirement *that* the method terminates. The theorem above is a specification for total functional correctness of the corresponding bytecode factorial method. Indeed, we characterize, with `(fact-clock n)`, exactly how many bytecode instructions must be executed. This theorem was proved mechanically with ACL2. Partial correctness results can also be proved [15]. An explicit clock function is not needed and some time could not be defined. ACL2 proofs that employ explicit *clock functions* and proofs that employ no such clock function constructs can be used together to reason about different parts of the same program [20].

Most of the JVM bytecode proofs we have produced to date have used the predecessor JVM model, M5, which does not support exceptions, class initialization or dynamic class loading. As noted, our focus recently has been on developing the more complete M6.

But using M5 we have proved the correctness (sometimes partial, sometimes total) of methods that create and alter objects in the heap. For example, we proved the total correctness of an applicative Java insertion sort method [14] that takes (a pointer to) a linked list object in the heap and returns (a new heap and a pointer to) a linked list object representing an ordered permutation of the original list.

We have also proved theorems about multi-threaded Java programs that synchronize through a monitor to gain exclusive access to the shared resources [17]. In one example, an unbounded number of threads repeatedly compete for a lock on an object to increase a counter in the object. We proved that the counter goes up monotonically until it wraps around because of an overflow.

Ray and Moore investigated different proof styles for reasoning program executions on machine models with an operational semantics [20]. They show that one can use clock style (as illustrated in the factorial proof example) and inductive invariant style proofs compositionally. Different parts of a program can be verified respectively with proofs of different styles. Proofs from one style can be mechanically converted into proofs of the other style to combine with other proofs in that style.

Our group has proved theorems relating program execution on one model to execution on another, simpler model. One such example is reported in [19] where the theorem allows us to move from a multi-threaded model to a single-threaded model for programs that are, in some suitable sense, inherently single-threaded. The final theorem in that paper basically says that for any program that satisfies a certain syntactic restriction, the effects of executing it on the multi-threaded JVM model can be obtained by executing the program on a single-threaded JVM. We intend to prove such reduction theorems for this model as well, to simplify code proofs for certain classes of methods.

5. Related work

Cohen uses ACL2 to formalize a single-threaded "defensive" machine [4]. His emphasis is on using a formal language to precisely specify the correct behaviors of the JVM. His work inspired us to develop a series of models for studying the various aspects of the JVM formally.

Unlike Cohen's work, we model multi-threading, synchronization, class loading and class initialization, as well as exception handling. We also provide implementations for 21 of 41 native methods to allow the execution of a wide range of realistic Java programs. The model can be used as a realistic JVM simulator.

The simpler single-threaded Java Card Virtual Machine was modeled at INRIA, Sophia-Antipolis, France [1]. They give a formal executable semantics for the Java Card Virtual Machine in Coq [7], an ML-like language. They claim the model is quite complete and can execute any JavaCard program — given that they provide the native implementations of necessary library functions. However it is not clear whether they have actually provided the native implementation.

Work in progress in our group includes both the study of the bytecode verifier and the development of tools for easing the human burden of producing proofs of bytecode programs. We have formalized the CLDC bytecode verifier algorithm as described in [6]

with the aim of mechanically proving that verified programs never cause anomalies on our JVM model.

Another member of our research group, Jeff Golden, is developing a methodology for systematically reasoning about the behavior of programs on our formal JVM models. As part of his research, he is developing facilities to support symbolic execution of bytecode. We believe that symbolic execution can be helpful in catching software bugs. In addition, he is developing tools to produce mechanically from bytecode the "functional semantics" described in ACL2. It is our intention that these tools will also generate machine-checked proofs of the correctness of that semantics with respect to our operational models. Thus, the problem of verifying a bytecoded method will be reduced to proving a theorem about a simpler (albeit still complicated) ACL2 function. The two projects mentioned above – the CLDC bytecode verifier work and the automated semantics work – are works in progress.

6. Conclusion

In this paper, we describe several selected aspects of our realistic JVM simulator. We presented our general approach for writing such a detailed JVM model in a functional programming language. The state is represented with a Lisp object. The semantics of executing a bytecode instruction in a JVM state is defined by a state transition function. We show that this approach allows us to model the delicate internals of a realistic JVM, use it as a simulator, and prove theorems about the resulting behaviors.

This executable JVM model has been the basis for our ongoing inquiry into the correctness of the bytecode verifier specification. We are still gaining experience in reasoning about such a realistic (and thus complicated) software system.

Although a realistic JVM model is necessary for studying the correctness of the bytecode verifier and the class loader, proving properties of concrete Java programs on this complicated JVM model may be difficult and unnecessary. This can be one of the limitations. After proving the properties of the bytecode verifier and the class loader, we may be able to reduce executions on this model to executions on an alternative simpler model via a proof. Porter has done a similar proof for relating a multi-threaded JVM model to a single-threaded model under necessary restrictions [19]. We will need to look into the proper thread reduction and heap abstraction techniques.

This model, although fairly complete, still contains certain omissions and simplifying assumptions. The following two might be of particular interest.

Our JVM simulator assumed the simplest memory model. Any memory access in our simulator is always atomic at the instruction level. Although we expect that such an omission would not affect our study on the correctness of the bytecode verifier, we realize that such an omission prevents us from reasoning about certain behaviors of legal Java programs.

The other limitation is that our existing model of the bytecode verifier does not interact with the class loader dynamically. The JVM specification allows a JVM implementation to delay the bytecode verification until link time. In our implementation, the bytecode verifier uses the class hierarchy information from "unloaded" classes to conduct the bytecode verification of a method. In fact, the bytecode verifier never causes dynamic class loading

itself. We expect to be able to prove that the bytecode verification result is independent of the class table being used, as long as the runtime class table is correctly loaded from the "environment". We also look forward to proving that the runtime class table is always correctly loaded from the environment if the JVM is always executing bytecode verified code.

In summary, our approach of writing an executable JVM simulator in ACL2, a precise language with clean axiomatic semantics and a computer aided deduction environment, provides an opportunity for analytically deducing the properties of the artifact being modeled. It brings the benefits and new opportunities of both simulation and machine-checked analytical reasoning. We are researching ways for making better use of such opportunities.

Acknowledgments

We thank the many people in the ACL2 user community, the ACL2 Research Group in Austin, and undergraduate students in Moore's class *Formal Model of the Java Virtual Machine* who contributed to this, the sixth machine in an evolving sequence of operational formal models of the JVM. In addition, we thank Sun Microsystems, Inc., for their support of our research.

References

[1] G. Barthe, G. Dufay, L. Jakubiec, B. Serpette, S. Melo de Sousa, A formal executable semantics of the JavaCard platform, in: D. Sands (Ed.), Proceedings of ESOP'01, 2001.
[2] R.S. Boyer, Y. Yu, Automated proofs of object code for a widely used microprocessor, Journal of the ACM 43 (1) (1996) 166–192.
[3] D. Burger, T.M. Austin, The SimpleScalar tool set, Version 2.0, Technical Report 1342, University of Wisconsin-Madison Computer Science Department, June 1997.
[4] R. Cohen, Defensive Java Virtual Machine Version 0.5 alpha Release. Available from http://www.cli.com/software/djvm/index.html, 1997.
[5] Connected Limited Device Configuration (CLDC) and the K Virtual Machine. http://java.sun.com/products/cldc/.
[6] Connected Limited Device Configuration (CLDC) Specification 1.1. http://jcp.org/en/jsr/detail?id=139.
[7] The Coq Development Team, The Coq Proof Assistant reference manual. Available from http://coq.inria.fr/, 2004.
[8] D. Greve, Symbolic simulation of the JEM1 microprocessor, in: G. Gopalakrishnan, P. Windley (Eds.), Formal Methods in Computer-Aided Design, FMCAD'98, Palo Alto, CA, 1998, pp. 321–333.
[9] D. Hardin, D. Greve, M. Wilding, J. Cowles, Single-threaded formal processor models: Enabling proof and high-speed execution, Technical Report, Rockwell Collins Advanced Technology Center, Cedar Rapids, IA 52498, 1999.
[10] Java 2 Platform, Micro edition. http://java.sun.com/j2me/.
[11] M. Kaufmann, P. Manolios, J.S. Moore, Computer-aided Reasoning: An Approach, Kluwer Academic Publishers, 2000.
[12] T. Lindholm, F. Yellin, The Java Virtual Machine Specification, 2nd edition, Addison-Wesley Publisher, 1999.
[13] H. Liu, J.S. Moore, JVM model: M6 source code. http://www.cs.utexas.edu/users/hbl/pub/M6/ivme03/, 2003 March.
[14] J.S. Moore, Proving theorems about Java and the JVM with ACL2, in: M. Broy (Ed.), in: Lecture Notes of the Marktoberdorf 2002 Summer School, Springer, 2002.

[15] J.S. Moore, Inductive assertions and operational semantics, in: D. Geist (Ed.), Proceedings of CHARME 2003, in: Lecture Notes in Computer Science, Springer Verlag, 2003, pp. 289–303.
[16] J.S. Moore, G. Porter, An executable formal java virtual machine thread model, in: Proceedings of 2001 JVM Usenix Symposium, Monterey, California, April 2001. USENIX.
[17] J.S. Moore, G. Porter, The apprentice challenge, ACM Transactions on Programming Languages and Systems (TOPLAS) 24 (3) (2002) 193–216.
[18] J.S. Moore, R. Krug, H. Liu, G. Porter, Formal models of Java at the JVM level a survey from the ACL2 perspective, in: Workshop on Formal Techniques for Java Programs, 2001.
[19] G. Porter, A commuting diagram relating threaded and non-threaded JVM models, Technical Report CS-TR-01-27, Honors Thesis, Department of Computer Sciences, University of Texas at Austin, 2001.
[20] S. Ray, J.S. Moore, Proof styles in operational semantics, in: Proceedings of FMCAD'04, Austin, TX, 2004 (in press).
[21] D. Russinoff, A mechanically checked proof of IEEE compliance of a register-transfer-level specification of the AMD-K7 floating-point multiplication, division, and square root instructions, London Mathematical Society Journal of Computation and Mathematics 1 (December) (1998) 148–200. http://www.onr.com/user/russ/david/k7-div-sqrt.html.
[22] M. Wilding, D. Greve, D. Hardin, Efficient simulation of formal processor models, Formal Methods in System Design 18 (3) (2001).