Executable JVM Model for Analytical Reasoning: A Study

Hanbing Liu
Department of Computer Science
University of Texas at Austin
hbl@cs.utexas.edu

J Strother Moore
Department of Computer Science
University of Texas at Austin
moore@cs.utexas.edu

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[16] implementation.

On the one hand, our JVM model is a conventional machine emulator. M6 models accurately almost all aspects of the KVM implementation, including the dynamic class loading, class initialization and synchronization via monitors. It executes most J2ME Java programs that do not use any I/O or floating point operations. Engineers may consider M6 an implementation of the JVM. It is implemented with around 10K lines in 20+ modules.

On the other hand, M6 is a novel model that allows for analytical reasoning besides 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. A property of M6 can be expressed as a formula. Rules of interference can be used analytically to derive properties of M6 and the Java programs that run on the model, using a mechanical theorem prover.

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.

Categories and Subject Descriptors

D.2 [**Software**]: Software Engineering; D.2.4 [**Software Engineering**]: Software/Program Verification [Formal methods, Validation, Correctness proofs]; F.3 [**Logics and Meanings of Programs**]: Specifying and Verifying and Reasoning about Programs; F.3.1 [**Logics and Meanings of Programs**]: Specifying and Verifying and Reasoning about Programs [Specification techniques]

IVME'03, June 12, 2003, San Diego, California, USA.

1. INTRODUCTION

Simulators have 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.

Most simulators of hardware artifacts are either used for detecting flaws in the designs of the artifacts or flaws in the software to be used with those physical artifacts before they are built.

It is our observation that those 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. Different from 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 C implementation of the Java virtual machine KVM[16]. It supports almost all interesting features of the J2ME JVM[18], including class loading, class initialization, synchronization via monitors as well as exception handling. We are still working towards modeling the access permissions checking accurately, as well as floating point operations.

We hope to achieve a simulation speed of millions of instructions/seconds based on Greve and Wilding's experience with using ACL2 to efficiently model Rockwell Collins JEM1, the world's first Java direct-execution microprocessor[4, 6, 12]. 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 actually used it to replace the

C simulator back-end in their program development environment, and users of the program-debugging front-end are unaware of the change. Similar optimization of our JVM model in ACL2 has not been done yet.

In addition, we can also treat the JVM simulator as a set of formulas in the ACL2 specification language and we can reason about the JVM model analytically with axioms and rules of inference, which define the meaning of the syntactic constructs and evaluation rules. We use the computer aided reasoning tool, ACL2, to help us produce mechanically checked proofs of the properties of the model. In fact, this aspect has been our group's ultimate focus, while building a complete and accurate JVM model is needed as a firm base for stating the properties of interests.

Our approach is different from conventional simulator designs because it provides support for reasoning analytically about the model and programs that run on the model.

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. For example, we explain how we model class initialization in some detail in this section. Section 4 describes some proofs for properties we established analytically. Then, we conclude after a discussion of related work.

2. M6: 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. Different from our previous JVM models in the series [10, 9, 14], the emphasis of the one presented here is on the completeness of the model. 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 APIs to allow executions 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 20+ modules and the source code is online at [8].

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 APIs are also modeled with respective Lisp functions.

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 byte code 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 generic 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 Java 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 super classes, 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 APIs such as the getClass method of java.lang.Object,

Thread table entry We have described the structure of the thread table informally in 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 factorial. 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, *jvm2-acl2*, 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 logical object. We define a set of primitives to manipulate the object. Many of those primitives correspond to the operations and procedures described in the JVM specification[19]. Some of them correspond to the native APIs 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 following primitives. We first use the primitive current-method-ptr to find method-ptr – the method pointer associated with the top frame of the call stack of the current thread, then we use deref-method to look up the the method referred by method-ptr in the instance-class-table of s, and we return the instruction at offset (pc s) of the method. In Lisp, (pc s) denotes the value of the Lisp function pc when applied to s, i.e., $pc(s)$.

```
(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 our internal primitive for invoking a Java method.

```
(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 method ...)))
           (if (mem '*synchronized* accessflags)
```

```
                    (mv-let (mstatus s3)
                            (monitorEnter this-ref s2)
                      (set-curframe-sync-obj this-ref s3))
                  s2))))))
```

To invoke a synchronized non-native, non-abstract method, we first need to pop the proper number of values off the current operand stack, and push a suitable call frame on top of the current call frame. We achieve these effects with pushFrameWithPop. Then, we call monitorEnter to obtain the monitor on the this-ref. We mark the call frame to indicate this invocation has the monitor on this-ref. The resulting state is returned so that the top level interpreter loop can proceed.

Notice 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 the 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 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 following Lisp function to implement the native method currentThread of java.lang.Thread. It takes a state as its input and pushes a reference to the Java object that represents current thread on the operand stack of the current frame.

```
(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 discussed above. The meaning of executing a JVM instruction is given by a state transition function on the 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 to advance 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 and 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)))
      (if (class-rep-in-error-state? class)
          (fatalError "class in error state!" new-s)
       (if (not (class-initialized? cname  new-s))
           (initializeClass cname new-s)
         (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 that the method-ptr resolves to. Notice that the 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.

3. M6: A SIMULATOR

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

We have translated the entire Sun CLDC API library implementation into our representation with 672 methods in 87 classes[16]. We provide implementations for 21 out of 41 native APIs that appear in Sun's CLDC API 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, 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 API functions, to parse the string and return an integer. Then it spawns a specified number of FactHelper threads (5 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 factorial of 10, 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 object are created, and 18 classes are loaded and initialized. A transcript of this execution is online.[1]

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

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, 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[16], our simulator follows the 11-step algorithm described in the JVM specification Sec. 2.17.5.[19] and completes a class initialization in 6 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. The complete details are in jvm-static-initializer.lisp of [8].

```
(defun runClinit1 (classname s)
 (let ((cid (current-thread s)))
```

[1]See pftrace.lisp in [8]

```
(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 (as a consequence of failing to acquire the monitor) and the class initialization process stage of the thread is set to 2 with (set-cinit-stage cid 2 new-s).

```
(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 super class has been initialized. If the super class 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 super class. We also properly mark the state, so that when class initialization for the super class 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 initilization method to the call stack. We also mark the state so that when the interpreter is done with the invocation of the class initilization method, it can recognize that it needs to resume the class initilization 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 notifyAll signal, so that threads that are waiting in stage 2 can proceed.

4. M6: A FORMAL MODEL

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 accurate models of the JVM (we have a series of less accurate models ourselves). Whether there is a simpler semantics that is as accurate 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[16] from which it was produced) 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.

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 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 super interfaces 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 super interfaces of class A.

The JVM relies on this invariant about loaded classes to behave correctly in field resolution and method dispatching. In fact, any operations that involves examining the class hierarchy by searching through super class 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 this invariant is preserved by the class loader in this formal model of the JVM. We achieve this in two steps. As the first step, we proved a different formulation of the same invariant, which roughly says, all classes reachable from any loaded class through its superclass chain and super interfaces chains are also loaded. We defined a Lisp function collect-assignableToName that crawls along the class hierarchy and collects all reachable classes. The invariant loader-inv asserts that for all classes currently in the class table, the following loader-inv-helper1 is true. We have proved that load_class preserves the invariant[2].

```
(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)))

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

In the second step, we show the above alternative formulation implies our original formulation of the invariant.

```
(defthm inv-and-isAssignableTo-env
  (implies
    (and (loader-inv s)
         (no-fatal-error? s)
         (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 to study the adequacy of current CLDC bytecode verification algorithm as described in JSR-139 for J2ME JVMs[17].

[2]To prove 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.

We want to show that "verified" bytecode will behave as we expected if the CLDC's bytecode verification algorithm is adequate.

Our approach to the ultimate goal of verifying a bytecode verifier is taken from Cohen's original work[1]: We introduce a "defensive" version of our model that checks for "all possible" runtime anomalies. We study whether the current CLDC specification can provide the guarantee that verified programs never cause anomalies in this JVM model by trying to prove a theorem that states the property.

We already have some partial results about the correctness of the bytecode verification alogrithm on one of our simpler JVM models, M3, see [7]. For example, we have proved that executing any program on a regular JVM (M3) computes the "same" result as executing it on a "defensive" but inefficient JVM, when the "defensive" JVM does not detect any anomaly. We also proved that any executions on "defensive" JVM preserves a *consistent-state* predicate, which implies that starting from a consistent initial state, a "defensive" JVM always has well-formed heaps, for example, pointers in the system always point to valid objects. We also proved that the original iterative bytecode verification algorithm terminates on all 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 a single threaded program for computing factorial is correct. The property is stated with the following ACL2 formula. The ACL2 system mechanically checked our proof of the theorem.

```
 (defthm factorial-is-correct
  (implies
      (and (poised-for-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)))))
```

where poised-for-execute-fact specifies that the state is a well formed state, the next instruction will invoke the factorial method, and the class table is a particular one where, after method resolution, the method found is actually a particular bytecoded factorial method. The theorem itself says the effect of executing fact-clock(n) steps from a state "poised" for executing the factorial program, is equal to setting the program counter to point to 3+pc(s) and replace the top of the operand stack with the lower 32 bits twos-complements of the mathematical fact(n).

As an observation, this theorem is a specification for total functional correctness of the corresponding bytecode factorial program. The proof of this specification is mechanically checked by the ACL2 system.

On a simpler model, we have also proved theorems about methods that change the heap such as a linked-list implementation of insertion sort[13]. We proved that executing insertion sort method with a pointer to the first node of a linked list. The sorting method returns returns a pointer that points to an ordered permutation of the input list.

We have proved theorems about multi-threaded Java programs that synchronize through a monitor to gain exclusive access to the shared resources[14]. In this example, an unbounded number of threads repeatedly compete for a lock on a container to increase a counter in the container. We proved that the counter goes up monotonically until it wraps around because of an overflow.

Our group has proved theorems relating program execution on one model to execution on another, simpler model. One such example is reported in [15] 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 syntactical restriction, 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 about 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[1]. 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 to study the various aspects of the JVM formally.

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

Our work is different from most other simulators because it is implemented in a functional programming language for which there is automated reasoning support.

The simpler single threaded Java Card Virtual Machine was modeled at INRIA, Sohphia-Antipolis, France [5]. They give a formal executable semantics for the Java Card Virtual Machine in Coq[2], an ML-like language. They claim the model is quite complete and can execute any JavaCard program — given they provide the native implementations of necessary API 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 [17] 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 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 approach brings the benefits of both simulation and machine-checked analytical reasoning.

7. REFERENCES

[1] R. Cohen. Defensive Java Virtual Machine Version 0.5 alpha Release. Available from `http://www.cli.com/software/djvm/index.html`, 1997.

[2] The Coq Development Team. Technical report.

[3] D. Burger and T.M. Austin. The SimpleScalar tool set, Version 2.0. Technical Report 1342, Universit of Wisconsin-Madison Computer Science Department, June 1997.

[4] D. Hardin, D. Greve, M. Wilding, and 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.

[5] G. Barthe, G. Dufay, L. Jakubiec, B. Serpette, and S. Melo de Sousa. A formal executable semantics of the javacard platform. In D. Sands, editor, *Proceedings of ESOP'01*, 2001.

[6] D. Greve. Symbolic simulation of the JEM1 microprocessor. In Ganesh Gopalakrishnan and Phillip Windley, editors, *Formal Methods in Computer-Aided Design (FMCAD'98)*, pages 321–333, Palo Alto, CA, 1998.

[7] H. Liu and J S. Moore. A formal study of bytecode verification in ACL2. Partial results on the correctness

of the bytecode verification algorithm for the JVM model M3, unpublished, March 2002.

[8] H. Liu and J S. Moore. JVM model: M6 source code. `http://www.cs.utexas.edu/users/hbl/pub/M6/ivme03/`, March 2003.

[9] J S. Moore and G. Porter. An executable formal java virtual machine thread model. In *Proceedings of 2001 JVM Usenix Symposium*, Monterey, California, April 2001. USENIX.

[10] J S. Moore, R. Krug, H. Liu, and 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.

[11] M. Kaufmann, P. Manolios, and J S. Moore. *Computer-aided Reasoning: An approach*. Kluwer Academic Publishers, 2000.

[12] M. Wilding, D. Greve, and D. Hardin. Efficient simulation of formal processor models. *Formal Methods in System Design*, 18(3), May 2001.

[13] J S. Moore. Proving theorems about Java and the JVM with ACL2. In M. Broy, editor, *Lecture Notes of the Marktoberdorf 2002 Summer School*. Springer, 2002.

[14] J S. Moore and G. Porter. The apprentice challenge. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 24(3):193–216, 2002.

[15] G. Porter. A commuting diagram relating threaded and non-threaded JVM models. Technical report, Honors Thesis, Department of Computer Sciences, University of Texas at Austin, 2001.

[16] Connected Limited Device Configuration (CLDC) and the K Virtual Machine. `http://java.sun.com/products/cldc/`.

[17] Connected Limited Device Configuration (CLDC) Specification 1.1. `http://jcp.org/en/jsr/detail?id=139`.

[18] Java 2 Platform, Micro Edition. `http://java.sun.com/j2me/`.

[19] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. Addison-Wesley Publisher, second edition, 1999.