Formal Models of Java at the JVM Level
A Survey from the ACL2 Perspective

J Strother Moore
Robert Krug
Hanbing Liu
George Porter

Department of Computer Sciences, University of Texas at Austin
`moore@cs.utexas.edu`
`http://www.cs.utexas.edu/users/moore`

Abstract. We argue that a practical way to apply formal methods to Java is to apply formal methods to the Java Virtual Machine (JVM) instead. A Java system can be proved correct by analyzing the bytecode produced for it. We believe that this clarifies the semantic issues without introducing inappropriate complexity. We say "inappropriate" because we believe the complexity present in the JVM view of a Java class is inherent in the Java, when accurately modeled. If it is desired to model a subset of Java or to model "Java" with a slightly simpler semantics, that can be done by formalizing a suitable abstraction of the JVM. In this paper we support these contentions by surveying recent applications of the ACL2 theorem proving system to the JVM. In particular, we describe how ACL2 is used to formalize operational semantics, we describe several models of the JVM, and we describe proofs of theorems involving these models. We are using these models to explore a variety of Java issues from a formal perspective, including Java's bounded arithmetic, object manipulation via the heap, class inheritance, method resolution, single- and multi-threaded programming, synchronization via monitors in the heap, and properties of the bytecode verifier.

1 ACL2 Background

ACL2 [11] is a functional programming language based on Common Lisp, a first-order mathematical logic with induction and recursive definition, and a mechanical theorem prover in the style of the Boyer-Moore theorem prover NQTHM [2, 4]. Among other successful industrial uses of ACL2 is the verification of the hardware designs for the elementary floating-point arithmetic operations on the AMD Athlon microprocessor [21] and the formalization of the first silicon version of the JVM [8, 9]. See [10] for other case studies.

In this paper we advocate the use of formal models of the JVM [13] to verify Java programs. Some readers may think this is an impractical suggestion. But work by Yu [5] with NQTHM (the predecessor of ACL2) supports our suggestion. Yu developed an operational formal model of the Motorola 68020 and then verified C programs from the Berkeley C String Library by verifying the machine code produced by `gcc`. Since the conceptual gap between C and 68020 machine code is much greater than the gap between Java and JVM bytecode, we believe it is reasonable to follow an analogous strategy to deal with Java programs.

2 Our Basic Approach

Our models of the JVM are operational ones. The state of the machine is represented by a list containing, say, a thread table, a heap, and a class table. The thread table is a list containing an entry for each thread. The entry includes the thread's call stack, scheduled status and other information. A call stack is a stack (list) of frames, each of which contains a program counter, the method body, a map from local variable names to values, an operand stack, and a flag indicating whether the method is synchronized. The heap is a finite mapping from reference "addresses" to instance objects. The class table is a list describing the superclasses, fields and methods and other attributes of each class. We then define in ACL2 the function that "steps" such a state, producing the next state. We finally define a function, `run`, that "runs" a state, by stepping it repeatedly. Such an ACL2 model of the JVM may be thought of as a system of Lisp programs that simulates the JVM.

We have produced several such models of the JVM, so that we can explore ways to prove various kinds of properties. Before discussing the variety of formal models we have, we will use one of them to illustrate the foregoing sketch. The model we use is a multi-threaded JVM with unbounded arithmetic. It support classes, instances, instance methods, monitors and synchronization, but not arrays, floats and certain other data types. It also completely ignores class loading, constructor methods, exceptions, the JVM's provisions for type safety, and a variety of other issues.

For each JVM opcode supported in the model we define an ACL2 function that produces the corresponding state change. Here, for example, is that part of the formal model for an instruction called `LOAD`, which is analogous to the JVM's family of typed load instructions `ILOAD`, `ALOAD`, `DLOAD`, etc. In this function, `inst` is a symbolic form of the particular load instruction to be executed; its value will be a list of the form `(LOAD` *var*`)`, where *var* is a variable name. The variable `th` identifies which thread is to be stepped and `s` is the JVM state.

```
(defun execute-LOAD (inst th s)
  (make-state
   (modify-tt th
    (push (make-frame (+ 1 (pc (top-frame s th)))
                      (locals (top-frame s th))
                      (push (binding (arg1 inst)
                                     (locals (top-frame s th)))
                            (stack (top-frame s th)))
                      (program (top-frame s th))
                      (sync-flg (top-frame s th)))
```

```
          (pop (call-stack s th)))
     'SCHEDULED
     (thread-table s))
    (heap s)
    (class-table s)))
```

Informally, this function returns a new state obtained by changing the thread table of `s` at thread `th`. The topmost item on the call stack of that entry is popped off and replaced by a new frame in which the program counter has been advanced and the value of *var* has been pushed onto the operand stack of that frame.

Here is our bytecode for the instance method

```
public int fact(int n){
 if (n<=0) return 1;
   else return n*fact(n-1);}
```

except in our model arithmetic is not bounded.

```
("fact" (N) NIL                      ;  Method int fact(int)
 (LOAD N)                            ;   0 iload_1
 (IFGT 3)                            ;   1 ifgt 6
 (PUSH 1)                            ;   4 iconst_1
 (XRETURN)                           ;   5 ireturn
 (LOAD N)                            ;   6 iload_1
 (LOAD THIS)                         ;   7 aload_0
 (LOAD N)                            ;   8 iload_1
 (PUSH 1)                            ;   9 iconst_1
 (SUB)                               ;  10 isub
 (INVOKEVIRTUAL "Alpha" "fact" 1)    ;  11 invokevirtual ...fact...
 (MUL)                               ;  14 imul
 (XRETURN))                          ;  15 ireturn
```

Because our model is an ACL2 program, it can be executed on concrete data to produce concrete results. Because ACL2 is a mathematical logic, it is possible to prove the following theorem:

```
(implies (poised-to-invoke-fact th s n)
         (equal (top
                  (stack
                   (top-frame
                    th
                    (run (fact-sched n th) s))))
                (factorial n)))
```

which says that, given any state poised, in thread `th`, to execute the `fact` bytecode the natural number `n`, the execution of a certain number of instructions in thread `th` will leave `n`! on top of the operand stack in thread `th`. The number of instructions required is given by the function `fact-sched`, which returns a schedule adequate to compute the method on input `n`. We have proved similar theorems about other arithmetic methods, methods manipulating the heap in destructive ways [15], and insertion sort implemented in a list processing class [12]. Insertion sort is discussed briefly below.

3 A Survey of Our Models

We have several JVM models and are in the process of building others. All of our current models ignore floats, class loading and initialization, exceptions, and interfaces. We do not consider floats a problem; there is so much work in modeling floating-point arithmetic in ACL2 (see for example [21]) that we have extensive floating-point models and libraries about them. Aspects of class loading and initialization, exceptions and interfaces have been modeled by others [19, 1]. Garbage collection is invisible on the JVM and so need not be modeled.

3.1 Single-Threaded/Non-Safe/Unbounded

Our basic model is a single-threaded JVM in which we ignore typing issues and support unbounded integer arithmetic only. Using this model we have proved a variety of theorems about bytecode programs, including a single-threaded version of the factorial theorem above and theorems involving the overriding of methods and the destructive modification of instance objects in the heap [15]. Using this model we can explore basic issues of code specification and verification, including control flow and data operations, instance object creation and manipulation, class inheritance, and method resolution and invocation.

For example, we have used the model to prove the correctness of a bytecoded insertion sort method that copies a linked list of numbers in the heap, producing a permutation of it in which the elements appear in ascending order. To state the theorem we had to define the sense in which a reference (into a given "non-circular" heap) denotes some structure. The theorem we proved says that if the `isort` method (not shown here) is invoked on a reference, ref_0 and allowed to run for a certain number of instructions, returning a reference ref_1, then the list denoted by ref_1 in the final heap is an ordered permutation of the list denoted by ref_0 in the original heap. The preconditions imposed certain constraints on the non-circularity of the initial reference [12]. Here is the theorem proved.

```
(implies (poised-to-invoke-isort s0)
         (let* ((x0 (top (stack (top-frame s0))))
                (heap0 (heap s0))
                (n0 (isort-clock x0 heap0))
                (s1 (run n0 s0))
                (x1 (top (stack (top-frame s1))))
                (heap1 (heap s1)))
           (let ((list0 (deref* x0 heap0))
                 (list1 (deref* x1 heap1)))
            (and (ordered list1)
                    (perm list1 list0)))))
```

One can prove theorems about non-terminating computations in ACL2. If one adds to the model an instruction for explicitly indicating the normal termination of a program (e.g., add a halt flag to the state and arrange for a bytecode instruction, e.g., `halt`, to set it and for the machine not to proceed afterwards), one can prove theorems about the conditions under which a program halts normally, including that halting never occurs. One can also eliminate the use of "clocks" [14].

3.2 Single-Threaded/Non-Safe/Bounded

We have produced a version of the simple machine that supports Java's `int` and `long` (bounded) arithmetic. It also supports arrays. Using this model we have a proved the analogous theorem about the bounded factorial method. The code for this method is like that shown for `fact` above, except that the arithmetic operations are those for 32-bit twos complement. The theorem states that the final answer is equal to the result of converting to an `int` the factorial of the input. This theorem correctly characterizes the actual behavior of the Java program for `fact` shown above.

The user input required to prove the bounded factorial is exactly analogous to that required to prove the unbound factorial, justifying our belief that the unbounded model is a simpler (though technically inaccurate) test bed. The "new" reasoning, about modular arithmetic, is handled automatically by an ACL2 library of lemmas. We are continuing the development of ACL2's already extensive collection of arithmetic theorems.

3.3 Multi-Threaded/Non-Safe/Unbounded

An orthogonal variation of the basic model introduces multiple threads [17].

Each entry in the thread table lists a unique thread number, a call stack, a status flag (e.g., indicating whether the thread has been started), and a reference to the instance object representing the thread object in the heap. We do not model the scheduler, which is unspecified in [13], but provide an "oracle" to the operational semantics.

With this model we have proved an interesting theorem about the Java classes shown in Figure 1. Inspection of the code shows that the `main` method in class `Apprentice` starts an unbounded number of `Job`s, each of which is contending for a shared object called the `Container`. Each `Job` is in an infinite loop incrementing the `counter` field of the `Container`. Each such increment is done within a synchronized block. (The model supports unbounded arithmetic.)

One might think that it is obvious that the value of the `counter` field of the `Container` increases monotonically. However, this is a nontrivial observation that requires showing that each `Job` has mutually exclusive access to the counter. Again, the naive Java user may think this mutual exclusion property is obvious.

```
class Container {
    public int counter; }
class Job extends Thread {
    Container objref;
    Object x;
    public Job incr () {
        synchronized(objref) {
            objref.counter = objref.counter + 1; }
        return this; }
    public void setref(Container o) {
        objref = o; }
    public void run() {
        for (;;) {
            incr(); } } }
class Apprentice {
    public static void main(String[] args){
        Container container = new Container();
        for (;;) {
            Job job = new Job();
            job.setref(container);
            job.start(); } } }
```

Fig. 1. The Apprentice Class: Unbounded Parallelism

We have had several programmers dismiss our theorem as trivial and claim that it may be observed merely by looking at the text

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

in the code for class `Job`. This claim is false.

A few changes to the `main` method of the `Apprentice` class can cause mutual exclusion to be violated and can permit the `counter` value to decrease under some scheduling regimes. These changes do not involve writing to the `counter` field of the `Container` or changing the `Job` class at all. The pathological behavior (of the counter decreasing) is ultimately manifested by the very assignment statement shown above. The changes we have in mind can cause that "synchronized" assignment statement to clobber the counter without owning the monitor for it.

Since many readers insist that it is "obvious" that the `Apprentice` class causes the counter to increase monotonically, we will not explain here how to cause the bad behavior. Ask someone who thinks it is obvious. Or try to prove it from a detailed formal model of multi-threaded Java. Our discussion of the problem and our proof is reported in [16].

Our multi-threaded model includes all of the functionality of our basic machine (e.g., classes, heap-allocated instance objects, virtual method invocation, etc.) plus support for the `Thread` class (including the significance of the `run` method for an extension of the `Thread` class, the native methods `start` and `stop`, monitors on all `Objects`, the opcodes `MONITORENTER` and `MONITOREXIT`, and support for synchronous methods.

4 Relations Between Models

So far we have only discussed theorems about particular bytecoded methods under the semantics formalized in particular models. Because our models are formal, we can reason about the models themselves and even relate them. Lack of space precludes much discussion.

4.1 Single- versus Multi-Threaded Models

We have proved [18] a theorem relating the single-threaded model to the multi-threaded one. If the multi-threaded machine is being used to do what is essentially a single-threaded computation, the single-threaded machine may be used instead. We formalize the hypothesis so that we are concerned with states in which only one thread is scheduled (meaning the `start` method has been called on only one thread) and the bytecode running in that thread does not create or interfere with other threads. The conclusion is a "commuting diagram" stating that the "same" computation could be done on the single-threaded model by transforming the states appropriately. The theorem allows us to "lift" certain verified programs from the single-threaded model to the multi-threaded model.

Ultimately we hope to be able to reason formally about "independent" concurrent threads by reasoning about each on the single-threaded model. The biggest problem will be combining the "independent" effects of the two threads on the shared heap. This involves reasoning not unlike that already done in analyzing the denotation of the object references in the heap produced by the insertion sort method.

4.2 Single-Threaded/Type Safe/Unbounded

We have developed a "type safe" version of the basic machine. Before each instruction is executed, this machine checks that the state is suitable for the execution of the instruction. For example, if an `ADD` instruction is to be executed, then the machine dynamically checks that the operand stack has at least two items on it and that the top two items are numbers. The machine sets a flag in the state and halts if the next instruction is to be executed in an unacceptable situation.

We are developing a formal version of the Java bytecode verifier described by [13] that crawls over a class declaration and does a certain syntactic check of the code therein. Our goal is to prove a theorem relating the type safe machine to the unsafe machine, namely, the two are "equivalent" on code that has been accepted by the bytecode verifier. This work can be thought of as leading towards the formal statement of the correctness of the bytecode verifier and the mechanized verification that for a particular verification algorithm.

5 Related Work

The earliest formal mechanized JVM model we know of was Cohen' "defensive JVM" [6], formalized in ACL2. Our series of models evolved from his: Moore and Cohen simplified Cohen's model and developed the series of successive elaborations to make it easier to teach at the undergraduate level.

Projects formalizing the JVM are ongoing in other mechanized logics with considerable success. The soundness of a bytecode verification algorithm is addressed in Isabelle/HOL in [20, 19]. The approach follows closely the class file format of [13] and model aspects of interfaces, signatures and exceptions, all of which we ignore. As in [6] and (some of) our work, type information is stored with data and instructions are modeled as state transforming functions. The Isabelle/HOL work is the first published mechanically checked proof of the soundness of a bytecode verifier.

Somewhat closer to our work is that done with Coq and described in [1]. In this work, an operational model of the entire JavaCard VM is presented. They provide a tool for converting class files into their formal format. They also verify a bytecode verifier mechanically. The authors of [1] stress the importance of executability – an emphasis with which we agree. They do not discuss the efficiency with which their model can be implemented.

ACL2 was used to model the Rockwell JEM1 microprocessor, the world's first silicon JVM, now marketed by Ajile Systems, Inc. The formal ACL2 model was actually used in the standard test bench on which Rockwell engineers tested the chip design against the requirements by executing compiled Java programs. The ACL2 model executed at approximately 90% of the speed of the previously used C model [8, 9]. In [7], Wilding and Greve describe how microprocessor models in ACL2 are made to execute fast. The model there executes at approximately 3 million simulated instructions per second on a 733 MHZ Pentium III host running Allegro Common Lisp.

As far as we know, ours is the first formal thread model for the JVM. In addition, the emphasis of our work is on the verification of bytecode programs with respect to the operational semantics. This is surely within the reach of the related work above, but has not, apparently, been a focus of their work. Because of the way previously proved lemmas in the ACL2 library can be used to configure ACL2 to do proofs automatically in a given domain, we anticipate that the continued development of correctness proofs for individual bytecoded methods will increase the ease with which new methods can be verified.

6 Conclusion

We have described a variety of formal models of the JVM and discussed Java and JVM programs that we have verified with respect to these models. We have also discussed formally verified relationships between some of our models.

These examples support the contention that with formal operational semantics of the JVM one can

– specify and verify Java code with respect to a detailed and accurate semantics,
– reuse much previously developed formal work,
– explore the specifications of code under various refinements of the semantics of Java,
– establish properties of the semantic models,
– formally relate different semantic models, and
– specify and verify the bytecode verifier.

Our models are inadequate for practical Java: among other omissions are floating point, exceptions, and class loading. But there is ample evidence [10] that ACL2 is rugged enough to permit the models to be sufficiently elaborated.

Among the compelling reasons to base a formal semantics of Java on an operational semantics of the JVM are the following. First, the Java compiler takes care of many subtle static semantics issues. Second, the operational semantics of the JVM can be executed, meaning it is possible to test the semantics against accepted implementations of the JVM. Third, the operational semantics is easily unwound by standard symbolic evaluation and induction techniques [3]. Fourth, and most important, the semantics is rendered *formally*, so it can be inspected by language experts and used directly by the verifier.

7 Acknowledgments

Our JVM models owe much to Rich Cohen who used ACL2 to formalize a single-threaded version of the "defensive JVM" [6]. We are grateful to Rich for his pioneering effort into the JVM formalization, as well as to the entire ACL2 and NQTHM communities for their development of techniques to formalize and reason about such machines. We are also grateful to David Hardin and Pete Manolios, who have each made many valuable suggestions in the course of this work.

References

[1] 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, *ESOP 2001*, volume LNCS 2028, pages 302–319. Springer-Verlag, 2001.
[2] R. S. Boyer and J S. Moore. *A Computational Logic*. Academic Press, New York, 1979.
[3] R. S. Boyer and J S. Moore. Mechanized formal reasoning about programs and computing machines. In R. Veroff, editor, *Automated Reasoning and Its Applications: Essays in Honor of Larry Wos*, pages 147–176. MIT Press, 1996.
[4] R. S. Boyer and J S. Moore. *A Computational Logic Handbook, Second Edition*. Academic Press, New York, 1997.
[5] Robert S. Boyer and Yuan Yu. Automated proofs of object code for a widely used microprocessor. *Journal of the ACM*, 43(1):166–192, January 1996.

[6] R. M. Cohen. The defensive Java Virtual Machine specification, version 0.53. Technical report, Electronic Data Systems Corp, Austin Technical Services Center, 98 San Jacinto Blvd, Suite 500, Austin, TX 78701, 1997.

[7] D. Greve, M. Wilding, and D. Hardin. High-speed, analyzable simulators. In Kaufmann et al. [10], pages 113–136.

[8] D. A. Greve and M. M. Wilding. Stack-based Java a back-to-future step. *Electronic Engineering Times*, page 92, Jan. 12, 1998.

[9] David A. Greve. Symbolic simulation of the JEM1 microprocessor. In *Formal Methods in Computer-Aided Design – FMCAD*, Lecture Notes in Computer Science. Springer-Verlag, 1998.

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

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

[12] M. Kaufmann and J S. Moore. A flying demo of ACL2. Technical Report `http://www.cs.utexas.edu/users/moore/publications/flying-% -demo/script.html`, Computer Sciences, University of Texas at Austin, 2000.

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

[14] P. Manolios and J S. Moore. Partial functions in acl2. Technical Report `http://-www.cs.utexas.edu/users/moore/publications/defpun/% -index.html`, Computer Sciences, University of Texas at Austin, 2001.

[15] J S. Moore. Proving theorems about Java-like byte code. In E.-R. Olderog and B. Steffen, editors, *Correct System Design – Recent Insights and Advances*, pages 139–162. LNCS 1710, 1999.

[16] J S. Moore and G. Porter. Mechanized reasoning about Java threads via a JVM thread model. *(submitted for publication)*, 2000. `http://www.cs.utexas.edu/users/moore/publications/m4/proofs.ps.gz`.

[17] J S. Moore and G. Porter. An executable formal JVM thread model. In *Java Virtual Machine Research and Technology Symposium (JVM '01)*, 2001 (to appear). `http://www.cs.utexas.edu/users/moore/publications/m4/model.ps.gz`.

[18] 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.

[19] Cornelia Pusch. Formalizing the Java virtual machine in Isabelle/HOL. Technical Report TUM-I9816, Institut für Informatik, Technische Universiät München, 1998. See URL `http://www.in.tum.de/~pusch/`.

[20] Cornelia Pusch. Proving the soundness of a Java bytecode verifier in Isabelle/HOL. Technical report, Institut für Informatik, Technische Universiät München, 1998. See URL `http://www.in.tum.de/~pusch/`.

[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*, 1:148–200, December 1998.