Java Program Verification via a JVM Deep Embedding in ACL2

Hanbing Liu and J Strother Moore

Department of Computer Science
University of Texas at Austin
Austin, Texas, 78712-1188, USA

Abstract. In this paper, we show that one can "deep-embed" the Java bytecode language, a fairly complicated language with a rich semantics, into the first order logic of ACL2 by modeling a realistic JVM. We show that with proper support from a semi-automatic theorem prover in that logic, one can reason about the correctness of Java programs. This reasoning can be done in a direct and intuitive way without incurring the extra burden that has often been associated with hand proofs, or proofs that make use of less automated proof assistance. We present proofs for two simple Java programs as a showcase.

1 Introduction

In order to reason about software/hardware artifacts mathematically, we need to represent the artifacts as mathematical objects. We often formalize them by assigning a precise semantics to the underlying language constructs or hardware primitives.

In cases where there exists an axiomatic semantics for the language, we can reason about the artifact directly using axioms and specialized derivation rules. A typical example is Hoare logic [7].

However, such an approach makes it hard to use existing general purpose theorem provers such as ACL2 [13] and PVS [3], because for each different logical system, a new computer aided reasoning engine must be constructed. Constructing a specialized theorem prover comparable to current mature general purpose ones is often time consuming and error prone. Generic theorem proving environments such as Isabelle [18] prove to be useful in this setting, because Isabelle can be configured to function as a specialized theorem prover for different formalisms. Alternatively, if we can embed the language into the formalism of a powerful general purpose theorem prover, we can use that theorem prover for program verification projects. We think that this approach is also practical.

There are two common choices for formalizing a program artifact in the logic of a theorem prover. In a "shallow embedding" one describes a process by which a conjecture about a given program may be converted to an "equivalent" formula. Neither the programs (the original forms in the old syntax) nor the process are defined within the logic – they are meta-level entities. In a "deep embedding",

programs and their environments are logical objects that are related by functions and relations formally defined within the logic. Usually, the syntax of the original programs is preserved. The semantics of the basic language constructs are formalized instead of the semantics of specific programs.

Each approach has its pros and cons. Shallow embedding requires less logical infrastructure and often produces simpler conjectures to prove. Deep embedding, however, allows one to reason formally not just about a given program but about relations between programs and properties of the semantics itself. For example, deep embedding a program with a semantics for the underlying programming language allow the user to reason about properties shared by a set of programs. Deep embedding also allows the user to derive new proof rules as theorems. However, much logical manipulation must occur to wade through the details of the semantics. Automation is highly desirable and brings new capabilities, such as simulation, symbolic evaluation, and other analysis tools.

It is this paper's thesis that Java program verification via a deep embedding of the JVM into the logic of ACL2 is a viable approach. In fact, we believe that with the proper support from a powerful semi-automatic theorem prover, the deep embedding approach is better than the shallow embedding approach in the sense that it brings more assurance of the verification result without incurring much extra burden.

In section 2, we present our deep embedding of a full featured JVM into ACL2. The executability of ACL2 models allows one to use such a complete deep embedding as a JVM simulator. In the section 3, we present correctness proofs for two simple Java programs to demonstrate the approach and illustrate some useful techniques in handling a deep embedding. In section 4, we review other work and comment on the proof effort required by our method and explain briefly the limitations of our work. We summarize and conclude in section 5.

2 Deep Embedding a JVM in ACL2

We wrote a precise model of the JVM in ACL2 to formally capture the meaning of Java bytecode programs. The JVM model is based on the JVM specification. We follow the KVM, a C implementation of the JVM, as a reference model in our "implementation"[1].

ACL2 is an applicative (side-effect free) subset of Common Lisp. Our JVM model can be executed as a Lisp program. It is implemented with around ten thousand lines of Lisp (ACL2) in about 25 modules. It implements most features of a JVM such as dynamic class loading, thread synchronization via monitors, together with 21 out of the 41 native methods defined in Java 2 Microeditions's CLDC library [22]. The features that are missing are the "reflection" capability in the full JVM, user defined class loaders, floating point arithmetic, and native methods related to some I/O operations.

[1] In the process, we discovered several implementation errors in the KVM. Some were already known to Sun. Some are forwarded to the KVM development team.

Realistic Java programs can execute on the model. We expect to run a suitable subset of some conformance test suite at some point. The details of the model are described in the paper [12], which we presented in the workshop of *Interpreter, Virtual Machine and Emulator 2003*, affiliated with PLDI [2].

2.1 Motivation to embed a JVM

We are interested in applying theorem proving techniques to software verifications projects. In particular, we are interested in reasoning about the properties of the Java virtual machine and Java software executing on the JVM.

This is one of the reasons that we decided to deep-embed the Java bytecode language via a JVM model. The other reason is that we feel more confident in our ability to formalize the semantics of the bytecode language of the Java Virtual Machine than our ability to correctly assign meanings to specific Java programs or the Java programming language.

Like most imperative programming languages, the semantics of Java are hard to formalize directly. The object oriented features such as method overriding, dynamic method resolution, access permissions, and constructs such as inner classes present significant challenges.

As expressed in our position paper [11], the JVM bytecode is simpler and more precisely defined than Java. We therefore define the semantics of the bytecode language with an operational JVM interpreter. We reason about Java programs by reasoning about the corresponding bytecode program via `javac` on the JVM model. This approach was demonstrated by Yu [1] using the predecessor of ACL2, Nqthm to reason about C via `gcc` and a model of the Motorola 68020.

Because the model is formally defined in the logic, we can also reason about it independent of the consideration of any particular program. This allows us to derive new proof rules from the semantics, as well as to explore the implications of semantics, i.e. properties of the JVM itself. Both activities increase our understanding of and confidence in the semantics; and both activities are supported by machine-checked reasoning rather than informal reasoning. Finally, the bytecode analyzed is more closely related to what is actually executed than the original Java. In summary, we regard deep embedding as offering higher assurance than shallow embedding.

2.2 The JVM Model in ACL2

The completeness of our JVM embedding determines the range of Java programs that we can reason about as well as the relevance of our formal statements about the Java programs. Our model is fairly complete — it is a realistic JVM simulator that executes most Java programs that do not use I/O nor floating point operations.

[2] A revised version was accepted for publication in a special issue of the journal "Science of Computer Programming" for IVME'03

Since ACL2 is applicative, we have to model the JVM state explicitly. All aspects of the machine state are encoded explicitly in one logical object denoted by a term. A JVM state in this model 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 the next time this thread is scheduled. 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.

As a concrete example of how a piece of state is represented, the following entry is taken from an actual thread table when we used 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 operand stack
         (LOCALS 104)
         (METHOD-PTR "FactHelper" "<init>" ...)
         (SYNC-OBJ-REF .  -1))
  (FRAME (RETURN_PC .  18)
         ...
         (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
```

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 representing the thread in the heap.

The semantics of the JVM instructions are modeled operationally as state transition functions. 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 divided by the first item on it. In ACL2, the function `truncate` returns an integer quotient rounded toward 0.

The top level interpreter loop is modeled as following:

```
(defun run (sched s)
 (if (endp sched) s        ; end of schedule
   (let ((nid (car sched)) ; else
         (cid (current-thread s)))
     (if (equal cid nid)
         (run (cdr sched) (step s)) ; execute one step
       (run (cdr sched)
         (loadExecutionEnvironment
           nid             ; proper thread context switch
          (storeExecutionEnvironment s)))))))))
```

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.

The scheduling policy is thus left unspecified. Any schedule can be simulated. However to use the model as an execution engine without providing a schedule list 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 bytecode instruction.

Before concluding this section, we observe that the `defun` of `run` (and of each of the other functions shown above) can be thought of in either of two ways. First, it defines a side-effect free Lisp program which can be executed on concrete data. Second, it introduces a new logical definitional equation which can be used to prove theorems about the newly defined function symbol. Preserving the view that we are "merely" defining an executable model often provides valuable clarity. Executing the model often provides assistance in the search for true statements about programs and in the search for proofs. In some sense, the "embedding" is so direct that it is transparent, i.e. we are reasoning about the JVM directly.

3 Java Program Verification

With our choice of a deep embedding of the Java bytecode language, reasoning about any Java bytecode program implies that we need to deal with the complexity of the JVM in addition to the program itself. The task seems to be formidable. This additional complexity is considered one of the major drawbacks of the deep embedding approach.

We acknowledge that deep embedding adds extra complexity in the verification of programs. But if one can accomplish the program verification task at this level, we believe that additional confidence is gained.

The central remaining question is whether one can reduce the "extra" complexity to an acceptable level. It is our experience with the JVM and ACL2 that one can achieve this reduction by configuring the rewriting engine of ACL2 using lemma libraries. Such configuration needs to be done only once for a class of programs.

In this section, we present proofs of two simple programs to show how we manage the complexity in ACL2. We show the proof for the first program in some detail and refer readers to the actual proof scripts for comments and other details in the supporting material [6].

3.1 ADD1 Program

The first program is trivial.

```
public class First {
    public static void main(String[] args) {
        int i=1;
        int j=i+1;
        i=j;
        return;}}
```

The `main` method is straight line code that only modifies the operand stack and local variables in the current call frame, i.e. the top most activation record from the call stack of the current thread. With this example, we illustrate how we can reason about programs and segments of programs which only manipulate the current call frame.

Our tool *jvm2acl2* transforms the `First.class` into the following format, which directly corresponds to the class file format [23].

```
'(class "First"               ; class name is First
        "java.lang.Object"    ; Superclass is java.lang.Object
         ....
         (fields)             ; list of field definitions
         (methods             ; list of method definitions
               (method "<init>"
                   ....)
               (method "main"    ; method name.
                   (parameters (array (class "java.lang.String")))
                   (returntype void)
                   (accessflags  *class*  *public*  *static* )
                   (code
                        (max_stack 2) ...
                        (parsedcode
                           (0 (iconst_1)) ;; *Note: (0 (iload_2))
                           (1 (istore_1))
                           (2 (iload_1))
                           (3 (iconst_1))
                           (4 (iadd))
                           (5 (istore_2))
                           (6 (iload_2))
                           (7 (istore_1))
                           (8 (return))
                           (endofcode 9))
                        (Exceptions )
                        (StackMap )))))
          ....)
```

This logical constant represents the `First.class` file. A list of such class constants together with the JVM interpreter gives the semantics of the original Java program. For this program, the semantics of the `main` method only depends on the JVM interpreter and this particular class itself; for more complicated programs, the meaning of a user-defined class often depends on other classes.

To make the example slightly more interesting, we change by hand the first instruction, `(0 (iconst_1))`, to `(0 (iload_2))`[3]. We prove that by starting in a state where the `pc` is 0 and executing 7 steps according to a round robin scheduling algorithm, we produce a state in which the value in the second slot of the locals is increased by one from its original value. We describe what is essential to configure ACL2 in deriving this.

The first step is to identify the appropriate abstractions of JVM executions and formalize those concepts properly. For example, consider the intuitive understanding of what the "next instruction" is. In our JVM model, such a concept is complex because the state is complex. The next instruction of a given state is the instruction that resides at a certain offset within the bytecode of the current method, where the offset is given as the value of the pc field of the state; and the current method is identified by consulting the current class table using the method identifier in the activation record of the current thread. One must also consider special conditions, such as when the current thread does not exist or has been stopped by another thread. Such complexity is reduced by defining a named function of state, `next-inst`, and using it consistently within the model so that the above details are not exposed. We regard this as just good modeling practice. We typically configure ACL2 so as not to expand the definitions of these abstractions ("disabling" the associated rules in ACL2's database). We will rely only on a set of properties of these operations on "states of interest", which we prove before we disable the definition.

The reason that the intuitive informal notion of "next instruction" appears simpler is probably because the user evaluates it only on symbolic states for which `next-inst` returns constants. That is, when considering the verification of a particular program in thread 0 informally, we do not contemplate whether there can be a context switch to a thread, or whether the current activation record corresponds to the program of interest.

In the second step, we formalize the concepts that capture the identified domain, i.e. "states of interest". We prove that in the identified domain, complicated primitive operations have the simple behavior as expected. To formalize this we introduce an equivalence relation on states, `equiv-state`, that means, roughly, "the states are executing the same program." We are more precise below. ACL2's rewrite engine can use arbitrary equivalences and congruence lemmas (which establish that certain functions cannot distinguish "equivalent" input) to descend through the subterms of a term and replace occurrences of target terms by equivalent terms.

To cause the next instruction concept to expand only on the states of interest we prove the following lemma and then disable the definition of `next-inst`

```
(defthm equiv-state-init-state-next-inst
  (implies (equiv-state s (init-state))
           (equal (next-inst s)
```

[3] In fact, this makes the class file fail to pass bytecode verification. Here we are trying to make the proof a little bit more interesting by proving an assertion in form of $\forall i, P(i)$.

```
                (inst-by-offset (pc s) (theMethod)))))
```

The theorem asserts that for any state running the program of interest (that in the constant `(init-state)`), the next instruction can be computed by looking at a certain offset of the program of interest. This is a trivial theorem to prove. On states equivalent to `(init-state)` the body of `next-inst` can be reduced to a constant, namely, the next instruction. Thus, by proving this lemma and disabling `next-inst`, ACL2 will reduce `(next-inst s)` to a constant instruction if `s` is running the program of interest, but will not change the `next-inst` term otherwise.

In order to use the just established theorem to rewrite `(next-inst s)` into a simpler form, where `s` is a `(round-robin-run s n)` term, we need to reason about the `run` function th a round robin scheduler. In particular, we need to prove that there is no context switch as the program steps from one instruction to the next. To prove there is no context switch, we proved three types of theorems around the equivalence relation that we identified:

- A congruence on the equivalence relation `equiv-state`, which asserts that the round robin scheduler always picks the same thread if two states are `equiv-state`.

```
(defthm round-robin-schedule-equal-in-equiv-state
   (implies (equiv-state s s-equiv)
            (equal (round-robin-schedule s-equiv)
                   (round-robin-schedule s)))
  :rule-classes :congruence)
```

- A theorem that states the properties of the initial state. In this case, the round robin scheduler picks the thread 0 to execute in the initial state.

```
(defthm round-robin-schedule-init-state
  (equal (round-robin-schedule (init-state)) 0))
```

- Theorems that state equivalence is preserved by executing each primitive, e.g., `pushStack`.

```
  (defthm pushStack-preserves-equiv-state
     (equiv-state (pushStack v s) s))
```

Having so configured ACL2 by proving these lemmas, JVM execution of straight line code can be expanded into a composition of primitives by ACL2 automatically. For example,

```
(defthm round-robin-run-expansion-example
  (implies (and (equiv-state  s1 (init-state))
                (equal (pc s1) 2))
           (equiv-state (round-robin-run s1 4)
                        (init-state))))
```

is proved automatically. The theorem prover expands the `(round-robin-run s1 4)` symbolically step by step using the rewrite rules derived from the proven theorems.

In this example, starting from `pc` equals 2, `(round-robin-run s 4)` executes `(iload_1)), (iconst_1), (iadd), and (istore_2)` in sequence. Because every instruction is one byte. Executing 4 instructions shall result in a term of the following form, where pc is 6.

```
(state-set-pc 6                          ;#
   (popStack                             ;#
     (state-set-local 2 (topStack ..)    ;# cf. ISTORE_2
        (state-set-pc 5                  ;*
          (pushStack                     ;*
              (int-fix (binary-+ ...)) ;*
                  (popStack (popStack  ;* cf. IADD
                        (state-set-pc 4 ....)))))))))
                                         ;% cf. ICONST_1
                                         ;$     ILOAD_1
```

Compare this expected form to one of the intermediate goals generated by ACL2:

```
Subgoal 1'5'
(IMPLIES
 (AND (EQUIV-STATE S1 (INIT-STATE))
      (EQUAL (PC S1) 2)  ; pc = 2 in starting state
      (EQUAL 0 (CURRENT-THREAD S1)))
 (EQUIV-STATE
  (POPSTACK                              ;*
   (POPSTACK                             ;* cf. partial IADD
    (STATE-SET-PC                        ;%
      4                                  ;%
      (PUSHSTACK 1                       ;% cf. ICONST_1
       (STATE-SET-PC 3                   ;$
         (PUSHSTACK (LOCAL-AT 1 (LOCALS (CURRENT-FRAME S1)))
                    S1))))))             ;$ cf. ILOAD_1
  (INIT-STATE))).
```

In `Subgoal 1'5'`, ACL2 has reduced

```
(equiv-state (state-set-pc 6 (popStack (state-set-local 2 ...)))
             (init-state))
```

into

```
(equiv-state (popStack (popStack (state-set-pc 4 ...)))
             (init-state))
```

after "peeling off" some outer primitives such as `(state-set-pc 6 ...)`.

This shows that reasoning about `(next-inst s)` is entirely automatic. The theorem prover "knows" enough to determine the next instruction and then to

execute it as a symbolic execution engine. A formula involving the `round-robin-run` is thus reduced to a composition of primitives, such a `pushStack`, `popStack`, `state-set-pc`. The structure of the composition of primitives can be traced back to the instructions in the original bytecode sequence.

The third configuration step is to arrange for the theorem prover to reason about compositions of different primitives. This is closely related to the second step – identifying conditions under which the primitives behave according to our intuitions.

For example, we have an understanding of the effects of the push and pop operations on a stack. The following should obviously be true.

```
(popStack (pushStack v s)) = s
```

However the above is not so obvious in a Java program without some implicit hypothesis. `PushStack` pushes a value onto the operand stack of the topmost call frame of the current thread. For the above to be true, we need to explicitly show (or configure the theorem prover to automatically recognize), no other part of the state is changed. The similar problem manifests itself in other places such as showing that setting the program counter does not affect the operand stack. This is the pattern called the "frame" problem in AI research. To describe the effect of an operation, we not only need to be explicit about what is changed, but also be explicit about what does not change.

Our current solution is built around equivalences and associated congruence rules. We identify what is not changing and introduce an equivalence that groups the states that share the unchanged part. We prove that primitive operations preserve those equivalences. We prove other properties of those equivalences in the form of congruence rules.

In this `ADD1` program proof, we recognize that the program is straight line code that only modifies the operand stack and the locals. We defined the state equivalence to capture the following: if the only difference between two states are the operand stack and locals of their respective current frame, they are equivalent.

This strategy has worked well. However we can foresee limitations in our approach. When dealing with more complicated operations such as the ones that manipulate the heap, we may face the need to define a hierarchy of equivalence relations to characterize differences between different operations.

The following is the final theorem we proved about the `ADD1` program[4]. The current proof script in ACL2 is about 2000 lines with over 140 user typed lemmas[5].

```
(defthm first-is-correct
  (let ((old (local-at 2 s1)))
  (implies (and (equiv-state s1 (init-state))
                (current-thread-exists? s1)
```

[4] ACL2 has implicit universal quantifiers over all free variables appearing in a formula.

[5] This proof script represents a first cut at the problem. It can be improved. The proof is available as part of the supporting material [6].

```
        (wff-state-regular s1)
        (wff-thread-table-regular (thread-table s1))
        (wff-call-frame-regular (current-frame s1))
        (unique-id-thread-table (thread-table s1))
        (equal (pc s1) 0)
        (integerp old))
   (equal (local-at 2 (round-robin-run s1 7))
          (int-fix (+ 1 old))))))
```

The apparent complexity in the statement is partly inherent in the JVM specification. Others result from our particular choice in implementation. Almost all efforts in this proof are devoted for defining a proper domain and configuring our theorem prover to reason about interactions between primitives in that domain.

One can argue that we could have saved effort by reasoning about this program at a higher level. We agree with this view. However, the effort expended to configure ACL2 to reason about this simple program does not have to be repeated. We have developed an ACL2 "book" (a file containing definitions and lemmas) that codifies the necessary "concepts" and "knowledge", and configures ACL2 to reason about straight-line programs automatically. We thus have high confidence in our semantics and can reason about it without difficulty. In fact, we have proved properties of a different piece of straight line program that computes `(int-fix (+ 4 (* 2 old)))`, with 100 lines[6].

3.2 Recursive Factorial Program

In this section, we briefly discuss our experience with a second proof effort that reuses the definitions and lemmas developed in the `ADD1` program proof. The program computes the factorial of its input, or, to be more precise, it computes the signed integer representation of the low order 32-bits of the mathematical factorial.

The program of interest is as follows

```
(class "Second"
  ....
  (method "fact"
      (parameters int)
      (returntype int)
      ....
      (code
          ....
         (parsedcode
         (0 (iload_0))
         (1 (ifgt 6))  ;;to TAG_0
         (4 (iconst_1))
```

[6] Details are available from the supporting materials.

```
        (5 (ireturn))
        (6 (iload_0)) ;;at TAG_0
        (7 (iload_0))
        (8 (iconst_1))
        (9 (isub))
        (10 (invokestatic (methodCP "fact" "Second" (int) int)))
        (13 (imul))
        (14 (ireturn))
        ....))))
```

This program is still very simple conceptually but much more complicated than the `ADD1` program. We proved the following theorem

```
(defthm second-is-correct
  (implies (and (poised-for-execute-fact s)
                (wff-state-regular s)
                (wff-thread-table-regular (thread-table s))
                (no-fatal-error? 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))))))
```

The theorem may be read as follows. Let `s` be a state poised to invoke our `fact` method, i.e., whose next instruction is an `invokestatic` of `fact`. Suppose the state is in some suitable sense well-formed, that `n` is a nonnegative 32-bit integer and that `n` is on top of the stack. Run `s` a certain number of steps, namely `(fact-clock n)`. The result is a state that could be alternately obtained by incrementing the pc of `s` by `3` (the number of bytes in the `invokestatic` instruction), popping the stack (to remove `n`), and pushing the `int` representation of `(fact n)`. Here `fact` is defined in the logic as the standard mathematical factorial.

What is new in the program is that it involves the method invocation that changes the call stack of the current thread.

What may at first be surprising is that the second proof is much shorter than the proof about the `ADD1` program. One reason is because in the first proof we reasoned about a round robin scheduler. However, the more essential reason is that we reused our results from the first proof about straight line code.

To explain, it is necessary to describe how ACL2 works. ACL2 is a semi-automatic theorem prover. The user submits definitions, and formulas that are asserted to be theorems. The system attempts to establish the legality of each definition and the validity of each alleged theorem. When a formula is proven to be a theorem it is converted into a rule and stored in the database. In most cases, a rewrite rule is generated. By submitting an appropriate sequence of lemmas

the user can configure ACL2 to prove theorems with certain strategies. The sequence of interactions is called a session. The file containing the definitions and lemmas is called a "book." Subsequent sessions may begin by including books taken "from the shelf." The ACL2 distribution contains many standard books on arithmetic, sets, vectors, floating point, etc. Using the `ADD1` program as a "challenge" problem, we created a book that codifies how to reason about straight-line programs that modify only the operand stack and locals.

To prove `fact` correct we start with the basic `ADD1` book (or just continue the `ADD1` session) and follow the same strategy. We introduce the abstraction of `pushFrame`, `popFrame`; we introduce a new state equivalence that captures what does not change during a call stack manipulation; and we prove theorems to guide ACL2 reasoning about compositions of operand stack primitives with call stack primitives.

The surprise in this proof effort is that the semantics of invoking a method in JVM (and Java) is rather rich. It involves dynamic class resolution, which in turn relies on primitives that load a class. Moreover, loading a class is related to creating objects dynamically in the heap. Thread synchronization and class initialization are also involved. We spend a major part of our efforts in reasoning about those primitives. In the final theorem, we assume `s` is a state in which the class is already loaded by asserting the starting state is "equivalent" to some constant state where the "Second" class is loaded. More details and some explanations are available in the supporting material [6] for this paper.

4 Review and Related Work

The challenge in using a deep embedding of a realistic programming language like JVM bytecode is managing the complexity at proof-time. We presented two proofs to show that the apparent complexity involved in the deep embedding can be alleviated by introducing the necessary abstractions and proving properties of those primitives in an identified domain.

Identifying the appropriate abstractions is relatively simple. Most work involves correctly identifying the domain where the primitives behave according to the intuition of the user. Another major effort is establishing properties of the abstract primitives and configuring the theorem proving engine to use them (typically, but not exclusively) as rewrite rules.

The main limitation of the current work is that we have not yet developed a good and concise set of primitives and their properties. Even though the proof of the `ADD1` program is automatic, it is quite long. On the other hand, our experience with the factorial program proof shows that even with the non-optimized set of lemmas, one can still benefit from the support of the computer aided reasoning tool. In developing the lemmas for the factorial program proof we do not have to think about how to reason about straight-line code — a problem solved once and for all in the `ADD1` proof. In the factorial proof we focus on the primitives that manipulate the call stack.

We have not explored proofs about more complicated JVM operations in our model, such as allocation of new objects in the heap or synchronization using monitors. Programs using such primitives have been verified with ACL2 using a simpler JVM model that does not include our modeling of dynamic class loading and exceptions [16].

The sample proofs presented here are proofs for complete correctness. The lemma library about primitives can be reused for proving partial correctness of Java programs. In his CHARME 2003 paper [15], Moore shows that Floyd-Hoare style assertion based program proofs can be constructed directly from the formal operational semantics with little extra logical overhead, i.e. with no need to write a verification condition generator or other meta-logical tools. To make effective use of the operational semantics in place of a conventional verification condition generator, ACL2 needs to be configured to simplify the compositions of JVM primitives. Thus the present work may be viewed as a follow-up to Moore's work on Floyd-Hoare style proofs for bytecode programs on a very complete JVM model.

In addition to using our model to verify properties of bytecode programs, we are using it to explore the correctness of the JVM bytecode verifier. In our approach, defining a realistic JVM is one of the necessary steps in that effort. This is an additional justification for the choice of a deep embedding: it allows us to state and prove "meta-level" properties. For existing works on formalizing bytecode verification, the special issue on Java bytecode verification from the Journal of Automated Reasoning is a good reference [17].

The collection of "Formal syntax and semantics of Java" edited by Alves-Foss contains many early works in formalizing the Java programming language [4]. The Java Language Specification [10] provides the informal specification. Although we feel it is hard to formalize a complex language by designing an axiomatic semantics, the LOOP project [21] has formalized the semantics of Java and a Java annotation language JML based on *coalgebra*. They are also deriving proof rules in the style of Hoare logic for embedding Java into PVS [8].

To us, a more feasible method is to give Java an operational semantics. In [9], Attali et.al. present an operational semantics for Java using the structural operational semantics approach [19]. We think that our operational semantics given by state transformation appeals to human intuition better than the operational semantics based on structural transformation. This in turn makes it easier to validate the formal semantics against informal specifications and benchmark implementations. In addition, we feel that a structural operational semantics would be awkward to support in ACL2.

Börger et. al use abstract state machines for modeling the dynamic semantics of Java [20]. This work seems close to our work at the JVM level. The work by T. Nipkow, et. al., on μJava [24] and the Bali project[5] embeds a subset of Java and the Java bytecode language into the theorem prover Isabelle/HOL to reason about the type safety of those languages. Recently, J. Meseguer's group in UIUC has used the rewriting logic and engine Maude [2] to formalize the semantics of Java and the JVM [14].

In contrast to the above efforts, our work presented in this paper is focused on modeling an executable JVM and reasoning about Java program via the direct and intuitive state transformation semantics of its corresponding bytecode program on the JVM model.

5 Conclusion

To use a general purpose theorem prover in formal program verification the semantics of a program to be verified must be expressed in the language of the theorem prover's logic.

In this paper, we show that one can deeply embed the Java bytecode language, a fairly complicated language with rich semantics, into the first order logic of ACL2 by modeling a realistic JVM. We reason "about Java programs" by compiling them with Sun's `javac` and then reasoning about the bytecode.

We claim that this is a viable approach in doing Java program verification. One of the obvious advantages of deep embedding is that its operational nature makes the semantics correspond closely to informal descriptions in the JVM specification and with benchmark implementations, increasing one's confidence in the model. The behavior of programs under the model and properties of the model are then derived by the theorem proving engine, increasing one's confidence that the reasoning is sound.

The central question is whether we can effectively deal with the complexity introduced by this approach. We show that with the support of a user guided, semi-automatic computer proof assistant, the user can reason about programs at a fairly intuitive level. In a system like ACL2 the necessary support can be arranged by defining appropriate abstractions and proving lemmas about them for automatic use by the system. We demonstrate this by covering two concrete proofs, with the later one reusing the results of the first one as a lemma library.

We feel the limitation of the current work is that our lemma libraries for reasoning about Java programs are still unoptimized and only cover selected JVM primitives. Our current focus has been in formalizing the correctness of the Java bytecode verifier. We are looking forward to extending our work to provide a full-fledged Java verification system.

Acknowledgement Authors would like to thank our reviewers for their valuable advices. We would also like to thank our group members in Austin Serita Van Groningen, Robert Krug and Omar El-Domeiri, for their efforts in "debugging" the paper with us.

References

1. R. S. Boyer and Y. Yu. Automated proofs of object code for a widely used microprocessor. *Journal of the ACM*, 43(1):166–192, January 1996.
2. M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and J. Quesada. *A Maude Tutorial*. SRI International, 2000.

3. J. Crow, S. Owre, J. Rushby, N. Shankar, and M. Srivas. A tutorial introduction to PVS. In *Workshop on Industrial-Strength Formal Specification Techniques*, Boca Raton, FL, April 1995.
4. J. Alves Foss, editor. *Formal Syntax and Semantics of Java*, volume 1523 of *LNCS*. Springer-Verlag, 1999.
5. G. Klein, T. Nipkow, D. von Oheimb, C. Pusch, and L. P. Nieto. Project Bali. Available from `http://isabelle.in.tum.de/bali/`, May 2004.
6. H. Liu and J S. Moore. Supplement: proof scripts, etc. `http://coldice.csres.utexas.edu/~hbl/tphol2004/`, Feb 2004.
7. C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, 1969.
8. Marieke Huisman and Bart Jacobs. Java program verification via a Hoare logic with abrupt termination. *Lecture Notes in Computer Science*, 1783:284+, 2000.
9. I. Attali, D. Caromel, and M. Russo. A formal executable semantics for java. In *Proceedings of Formal Underpinnings of Java Workshop, OOPSLA'98*, 1998.
10. J. Gosling, B. Joy, G. L. Steele Jr., and G. Bracha. *The Java Language Specification*. Addison-Wesley Publisher, second edition, 2000.
11. 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, ECOOP 2001*. 2001.
12. H. Liu and J S. Moore. Executable JVM model for analytical reasoning: a study. In *Proceedings of the 2003 workshop on Interpreters, Virtual Machines and Emulators*, pages 15–23. ACM Press, 2003.
13. M. Kaufmann, P. Manolios, and J S. Moore. *Computer-aided Reasoning: An approach*. Kluwer Academic Publishers, 2000.
14. J. Meseguer. Rewriting logic based semantics and analysis of concurrent programs. Talk at UT-Austin, Feb 2004.
15. J S. Moore. Inductive assertions and operational semantics. In *the 12th Advanced Research Working Conference on Correct Hardware Design and Verification Methods (CHARME 2003)*, volume 2860 of *LNCS*, October 2003.
16. J S. Moore. Proving theorems about Java and the JVM with ACL2. In M. Broy and M. Pizka, editors, *Models, Algebras and Logic of Engineering Software*, pages 227–290. IOS Press, Amsterdam, 2003. `http://www.cs.utexas.edu/users/moore/publications/marktoberdorf-03`.
17. T. Nipkow, editor. *Java Bytecode Verification*, volume 30(3-4), 2003.
18. L. C. Paulson. *Isabelle: a generic theorem prover*. Springer-Verlag, 1994.
19. G. Plotkin. A structural approach to operational semantics. Technical report, University of Aarhus, Denmark, 1981.
20. R. F. Stärk, J. Schmid, and E. Börger. *Java and the Java Virtual Machine: Definition, Verification, Validation*. Springer, 2001.
21. University of Nijmegen Security of System Group. LOOP project. `http://www.cs.kun.nl/~bart/LOOP/`.
22. Connected Limited Device Configuration (CLDC) Specification 1.1. `http://jcp.org/en/jsr/detail?id=139`.
23. T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. Addison-Wesley Publisher, second edition, 1999.
24. T. Nipkow, D. Oheimb, and C. Pusch. μJava: Embedding a programming language in a theorem prover. In F. L. Bauer and R. Steinbrüggen, editors, *Foundations of Secure Computation*, volume 175 of *NATO Science Series F: Computer and Systems Sciences*, 2000.