ELSEVIER

Future Generation Computer Systems 16 (2000) 841–850

FGCS
FUTURE GENERATION COMPUTER SYSTEMS

Dynamic semantics of Java bytecode

Peter Bertelsen

Department of Mathematics and Physics, Royal Veterinary and Agricultural University, Copenhagen, Denmark

Accepted 24 May 1999

Abstract

We give a formal specification of the dynamic semantics of Java bytecode, in the form of an operational semantics for the Java Virtual Machine (JVM). For each JVM instruction we give a rule describing the instruction's effect on the machine state and the conditions under which the instruction will execute without error.

This paper outlines the formalization of the JVM machine state and illustrates our approach for a few select JVM instructions. Our full specification, covering the entire JVM instruction set except for synchronization instructions, is available in the work of Bertelsen (Semantics of Java byte code, Technical Report, Department of Mathematics and Physics, Royal Veterinary and Agricultural University, Copenhagen, Denmark, April 1997).

Keywords: Java; Java Virtual Machine; Formal specification; Semantics

1. Introduction

The Java Virtual Machine (JVM) is a virtual machine for safely implementing object oriented languages. It is rather complicated because each instruction must check a number of conditions. For instance, the `getstatic` instruction, which accesses a static field of a class, must check that the class is accessible to the current method, that the class has been loaded, that the class declares the requested field, that the field is accessible to the current method, etc.

The original definition of the JVM was given in the book by Lindholm and Yellin [10] and refined in the second edition of the book [11]. To describe the pre-conditions and the effect of JVM instructions, the book uses natural language, pseudo-C constructs and runtime stack pictures. Consequently, Lindholm and Yellin's book is rather long (475 pages) and it is hard to fully understand the semantics of the JVM, e.g. the precise conditions for executing an instruction.

The objective of the present work was to gain a thorough understanding of the JVM as described by Lindholm and Yellin and to express that understanding clearly and compactly. Another goal was that of uncovering possible omissions and ambiguities in the JVM specification. Hence, we describe the JVM at a higher level of abstraction, using ordinary mathematical concepts, and we use a semi-formal notation rather than natural language. Our full specification [2] is less than 80 pages long.

1.1. What is covered by our specification

We formalize the JVM machine state and for every JVM instruction, we describe the conditions for its successful execution and its precise effect on the machine state. We do *not* describe the following aspects of the JVM:

* *E-mail address:* pmb@dina.kvl.dk (P. Bertelsen)

0167-739X/00/$ – see front matter
PII: S0167-739X(99)00094-1

- class file verification: what checks can or must be done at class loading time;
- multiple threads, the `monitorenter` and `monitorexit` instructions;
- what happens when a pre-condition of an instruction fails (e.g. which exception is thrown);
- the Java Class Library, native methods and garbage collection.

The JVM instruction set does not include any instructions for starting, suspending, or stopping a thread; these mechanisms are supported only via methods in the Java Class Library [4]. Hence, a formalization of the JVM semantics covering multi-threading would have to include the semantics of (parts of) the Java Class Library as well. In our specification, we focus solely on the JVM and consider only one single thread of execution.

1.2. Format of the specification

Our formalization uses ordinary mathematical concepts: partial functions,[1] sets, sequences, disjoint sums, etc. The presentation is, however, semi-formal in that some details are left undefined, e.g. the precise semantics of integer operations. As a consequence of this, our formalization of the JVM semantics has not been machine checked.

Alternatively, one could use a particular specification language, such as VDM [9], or develop a theory within a theorem prover, such as HOL [8], Isabelle [13] or PVS [12]. This would make the specification even more precise and would provide for validation of the semantics by mechanical proof-checking.

However, the resulting specification would probably be less accessible to the general reader. The approach that we have chosen is semi-formal and hence less precise, but hopefully more comprehensible.

2. Modelling the JVM machine state

In this section we outline a formal specification of select JVM instructions. The details of our notation and the precise definition of auxilliary functions used in the following sections are presented in the full report [2].

[1] We use the notation $A \xrightarrow{\text{fin}} B$ for a partial function from A to B.

Our specification is more abstract than that described in the 'official' JVM specification [10,11], yet describes the JVM semantics at a level very close to the actual instruction set, including many details of the bytecode instructions.

In our model, the run-time state of the JVM consists of two parts: the global environment, which remains fixed, and the thread state, which changes during execution. Note that we model only a single thread of execution.

2.1. The global environment

The *global environment* maps class names to class files. Concretely, the global environment represents the file system and the network, from which class files may be loaded. In our modelling, a class file has eight components:

$$C = \mathcal{P}(Acc_c) \times [Id_c] \times \mathcal{P}(Id_c) \times FD \times CV \times MD \times MI \times CP$$

The components of a class or interface C are its access modifiers ($\mathcal{P}(Acc_c)$), the name of its direct superclass ($[Id_c]$), the names of its direct superinterfaces ($\mathcal{P}(Id_c)$), its field declarations (*FD*), its constant values (*CV*), its method declarations (*MD*), its method implementations (*MI*) and its constant pool (*CP*). The constant pool holds string literals, 'large' constants and symbolic references to classes, interfaces and members.

The precise definitions of the sets $\mathcal{P}(Acc_c)$, $[Id_c]$, etc. are given in the full report [2] in the same style.

2.2. The thread state

The *thread state* is the state of an executing thread. We model the thread states *TS* as follows:

$$TS = Frame^* \times Heap \times Env$$

A thread state consists of a frame stack ($Frame^*$), a heap (*Heap*) and an environment (*Env*). Each frame in the frame stack corresponds to a method invocation. The topmost frame is the frame of the currently executing method.

A frame $f \in Frame$ contains a program counter $pc \in PC$, an operand stack $s \in Oper^*$, a local variable table $l \in Locals$ and the current method's class name

$id_c \in Id_c$ as well as its signature (method name and argument types) $sig \in Sig$:

$$Frame = PC \times Oper^* \times Locals \times (Id_c \times Sig)$$

An operand (*Oper*) is either a word (*W*) or a double-word (*DW*). A word is either a proper word (W_v) or a program counter value (*PC*). A proper word is either an integer[2] (*Int*), a float (*Float*) or a reference (Ref_0) which is possibly null. A double-word is either a long integer (*Long*) or a double (*Double*).

A local variable table (*Locals*) maps a non-negative integer index to the (one- or two-word) local variable value (*Oper*) at that index. A two-word value occupies two entries in the table.

A heap $h \in Heap$ maps a non-null reference (*Ref*) to an object:

$$\begin{aligned} Heap &= Ref \xrightarrow{\text{fin}} Obj \\ Obj &= Obj_u \cup Obj_a \\ Obj_u &= Id_c \times IV \\ Obj_c &= Id_c \times IV \\ IV &= (Id_c) \times Id_f) \xrightarrow{\text{fin}} V \\ V &= W_v \cup DW \end{aligned}$$

An object (*Obj*) is either an instance of a class type or an instance (Obj_a) of an array type; we distinguish between uninitialized (Obj_u) and initialized (Obj_c) instances of class types.

An object of class type consists of the name (Id_c) of the class and its instance field values (*IV*). The latter maps a class name (Id_c) and a field name (Id_f) to the value (*V*) of that instance field. The value of a field must be a proper value (*V*): it must be either a proper word (W_v) or a double-word (*DW*). Hence, a program counter value (*PC*) cannot be stored in a field.

Array objects are described as maps from non-negative indices to proper values (*V*). A multi-dimensional array is represented as an array of references to arrays: each element of the 'topmost' array is a reference to an array object at the next level, etc.

The environment $e \in Env$ in a thread state holds the classes that have been loaded by the JVM from the global environment (e.g. from the file system). It maps a class or interface name to its declaration (*C*) and static field values table (SV):

$$Env = Id_c \xrightarrow{\text{fin}} (C \times SV)$$

$$SV = Id_f \xrightarrow{\text{fin}} V$$

A table of static field values (*SV*) maps the field name (Id_f) of a class or interface field to its value.

3. Formalizing the effect of JVM instructions

The effect of JVM instruction execution on the thread state is described in the style of small-step operational semantics [15]. Each JVM instruction is defined by an inference rule, as illustrated by rule (1) below. The premises above the line describe the conditions that must hold for the instruction to execute successfully, that is, without throwing an exception. The conclusion $ts \Rightarrow ts'$ below the line states that execution of the instruction will change the thread state from *ts* to ts'.

In each of the rules, the symbols s, l, m, fr, h and e refer to the components of the thread state $ts = ((pc, s, l, m) :: fr, h, e)$.

By convention, the premises are read from the top down and from left to right. Although immaterial from a logical point of view, this suggests a more operational interpretation of the rules.

3.1. Example 1: the dup instruction

The JVM instruction `dup` duplicates the topmost stack operand:

$$\frac{\begin{array}{l} instr(ts) = \texttt{dup} \\ s = (v : W) :: sr \\ size_s + size_v \leq max_s(ts) \\ succ(ts) = pc' \end{array}}{ts \Rightarrow ((pc', v :: v :: sr, l, m) :: fr, h, e)} \quad (1)$$

If all of the premises above the line hold, then the current thread state *ts* will change into the state specified after the $\Rightarrow$ symbol.

The premise $instr(ts) = \texttt{dup}$ states that this rule applies to the `dup` instruction.

The premise $s = (v : W) :: sr$ asserts that the operand stack s in *ts* has top-most element v

[2] The JVM uses integers (*Int*) to represent the Java types `boolean`, `byte`, `char`, `short` and `int`.

and remainder *sr*, and that *v* is a word (*W*), not a double-word. [3]

The premise $size(s) + size(v) \leq max_s(ts)$ asserts that the operand stack will not overflow: the combined sizes of the old stack *s* and the duplicated value *v* does not exceed the declared maximal size of the operand stack [4] for the current method.

The premise $succ(ts) = pc'$ asserts that there is a successor instruction and that its address is pc'.

For brevity, the rules use a number of semantic utility functions. For instance, *instr*(*ts*) finds the current instruction in the current thread state *ts*, *size* computes the size (in words) of a semantic object, $max_s(ts)$ finds the declared maximal stack size for the method currently executing, *succ*(*ts*) finds the address of the next bytecode instruction to execute, etc. Formal definitions of these functions (and other functions used below) are given in the full report [2].

The notation $v : W$ in the second premise means that the value *v* has type *W*. This notation is used in two ways: (1) to assert a condition and (2) to tag a value with a given type. For instance, 117: *Double* is the number 117 of type *Double* (as opposed to, e.g. *Int* or *Long*).

3.2. Example 2: the `istore` instruction

The JVM instruction `istore` j removes the top-most (integer) operand from the stack and stores it in local variable number *j*:

$$\frac{\begin{array}{l} instr(ts) = \texttt{istore}\ j \\ s = (k : Int) :: sr \\ j < max_l(ts) \\ succ(ts) = pc' \end{array}}{ts \Rightarrow ((pc', sr, rmDW(l, j) + \{j \mapsto k\}, m) :: fr, h, e)} \quad (2)$$

The premises state that this rule concerns the `istore` instruction, that the top-most value on the stack *s* must exist and be an integer *k*, that *j* must be within the declared range of local variable indices [5] and that the current instruction must have a successor at pc'. If so, the thread state changes to $((pc', sr, rmDW(l, j) + \{j \mapsto k\}, m) :: fr, h, e)$ in which *k* has been popped off the stack and local variable *j* has been changed to *k*.

If writing into local variable *j* happens to destroy the second half of a double-word value, then the first half of that double-word must be removed from *l*. This is handled by the semantic function $rmDW(l, j)$.

3.3. Example 3: the `baload` instruction

The `baload` instruction loads an element from an array of element type `byte` or `boolean`:

$$\frac{\begin{array}{l} instr(ts) = \texttt{baload} \\ s = (k : Int) :: (r : Ref) :: sr \\ h(r) = ((1, \texttt{byte}), k', av) : Obj_a \\ av(k) = k'' : Int \\ succ(ts) = pc' \end{array}}{ts \Rightarrow ((pc', k'' :: sr, l, m) :: fr, h, e)} \quad (3)$$

If the top-most stack operand is an integer *k*, if the second top-most stack operand is a reference *r* to a one-dimensional array object of element type `byte` or boolean and if *k* is a valid index into the array, then the value *k* and the array reference *r* are popped off the stack, the array component k'' at index *k* is pushed onto the stack and execution continues with the next instruction.

We model arrays of element type `boolean` as having element type `byte`. This corresponds to the representation used in early implementations of the JVM.

The component k' of the array object at position *r* in the heap *h* is the length of the array object. It is immaterial in the rule for `baload` since the premise $av(k) = \ldots$ ensures that *k* is in the domain of *av* (which maps an array index to the corresponding array element).

Note that an integer value (*Int*) is loaded from the array. It is assumed that a component of a `byte` or boolean array has already been truncated to a `byte` or boolean value, respectively, and then expanded back

[3] The JVM instruction `dup` cannot be used for duplicating a double-word stack operand. Instead, the JVM instruction `dup2` must be used.

[4] In a Java class file, a maximum operand stack size is given for each method. For each invocation of the method, the operand stack cannot exceed the specified limit. The Java class file format and the JVM do not have special support for recursive methods.

[5] In a Java class file, a maximum number of local variables is given for each method. A local variable with an index greater than or equal to the specified limit cannot be used within the method.

into an integer value. We specify the truncation and expansion to/from `byte` and boolean values in connection with the `bastore` instruction (not shown here). See the full report [2] for details.

3.4. Example 4: the `new` instruction

The JVM instruction `new` i creates an instance of the class specified by the constant pool entry at index i:

$$\frac{\begin{array}{l} instr(ts) = \texttt{new}\,i \\ pool(ts)(i) = id'_c : Id_c \\ e(id'_c) = ((acc_c, id''_c, is, fd, cv, md, mi, cp), sv) \\ acc_c \cap \{\texttt{interface}, \texttt{abstract}\} = \emptyset \\ access(id'_c, acc_c, id_c) \\ r \in Ref \backslash dom(h) \\ fields(id'_c, e) = iv \\ (id'_c, iv) = o : Obj_u \\ size(s) + size(r) \leq max_s(ts) \\ succ(ts) = pc' \end{array}}{ts \Rightarrow ((pc', r :: s, l, m) :: fr, h + \{r \mapsto o\}, e)} \quad (4)$$

If it holds that

- i is a valid index into the constant pool of the current class,
- the constant pool entry at index i is a class (or interface) reference id'_c,
- the declaration of id'_c is in the domain of the environment e (i.e., has been loaded),
- id'_c is an instantiable class (not `abstract` or `interface`),
- the current class id_c can access class id'_c, and
- there is an unused location r in the heap

then the execution of the `new` instruction proceeds:

- the instance fields of class id'_c and all its superclasses are prepared using the auxiliary function *fields*;
- an instance o of class id'_c with instance field values iv is created;
- the reference r is pushed onto the operand stack;
- o is bound at location r in the heap; and
- execution continues with the next instruction at address pc'.

The new object o is tagged with type Obj_u, which means that it is uninitialized. Members of the object cannot be accessed until it has been initialized by invocation of an instance initialization method. [6]

[6] An instance initialization method has the special method name < `init` >. Such a method corresponds to a constructor in Java.

The above rule does not specify any details with respect to memory allocation or garbage collection. Instead, we assume an infinite heap in which a fresh heap location r is always available. This is similar to the JVM specification [10,11], which does not mandate any particular memory management technique, but assumes the presence of an automatic memory management system, e.g. using garbage collection.

The situation where the class id'_c to be instantiated by a `new` instruction has not already been loaded is described in a separate rule (not shown here). See the full report [2] for details.

3.5. Example 5: the `getstatic` instruction

The `getstatic` instruction pushes the value of a class or interface field onto the stack:

$$\frac{\begin{array}{l} instr(ts) = \texttt{getstatic}\,i \\ pool(ts)(i) = (id'_c, id_f, d) : Const_f \\ e(id'_c) = ((acc_c, id''_c, is, fd, cv, md, mi, cp), sv) \\ access(id'_c, acc_c, id_c) \\ fd(id_f) = (acc_f, d') \\ d = d' \\ \texttt{static} \in (acc_f) \\ \texttt{private} \in acc_f \Rightarrow id_c = id'_c \\ \texttt{protected} \in acc_f \Rightarrow id'_c \in supers(id_c, e) \\ sv(id_f) = v : V \\ size(s) + size(v) \leq max_s(ts) \\ succ(ts) = pc' \end{array}}{ts \Rightarrow ((pc', v :: s, l, m) :: fr, h, e)} \quad (5)$$

If it holds that

- i is a valid index into the constant pool of the current class,
- the constant pool entry at index i is a symbolic field reference, referring to a field id_f with type descriptor d in class or interface id'_c,
- the name id'_c is in the domain of the environment e (i.e., the declaration of a class or interface of that name has been loaded),
- the current class id_c can access class/interface id'_c,
- class/interface id'_c declares a field id_f of the same type as that specified by the descriptor d,
- the field id_f is `static`,
- the field id_f is not `private` unless the current class is the same as that declaring the field, and

- the field id_f is not `protected` unless the current class is the same as that declaring the field or a subclass thereof

then the value v of the field id_f is retrieved from the static values sv of class/interface id'_c, v is pushed onto the operand stack and execution proceeds with the next instruction.

It is possible to get a value from a field before putting anything into it; the value loaded from the field will then be the initial (default) value corresponding to the type of the field.

In our modelling, a field reference must refer directly to the class or interface that declares the field. Hence, in the above rule, it is not sufficient for class/interface id'_c to inherit the field id_f from a superclass or superinterface. The specification of this in the original JVM specification [10] was unclear; in the second edition of the JVM specification [11] it is specified that a field reference may refer to a class/interface which inherits the field from a superclass or superinterface.

3.6. Example 6: the `getfield` instruction

This example demonstrates one of the object- oriented features of the JVM. The `getfield` instruction pushes the value of an instance field onto the stack:

$$\begin{array}{l} instr(ts) = \text{getfield}\, i \\ pool(ts)(i) = (id'_c, id_f, d) : Const_f \\ e(id'_c) = ((acc_c, id''_c, is, fd, cv, md, mi, cp), sv) \\ \texttt{interface} \notin acc_c \\ access(id'_c, acc_c, id_c) \\ fd(id_f) = (acc_f, d') \\ \texttt{static} \notin acc_f \\ \texttt{private} \in acc_f \Rightarrow id_c = id'_c \\ d = d' \\ s = (r : Ref) :: sr \\ h(r) = (id'''_c, iv) : Obj_c \\ id'_c \in supers(id'''_c, e) \\ \texttt{protected} \in acc_f \Rightarrow (id'_c \in supers(id_c, e) \\ \wedge\, id_c \in supers(id'''_c, e)) \\ iv\ (id'_c, id_f) = v : V \\ size(sr) + size(v) \leq max_s(ts) \\ \hline succ(ts) = pc' \\ ts \Rightarrow ((pc', v :: sr, l, m) :: fr, h, e) \end{array} \tag{6}$$

The `getfield` instruction differs from the `getstatic` instruction in that the referenced field must be an instance field (it cannot be an interface or class field) and that the value of the field is retrieved from a specific class instance rather than via the environment e of the thread state.

If it furthermore holds that

- the top-most stack operand is a non-`null` reference r to an initialized object,
- the object is an instance of the class id'_c that declares the referenced field id_f, and
- the field is not `protected` unless the current class is the same as id'_c or a subclass thereof and the object is an instance of the current class,

then the object reference r is popped off the stack, the value v of the field id_f is retrieved from the instance values iv of the object, v is pushed onto the stack and execution continues with the next instruction.

3.7. Example 7: the `invokespecial` instruction

This example shows one of the four JVM instructions for method invocation. The `invokespecial` instruction may be used for invoking a constructor, a `private` instance method, or a method of the current class:

$$\begin{array}{l} instr(ts) = \texttt{invokespecial}\, i \\ pool(ts)(i) = (id'_c, sig', d) : Const_m \\ e(id'_c) = ((acc_c, id''_c, is, fd, cv, md, mi, cp), sv) \\ \texttt{interface} \notin acc_c \\ access(id'_c, acc_c, id_c) \\ sig' = (id_m, < d_1, d_2, d_3, \ldots, d_k >) \\ id_m \neq < \texttt{clinit} > \\ md(sig') = (acc_m, d', excs) \\ (id_m = < \texttt{init} > \ \vee\ \texttt{private} \in acc_m \\ \vee\ id_c = id'_c \ \vee\ id'_c \notin supers(id_c, e) \\ \quad \vee \texttt{super} \notin acc_c) \\ acc_m \cap \{\texttt{static}, \texttt{abstract}, \texttt{native}\} = \emptyset \\ \texttt{private} \in acc_m \Rightarrow id_c = id'_c \\ mi(sig') = (n_s, n_l, code', hdls') \\ s = as@sr \\ as = < a_k, a_{k-1}, a_{k-2}, \ldots, a_2, a_1 > @ < r : Ref > \\ size(as) \leq n_l \\ \forall 1 \leq j \leq k.(initialized(a_j, h) \\ \quad \wedge compatVal\,(d_j, a_j, h, e)) \\ h(r) = o : (Obj_u \cup Obj_c) = (id'''_c, iv) \\ (id_m = < \texttt{init} > \wedge\, o \in Obj_u) \ \vee \\ \quad (id_m \neq < \texttt{init} > \wedge o \in Obj_c) \\ id'_c \in supers(id'''_c, e)) \\ \texttt{protected} \in acc_m \Rightarrow (id'_c \in supers(id_c, e) \\ \wedge\, id_c \in supers(id'''_c, e)) \\ (min_{pc}(code'), <>, args(as), (id'_c, sig')) \\ = f' : Frame \\ \hline initObj(id'_c, sig', r, h) = h' \\ ts \Rightarrow (f' :: (pc, sr, l, m) :: fr, h', e) \end{array} \tag{7}$$

If it holds that

- i is a valid index into the constant pool of the current class,
- the constant pool entry at index i is a symbolic method reference, referring to a method in class id'_c with signature sig' and return type descriptor d,
- the declaration of class id'_c is in the environment e,
- the current class has permission to access class id'_c,
- the referenced method is not a class or interface initialization method, [7]
- class id'_c declares and implements a method with signature sig' and the same return type descriptor as that specified by the symbolic method reference,
- the method is not declared to be either `static`, `abstract` or `native`,
- the method is not declared to be `private`, unless the current class is the same as that declaring the method,
- the topmost k stack operands [8] are the method arguments $a_k, \ldots, a_1$,
- the next stack operand r is a reference to a class instance o in the heap h,
- the total size of the stack operands as is less than or equal to the local variable limit n_l of the invoked method,
- the method arguments $a_k, \ldots, a_1$ are assignment compatible with the corresponding k parameter type descriptors in the method signature sig' and do not refer to any uninitialized objects,
- the referenced object o is uninitialized iff the referenced method is an instance initialization method `<init>`,
- the class id'''_c of the object o is the same as id'_c or a subclass thereof, and
- the method is not declared to be `protected` unless the current class is the same as class id'_c or a subclass thereof and the class id''''_c implementing the method is the same as the current class or a subclass thereof

then the method arguments as are popped from the operand stack of the current frame, a new frame f' representing the invoked method is pushed onto the frame stack, and execution continues with the first instruction in the invoked method.

The new frame f' initially contains an empty operand stack and the values of the method parameters in the first local variables (initialized by means of the utility function *args*).

It is unclear from the official JVM specification [10,11] when an object is considered to have been initialized. We model this as happening when the instance initialization method of class `java.lang.Object` (the superclass of all other classes) has been invoked. To this end the utility function *initObj*, which is used in the last premise of the above rule, tags the object referenced by the stack operand r as initialized (that is, as having type Obj_c) in case the method being invoked is method `<init>` of class `java.lang.Object`.

The `invokespecial` instruction may also be used for invoking other kinds of 'special' instance methods than the ones indicated above; this is described in a separate rule in our full report [2].

3.8. Other instructions

In the above examples we have demonstrated our approach for JVM instructions operating on the stack, the local variables, arrays, the constant pool, the static fields of a class and the instance fields of an object. Our full specification comprises 62 rules of which the most complicated one describes the method invocation instruction `invokespecial` (cf. Section 3.7).

We do not define separate rules for all of the 201 JVM instructions since many of these are *very* similar to each other. Instead, we define rules for instructions representing the different 'families' of instructions and describe (in less formal terms) how the remaining instructions in the JVM instruction set differ from those.

4. Problems with the official JVM specification

Based on the insights obtained in the development of our formal specification of Java bytecode semantics, a number of errors and ambiguities in the original JVM specification [10] were discovered.

[7] A class or interface initialization method has the special method name `<clinit>` and corresponds to one or more static initializers in Java. Such a method cannot be invoked directly using any of the JVM method invocation instructions, but is executed automatically by the JVM when the class or interface is to be initialized.

[8] We use the notation $as@sr$ for the concatention of the sequences as and sr.

For example:

- The necessary integration between the JVM and a number of important classes in the Java Class Libraries [4] was not defined.
- The interface to native methods had not been specified.[9] It was also not specified how a JVM implementation that does not support native methods is supposed to handle class files that declare native methods.
- The specified algorithm for class/interface initialization could not handle the very important classes `java.lang.Object` and `java.lang.Class` since these classes are mutually dependant.
- It was not specified how `boolean` values are encoded, except that int values are used for this purpose. It was, for example, not defined whether 0 means false and 1 means true or if any other encoding is to be used (e.g. the opposite).
- It was not clear whether the values of fields and method parameters of type `boolean`, `byte`, `short` and `char` are guaranteed to be within the (integer) ranges indicated by these types.
- It was not specified whether a class file may contain more than one declaration of a member, e.g. two methods with the same name and argument types.
- It was specified that "execution never falls of the bottom" of a method. It was, however, not specified how 'clever' a JVM implementation must be in order to determine whether the last instruction in a method breaks this requirement.
- It was not clear when an instance of a class is considered to have been initialized (cf. section 3.7).
- It was unclear which instructions may operate on references to uninitialized class instances.
- It was not clear whether a JVM implementation must perform bytecode verification or when this should happen.
- It was not specified what the semantics of the `aastore`, `checkcast` and `instanceof` instructions is in case the 'source type' is an array type and the 'target type' is an interface type.
- It was not specified how initialization of `String` constants is supposed to take place in connection with execution of the `ldc` and `ldc_w` instructions. This is important in connection with the Java concept of 'interned' strings.
- It was not specified whether a `final` field may be stored into by a `putstatic` or `putfield` instruction.
- It was not specified whether the value of an interface field may be loaded by the `getstatic` instruction.
- Dynamic method lookup in connection with the `invokeinterface` and `invokevirtual` instructions was only outlined. Details in Sun's implementation of the JVM were briefly described, but not detailed enough to determine e.g. the order in which the relevant classes are searched for an implementation of the method in question.

The above issues — and many more — have been listed in the *Java Spec Report* [14]. The purpose of this web-site is to make unofficial errata for the 'core' Java platform specifications widely available.

Along with issues raised by other people, the above issues have also served as input to the revision of the original JVM specification. Many of them appear to have been solved in the second edition of the JVM specification [11].

5. Future work

Topics for further work towards a complete JVM specification include:

- Specifying what happens when a given premise of a rule fails to hold: which exception is thrown and at what point in time during execution. For each instruction, and for each premise that can fail, one may introduce an additional rule which lists those premises that do hold, a single premise that fails and the exception that must be thrown in that case. However, this will considerably increase the number of rules. A more suitable approach to describing the precise semantics of 'failing' JVM instructions should be considered.
- Specifying the bytecode verification conditions. The informal specification in the JVM specification [10,11] is rather unclear, especially concerning the verification conditions for local subroutines (the `jsr` and `ret` instructions), exceptions handlers and their interaction.
- Finding the 'early' premises for each JVM instruction: those that may be checked by a bytecode veri-

[9] Native methods are methods that are implemented in some other way than by Java bytecodes; for example, a native method could be written in C. The JVM loads the code for such a method in a platform-dependent way.

fier at load-time. Prove that bytecode which passes such a verifier cannot fail the 'early' premises, and then remove those premises from the rules for the dynamic semantics.
- Specify the semantics of parallelism (threads). This would require rethinking the specification, although many parts of the current specification could be reused (specifically, all parts not related to field or method access).

6. Related work

Börger and Schulte give a formal semantics of Java bytecode, factorized into a number of sub-languages [3]. Their JVM semantics serves as a basis for defining a compilation scheme from Java source programs to Java bytecode. In their specification, Börger and Schulte assume that the Java bytecodes have already been verified by a bytecode verifier. Hence, they deal with fewer details in the JVM instruction set, although their approach resembles ours to some extent.

Also closely related to our work is Cohen's *Defensive Java Virtual Machine*, an executable specification expressed in ACL2, developed at Computational Logic Inc. [5]. Cohen's specification is fully formal and hence more precise than ours, but leaves out many aspects of the JVM, yet is far longer than ours (385 pages). It is probably better suited for machine manipulation and less suited for human readers.

Stata and Abadi show how to formalize some aspects of Java bytecode verification as a type system [16]. Thus while our work concerns the dynamic semantics of Java bytecode, their work concerns its static semantics.

We are also aware of other related work, e.g. the Alves-Foss book on Java semantics [7] and Diehl's formalization of Java compilation [6], but have not yet compared our work to theirs.

7. Conclusion

We have gained a thorough understanding of the JVM in a relatively short time. This has been an invaluable aid when subsequently implementing Java bytecode generators. More concretely, the present work has served as a basis for the design of the SML-JVM Toolkit [1], a toolkit for manipulating Java class files and Java bytecode.

Furthermore, several ambiguities in the orignal JVM specification [10] have been revealed, many of which appear to have been resolved in the second edition of the JVM specification [11]. These ambiguities in the informal specification (and more) are listed in the unofficial *Java Spec Report* [14], whose section on the JVM specification owes much to the present work.

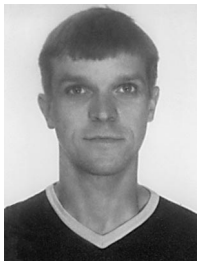
The dynamic semantics of Java bytecode presented here has not been validated formally. It shows, however, that the JVM can be given a fairly precise yet comprehensible description using well-known mathematical concepts and notation.

We believe that the standardization of the JVM would benefit from using a semi-formal specification style similar to that presented here.

References

[1] P. Bertelsen, The SML-JVM toolkit, version 0.5, Web-pages at `http://www.dina.kvl.dk/~pmb`. A toolkit for manipulating Java class files and Java byte-code.
[2] P. Bertelsen, Semantics of Java byte code, Technical report, Department of Mathematics and Physics, Royal Veterinary and Agricultural University, Copenhagen, Denmark, April 1997, Web pages at `http://www.dina.kvl.dk/~pmb`.
[3] E. Börger, W. Schulte, Defining the Java Virtual Machine as platform for provably correct Java compilation, MFCS Proceedings, 1998.
[4] P. Chan, R. Lee, D. Kramer, The Java Class Libraries, 2nd ed., vol. 1, Addision-Wesley, Reading, MA, 1998, ISBN 0-201-31002-3.
[5] R.M. Cohen, The Defensive Java Virtual Machine, version 0.5, Technical report, Computational Logic Inc., Austin, Texas, 1997, Web pages at `http://www.cli.com`.
[6] S. Diehl, A formal introduction to the compilation of Java, Software-Practice and Experience 28 (3) (1998) 297–327.
[7] J. Alves-Foss (Ed.), Formal Syntax and Semantics of Java, Springer, Berlin, 1998.
[8] M.J.C. Gordon, T.F. Melham (Eds.), Introduction to HOL: A Theorem Proving Environment for Higher Order Logic, Cambridge University Press, Cambridge, 1993, ISBN 0-521-44189-7.
[9] C.B. Jones, Systematic Software Development using VDM, Prentice-Hall, Englewood Cliffs, NJ, 1990, ISBN 0-13-880733-7.
[10] T. Lindholm, F. Yellin, The Java Virtual Machine Specification, Addison-Wesley, Reading, MA, 1996, ISBN 0-201-63452-X.
[11] T. Lindholm, F. Yellin, The Java Virtual Machine Specification, 2nd ed., Addison-Wesley, Reading, MA, 1999, ISBN 0-201-43294-3.

[12] S. Owre, S. Rajan, J.M. Rushby, N. Shankar, M.K. Srivas, PVS: Combining specification, proof checking, and model checking, in: R. Alur, T.A. Henzinger (Eds.), Computer-Aided Verification, CAV '96, New Brunswick, NJ, July/August 1996, Lecture Notes in Computer Science, vol. 1102, Springer, Berlin, pp. 411–414.
[13] L.C. Paulson, Isabelle: A Generic Theorem Prover, Lecture Notes in Computer Science, Springer, Berlin, 1994, ISBN 3-540-58244-4.
[14] R. Perera, P. Bertelsen, The unofficial Java Spec Report, Webpages at `http://www.dina.kvl.dk/~jsr`, September 1998.
[15] D. Gordon, A. Plotkin, A structural approach to operational semantics, Technical Report FN-19, DAIMI, Aarhus University, Denmark, 1981.
[16] R. Stata, M. Abadi, A type system for Java bytecode subroutines, Technical report, Digital Equipment Corporation Systems Research Center, July 1997.



Peter Bertelsen is a Ph.D. Student at the Department of Mathematics and Physics of the Royal Veterinary and Agricultural University in Copenhagen, Denmark. He has been investigating Java bytecode and the semantics of the Java Virtual Machine for more than two years, and is working towards automatic partial evaluation (aka specialization) of Java bytecode programs. Additionally, Peter Bertelsen is co-author of the Java Spec Report, an independent Internet site listing unofficial errata for the official Java specifications.