Improving the Official Specification of Java Bytecode Verification

Alessandro Coglio

Kestrel Institute
3260 Hillview Avenue, Palo Alto, CA 94304
http://www.kestrel.edu

coglio@kestrel.edu

Abstract

Bytecode verification is the main mechanism to enforce type safety in the Java Virtual Machine. Since Java security is based on type safety, inadequacies and ambiguities in the official specification of bytecode verification can lead to incorrect implementations where security can be broken. This paper analyzes the specification and proposes concrete improvements. The goal of this work is to increase the understanding, assurance, and usability of the Java platform.

1 Introduction

Bytecode verification is the main mechanism to enforce type safety in the Java Virtual Machine (JVM)[1]. Its purpose is to establish that certain type safety properties are always satisfied when bytecode is run. Therefore, the interpreter or just-in-time compiler can omit checks of such properties, resulting in better performance.

Because Java security is based on type safety [Gon99], correct implementation of bytecode verification is of paramount importance to the security of an implementation of the JVM. Holes in bytecode verification constitute potential points of attack exploitable by malicious programs (e.g., applets from remote sites of the Internet).

The official specification of the JVM [LY99], which includes a description of bytecode verification, is written in informal English prose. While this specification is mostly rather clear, it does contain some inadequacies and ambiguities. Their presence is particularly problematic for security-critical features such as bytecode verification, because erroneous interpretation can lead to erroneous implementation.

This paper exposes and analyzes the inadequacies and ambiguities in the official specification of bytecode verification in Section 2. It then proposes concrete improvements in Section 3. Concluding remarks are given in Section 4. For brevity, the official JVM specification is denoted by "JS". Individual chapters or (sub)sections of it are denoted by appending their number, e.g., "JS3.1". Some knowledge of JS is assumed.

[1]The other mechanisms, which complement bytecode verification, are resolution, class loading, as well as residual run-time checks (e.g., of array indices).

The goal of this paper, like others in the field, is to improve the understanding, assurance, and usability of the Java platform.

2 Analysis

JS4 describes the format of valid class files. Class verification is the process of checking that a byte sequence constitutes a valid class file. Class verification is described in JS4.9.1 as consisting of four passes. Passes 1 and 2 are in charge of checking the format of the class file, excluding those byte subsequences that constitute methods' code. The boundaries of these subsequences are identified during passes 1 and 2, but it is pass 3's responsibility to verify that each of them encodes valid code for an individual method. Pass 4 consists of the checks performed by resolution, i.e., that symbolic references to classes, fields, and methods are correct; even though it may not take place before execution, pass 4 is considered logically part of class verification.

Bytecode verification is pass 3. So, it is a part of class verification. In fact, it is the most interesting and delicate part; the rest of class verification is relatively straightforward and does not present major difficulties.

Bytecode verification is described in JS4.8 and JS4.9. JS4.8 presents constraints that a byte sequence must satisfy in order to represent valid method code. They are divided into *static* and *structural* constraints. JS4.9 explains, in a quite algorithmic way, how such constraints are checked. These descriptions are now analyzed.

2.1 Static Constraints

JS4.8.1 presents static constraints, which a byte sequence must satisfy in order to represent a sequence of correct bytecode instructions with correct opcodes and operands. All the instructions' operands are covered by the list of constraints in JS4.8.1, and the requirements are quite straightforward.

There is one requirement, though, that seems out of place. It states that the `new` instructions cannot be used to create an instance of an interface or of an abstract class.

This is true, but it is given in the context of restrictions on `new`'s operand. Since `new` references a class by name, the class should be resolved in order to determine whether it is an interface, an abstract class, or a non-abstract class. So, unlike all the other constraints given in JS4.8.1, this one cannot be checked on the operand alone.

Further evidence that the requirement is out of place is gathered from the following two observations. The first is that the specification of the `new` instruction in JS6 states that an exception is thrown if the resolved class is abstract or if it is an interface. This means that this check is done at run time.

The second observation is that other instructions that reference classes, fields, or methods in their operands also require checks similar to this one. For example, the `getfield` instruction requires the resolved field to not be static. This requirement is given as a run-time check in the specification of `getfield` in JS6, not as a static constraint in JS4.8.1. The same holds for the other instructions.

2.2 Structural Constraints

Static constraints constitute the straightforward part of bytecode verification. On the other hand, structural constraints, presented in JS4.8.2, constitute the interesting part, whose goal is to verify that certain type safety properties are satisfied when the code is executed. Structural constraints apply to instruction sequences. So, they assume that static constraints are satisfied. Remarks about structural constraints follow.

2.2.1 Terminology

The terms "static" and "structural", applied to the constraints given in JS4.8.1 and JS4.8.2, respectively, sound a little confusing and misleading. Both kinds of constraints are meant to be checked *statically*, without executing the code—in this case, prior to executing the code. So, using the adjective "static" only for the first kind of constraints does not seem ideal.

With a stretch, one could interpret structural constraints as requirements that must be satisfied at run time, even though they are checked statically, at verification time. However, static constraints must be satisfied at run time as well. This point about structural constraints referring to run time versus verification time is revisited in Section 2.2.2.

2.2.2 Undecidability

Many of the structural constraints express facts that must hold when certain instructions are executed. For example, when `getfield` is executed, the top of the operand stack must contain a reference to an object whose class is (a subclass of) the one specified by the instruction's operand.

However, establishing this kind of properties, taken literally, is undecidable. As it is well-known in program analysis, only conservative approximations can be algorithmically computed. Certainly, these structural constraints are meant to be checked only approximately. This supports the view that structural constraints, as stated, apply to run time and not to verification time, as mentioned in Section 2.2.1.

But structural constraints are given in the context of the format of valid class files, i.e., in JS4. So, it would seem more appropriate to provide constraints that can be computed, and that can be used as a public definition of valid class files. This is discussed in more detail in Section 3.2.2.

2.2.3 Redundancy

The first constraint of the list states that each instruction must be executed with an appropriate number and type of values in the operand stack and local variables. The notion of "appropriate" can be determined by the specification of the instruction given in JS6. For example, `getfield` requires the top of the operand stack to contain a reference to an instance of the class specified by the instruction's operand.

It turns out that the same requirement on `getfield` is stated explicitly by a separate structural constraint in the list. However, it is clearly redundant, because it can be derived from the first one.

Another constraint in the list states that no instruction must pop more values from the operand stack than it contains (i.e., no stack underflow must occur). Again,

this is a simple consequence of the first constraint: each instruction requires the presence of a certain number of values in the operand stack, and only those values are popped.

As a matter of fact, it could be argued that even the first constraint of the list is redundant. The specification of instructions given in JS6 includes the types of the values that each instruction expects to find in the operand stack and local variables. As explained in JS6.1, it is the task of bytecode verification to ensure that these expectations are met. So, the first constraint of the list, as stated, does not really say anything new.

2.2.4 Lack of Explanation

The motivation for several of the structural constraints is type safety. However, some constraints in the list do not appear directly related to type safety, and no explicit motivation is given for them in JS4.8.2.

An example is the constraint stating that no uninitialized object can be present in the operand stack or local variables when a backward branch is taken. It also states that no uninitialized object can be present in a local variable in code protected by an exception handler.

Other examples are the constraints related to subroutines: subroutines cannot be called recursively; the instruction following a `jsr` may be returned to only by a single `ret`; each return address can be returned to at most once; etc.

Yet another example is the constraint requiring that, if an instruction can be executed along different execution paths, the operand stack must have the same size prior to execution of the instruction.

Actually, *some* explanation for these constraints can be derived from JS4.9. This is discussed in Section 2.3.

2.2.5 Contradiction

There are two structural constraints that contradict each other. The first says that fields of an uninitialized object cannot be accessed; the object must be initialized first. But the second says that the code of a constructor can store values into some fields of the object that is being initialized—which, strictly speaking, is still uninitialized.

2.2.6 Possibly Out of Place

A structural constraint states that the `invokespecial` instruction must reference an instance initialization method, a method in the current class, or a method in a superclass of the current class. Checking if the method belongs to a superclass of the current class requires resolving the method and the class. Along the same line of reasoning for the constraint on `new` discussed in Section 2.1, it could be argued that this property must be checked at run time, not at verification time.

A structural constraint requires each method return instruction (such as `ireturn` and `freturn`) to match the method's return type. This is a simple check that does not involve type analysis[2]. So, this could well be a static constraint among those in

[2] Of course, type analysis is required to check that the operand stack always has a value of the right type when the instruction is executed. However, this is implied by the first structural constraint in the list, namely that each instruction must be executed with an appropriate number

JS4.8.1.

Another structural constraint is that execution never falls off the bottom of the instruction sequence. This is also rather simple to check and does not involve type analysis. Basically, the last instruction of the sequence must be one that cannot transfer control to the non-existent following one, directly or indirectly[3]. Some static constraints in JS4.8.1 ensure that the operand of each control transfer instruction (e.g., branches) points to an instruction in the code, and not outside it or in the middle of an instruction. For these reasons, the requirement that execution cannot fall off the end of code would fit well as a static constraint among those in JS4.8.1.

2.2.7 Heterogeneity

A final remark is that the structural constraints listed in JS4.8.2 are somewhat "heterogeneous". Most of them state properties that must hold when certain instructions are executed. Others state, instead, requirements relating all possible executions: for example, that the instruction following a `jsr` can be returned to by a single `ret`.

Heterogeneity is not necessarily bad. However, it certainly contrasts with the homogeneity of static constraints, which are all about instruction opcodes and operands[4].

2.3 Verification Algorithm

JS4.9 sketches an algorithm for bytecode verification. The first part of JS4.9.2 essentially explains how to check static constraints and is quite straightforward. A mildly interesting point is that the explanation includes checking that execution cannot fall off the end of code. This supports the view of this requirement as a static constraint, as argued in Section 2.2.6.

The second part of JS4.9.2 explains how to check a decidable approximation of structural constraints, by means of a data flow analysis [NNH98]. This description is followed, in JS4.9.3 through JS4.9.6, by some clarifications concerning the treatment of certain instructions. Remarks about these descriptions follow.

2.3.1 Merging of Operand Stack Types

Type information for the operand stack consists of a sequence of types, modeling the size and contents of the operand stack. The algorithmic description of the data flow analysis in JS4.9.2 prescribes that type information for the operand stack can be merged only when sizes are the same.

Though not the only possibility [Cog01b], this is certainly a sensible strategy that simplifies the analysis. In any case, it provides some explanation for one of the structural constraints mentioned in Section 2.2.4, namely that the operand stack must have the same size along all execution paths. So, that structural constraint is in some sense a "forward reference" to the data flow analysis algorithm. This somewhat challenges the view that structural constraints are just undecidable requirements that must hold at run time, as opposed to describing a computable approximation.

---

and type of values.

[3]The "indirect" case applies to `jsr`: control can be transfered to the following instruction when returning from the called subroutine.

[4]With the exception of the requirement on `new` discussed in Section 2.1.

This means that undecidable requirements and decidable approximations are mixed in JS4.8.2.

2.3.2 Reference Types

The algorithmic description of the data flow analysis talks about "reference types" that are assigned to operand stack positions and local variables, propagated through control flow paths, merged, etc. However, it fails to make it explicit whether these reference types are just names or names plus class loaders[5].

Given that class loaders are runtime objects, the only way for the algorithm to know the identities of the loaders would be to actually load the classes. This is in contrast with the general principle, stated in several points of JS, that classes may be loaded lazily, if and when they are required for execution.

Therefore, the most reasonable interpretation of the "reference types" used in the algorithm is that they are only names. However, it turns out that this lack of clarity is the origin of some type safety bugs [CG01, TH99].

2.3.3 Interface Types

The first edition of JS prescribes that the result of merging two reference types is their first common superclass or superinterface. This works fine for classes, but the first common superinterface of two interfaces may not be unique, because of multiple inheritance.

Therefore, in the second edition of JS the statement has been changed to just say that the result of merging two reference types is their first common superclass. Since `java.lang.Object` is considered a superclass of every interface, the result of merging two interfaces is always `java.lang.Object`.

But this requires a special treatment of `java.lang.Object` when it is the target of an `invokeinterface` instruction. Since `java.lang.Object` may derive from merging two interface types, bytecode verification should allow `invokeinterface` to operate on it. Otherwise, bytecode produced by Java compilers would be rejected.

Normally, `invokeinterface` is only allowed to operate on a class that implements the interface referenced by `invokeinterface` or on a subinterface of it—and `java.lang.Object` is none of them. In order to maintain type safety, a run-time check is necessary when an `invokeinterface` instruction is executed.

Two observations apply to this treatment of interface types. The first is that it is not particularly clean. The second is that JS does not make clear the implications of this approach, e.g., that run-time checks are necessary for `invokeinterface`.

2.3.4 Object Initialization

Bytecode verification must ensure that objects are initialized before they are used. In order to do that, the data flow analysis algorithm must use a special type for the reference to a newly created, uninitialized object. This special type is changed to a regular class type after the constructor is invoked. Since several copies of the object reference can be stored in the operand stack or local variables before invoking the constructor, all the occurrences of the special type must be changed.

[5] As stated in JS5.3, a class or interface in the JVM is identified by its (fully qualified) name plus its defining loader.

After an object of a class is created and before the object is initialized, another object of the same class could be created. The algorithm must distinguish between types for references to the first object and types for references to the second object. The solution prescribed in JS4.9.4 is to use the index of the `new` instruction (i.e., its position in the code) as part of the special type for uninitialized objects. In this way, since different objects are created by different instructions, they have different types assigned to them.

A potential source of trouble is a `new` instruction being part of a loop. If it is possible to go through the loop without initializing the object, how can the algorithm distinguish between objects created during two different iterations of the loop? To avoid this problem, JS4.9.4 prescribes that the algorithm must make sure that no type for an uninitialized object exists in the operand stack or local variables, whenever a backward branch is taken. For analogous reasons, it prescribes that no type for an uninitialized object must exists in a local variable in instructions protected by an exception handler[6].

This provides some explanation for the structural constraints mentioned in Section 2.2.4, about absence of uninitialized objects when backward branches are taken and in code protected by exception handlers. The discussion in Section 2.3.1 applies to them: they are forward references to the algorithm and they somewhat mix undecidable constraints with decidable approximations.

But the main point is that these restrictions on (types for) uninitialized objects are completely unnecessary. This is described in detail and formally proved in [Cog01c].

Intuitively, the reason is the following. Consider, for example, how the data flow analysis computes type assignments for the instructions of a loop. Starting from the first instruction of the method, a type for an uninitialized object can be only introduced by a `new` instruction. After it, the type can be copied around. If a backward branch is taken to an instruction preceding the object creation instruction, the types assigned to the branch instruction are merged with those at the target of the branch. Since the latter do not include the same type for the uninitialized object, such a type disappears as a result of merging. So, when the `new` instruction is reached again, there is no copy of the type for the uninitialized object introduced during the first iteration, and hence no confusion can arise.

2.3.5 Subroutines

Subroutines constitute the trickiest aspect of bytecode verification. The reason is that, in order to perform accurate type inference, the flow of control determined by subroutines must be taken into account. However, subroutines may not be textually delimited and may be exited implicitly by branching or throwing exceptions (i.e., not by a `ret`).

Here, "accurate type inference" means type inference that is both correct (i.e., does not allow type safety to be broken) and not excessively approximate. On one hand, simple-minded treatments of subroutines would ensure correctness but would reject programs produced by compilers. On the other hand, more elaborate treatments would accept all compiled programs, but would raise the risk that type

[6] No restrictions are given on the operand stack because the stack is emptied when an exception is thrown—the thrown exception is then pushed onto the stack.

safety could be broken, if the properties and implications of such treatments are not fully understood.

A detailed discussion of these issues can be found in [Cog01a], which also includes a thorough analysis of the treatment of subroutines described in JS4.9.6 and JS4.8.2, along with its motivations. Replicating this analysis here would make this paper inappropriately long: the reader is referred to [Cog01a].

In a nutshell, the main points are the following. In JS4.9.6 an approach to verify bytecode with subroutines is sketched. The structural constraints in JS4.8.2 that refer to subroutines can be considered to provide further information about the approach.

But the overall description omits important details, such as how to merge lists of `jsr` targets (which are associated to instructions, as prescribed in JS4.9.6) from converging control flow paths.

Some aspects of the approach, such as keeping track of modified variables by means of bit vectors, are explicitly motivated in JS4.9.6. Other aspects of the approach, such as prohibiting recursive subroutine calls, are not explicitly motivated. It turns out that the reason behind some of these unexplained restrictions is to guarantee type safety. However, others turn out to be unnecessary to this purpose.

In addition, the approach ends up rejecting bytecode that can be produced by compilers. For example, in Java 2 SDK 1.2 for Solaris bytecode verification rejects code produced by the compiler; see [Cog01a] for details.

The incompleteness of the description and the failure to clarify how some of the indicated restrictions serve to guarantee type safety, open the potential to incorrect interpretation leading to implementations where type safety could be broken. In addition, the complexity of the approach translates into complexity of the implementation, which requires more effort and is more susceptible to bugs and attacks. Last but not least, the rejection of bytecode produced by compilers is undesirable, as discussed in Section 3.2.2.

3 Improvements

In my opinion, there are two main ways in which the official specification of bytecode verification should be improved. First, the scope of bytecode verification should be clarified. Second, a precise characterization of it should be given.

3.1 Scope

The goal of bytecode verification is to statically establish that certain type safety properties always hold at run time. In this way, the interpreter or just-in-time compiler can omit checks of such properties, resulting in better performance.

The exact type safety properties that bytecode verification must establish can be determined from the specification of bytecode instructions in JS6. That specification includes statements using "must", such as "the top of the stack *must* contain a value of type `int`". As explained in JS6.1, the meaning of "must" is that the execution engine expects the expressed requirements to hold, and it is the task of class verification to make sure they indeed hold[7].

[7] There are a few exceptions to this statement, i.e., specifications of instructions in JS6 that use "must" for checks intended to be performed at run time. An example is the `aastore` instruction.

Some of these requirements are ensured by the resolution process. An example is that the method symbolically referenced by a method invocation instruction must exist, have the indicated argument and return types, and be accessible to the class where the method invocation instruction is.

Other requirements are ensured by checking the static constraints on method code given in JS4.8.1. An example is that the index of a local variable, used as an operand of an instruction, must be within the range of local variables for the method.

The remaining requirements are the type analysis portion of bytecode verification. An example is that the top of the operand stack must contain a value of type `int` when certain instructions are executed.

There are a few complications about the requirements ensured by the type analysis. These complications and their solutions are now discussed.

3.1.1 Disambiguation of Reference Types

As discussed in Section 2.3.2, JS fails to clarify whether bytecode verification should use names only or names plus loaders. The most reasonable interpretation, which has the advantage of allowing lazier class loading, is that bytecode verification uses names only.

This is correct as long as there is an intended disambiguation for class names, accompanied by mechanisms to enforce that the disambiguation in a method is consistent with the one in another method with which objects are exchanged. Consider the following example. A method m_1 has an argument of type `C` (a class name), whose intended disambiguation in m_1 is a class c (identified by the name, `C`, plus a loader). Now, suppose that another method m_2 calls m_1, passing an object of type `C` to it. If `C` is disambiguated to a class c' in m_2, it must be the case that $c = c'$. Otherwise, type safety could be broken.

In the first edition of JS, these issues were not mentioned. Type safety bugs related to these issues were found in earlier implementations of the JVM [Sar97, DFW96].

Those bugs were corrected by the introduction of loading constraints [LB98], described in JS5. Loading constraints ensure that classes exchanging objects, through methods and fields, agree on the actual classes of these objects, not only on their names. Loading constraints are external to bytecode verification. They are part of the class loading mechanisms, which complement bytecode verification to enforce type safety, together with resolution and residual run-time checks.

Formal evidence that bytecode verification can use names only and leave to loading constraints the task of avoiding ambiguities between classes with the same name, is given in [QGC00].

3.1.2 Merging of Reference Types

JS4.9.2 prescribes that the result of merging two class names `C` and `D` is (the name of) their first common superclass. This requires resolving `C` and `D` to actual classes, and

Its specification says that the type of the object to store must be assignment-compatible with the component type of the array. However, the same specification states that a run-time exception is thrown if that is not the case. So, despite the use of "must", it is clear that it is not the task of bytecode verification to ensure that property. The same applies to the other few instructions where "must" is used for a run-time check.

then traversing their ancestry to find their first common superclass. An immediate drawback of this approach is premature class loading. A more serious drawback is that type safety can be broken [CG01].

It turns out that it is possible to avoid this premature loading by assigning finite sets of names, instead of just names, to local variables and operand stack positions [Gol98, Qia99]. Merging of {`C`} and {`D`} yields {`C`, `D`}. This also prevents type safety from being broken [CG01].

The use of sets of names also provides a cleaner treatment of interfaces. Interface names are treated exactly like class names[8]. Since merging is set union, multiple inheritance of interfaces does not constitute a problem, and there is no need to use `java.lang.Object` as the result of merging. Therefore, no special treatment of `java.lang.Object` is necessary and hence no run-time checks must be performed when `invokeinterface` is executed.

3.1.3 Subtype Relation

Occasionally, during bytecode verification, a reference type `C` is the target of a method invocation instruction whose operand specifies `D` as the class or interface of membership for the method to be invoked. While the existence of the method is checked during resolution, bytecode verification must ensure that `C` is a subtype of `D`. JS4.9.1 prescribes that the two names are resolved in order to check that the desired subtype relation holds. But this results in premature loading.

A better approach is to generate a subtype constraint of the form $\mathtt{C} < \mathtt{D}$ [Gol98, QGC00, CG01]. Subtype constraints are checked if and when classes are loaded. This is analogous to the treatment of the loading constraints introduced in [LB98]. In fact, these two kinds of constraints can be integrated in the overall class loading mechanisms and formal arguments can be provided that type safety is guaranteed [QGC00].

3.1.4 Protected Fields and Methods

A subtle and often neglected point of bytecode verification is related to protected fields and methods. The specifications of the instructions to access fields and methods include the following requirement: if the referenced field or method is protected, and is declared either in the current class (i.e., the one whose code performs the field or method access) or in one of its superclasses, then the class of the object whose field or method is being accessed must be either the current class or a subclass of it.

This requirement derives from an analogous requirement stated in the specification of the Java programming language [GJSB00]. It has to do with the principle that protected fields or methods of an object can only be accessed from outside their package by code that is responsible to implement that object.

Anyhow, this requirement should be somehow checked by bytecode verification. Otherwise, additional run-time checks would be needed. This is not explicitly mentioned in JS4.8 or JS4.9, but it can be derived from JS6.

[8] In fact, whether a name denotes a class or an interface can be only determined by resolving the name. For instance, if a method has an argument of type `R`, that could denote a class as well as an interface.

A straightforward solution is to resolve the field or method, see if it is protected and declared in the current class or in a superclass, and in that case check that the class type assigned to the operand stack position in question is the current class or a subtype of it.

To avoid premature loading, the last check could be replaced by the generation of a subtype constraints. But the subtype constraint must be satisfied only if the field or method is protected and declared in the current class or a superclass. Otherwise, it does not need to hold, and it is actually wrong to require it to hold.

The solution is to generate a "conditional" subtype constraint, whose condition contains information about the field or method [Cog01b, Cog01c]. Conditional subtype constraints can be integrated with the unconditional ones and checked lazily, if and when classes are loaded and fields or methods are resolved.

3.2 Characterization

The summary of the discussion above is that the type safety properties that bytecode verification must guarantee can be derived from the specification of instructions given in JS6. In addition, bytecode verification can be made a purely functional component of the JVM. It is activated by passing a byte sequence as input, purported to be method code, accompanied by some context information such as the method's signature. The returned answer is either failure or success. In the latter case, some subtype constraints, conditional or unconditional, can also be generated. Bytecode verification never needs to load any class.

If the rectangular space of Figure 1 represents all possible code (i.e., all byte sequences), the larger, solid-line oval delimits the code that is *acceptable* by bytecode verification. This consists of all byte sequences satisfying the following two requirements: first, they must represent sequences of well-formed instructions; second, the instruction sequence must satisfy the type safety properties derived from JS6.

3.2.1 Sequences of Well-Formed Instructions

The first requirement essentially amounts to satisfaction of the static constraints given in JS4.8.1. Static constraints give a precise and computable characterization of which byte sequences represent sequences of valid instructions.

While they can be derived from the specification of instructions in JS6, it may be useful to collect them all in one place of JS—currently, JS4.8.1.

In my opinion, the only necessary or desired improvements are removing the constraint about `new` discussed in Section 2.1 and adding the two constraints about return instructions and execution not falling off the end of code, both discussed in Section 2.2.6. Also, based on the arguments in Section 2.2.1, the term "static constraints" could be replaced by something like "well-formedness constraints".

3.2.2 Type Safety Properties

The second requirement for acceptable code, i.e., the type safety properties that must be guaranteed at run time, can be stated precisely. They essentially depend only on the semantics of bytecode instructions. Unfortunately, it is undecidable whether an instruction sequence satisfies this characterization or not. So, bytecode verification must be a decidable approximation of this characterization, indicated by the mid-size, dashed-line oval in Figure 1: the oval delimits *accepted* code.

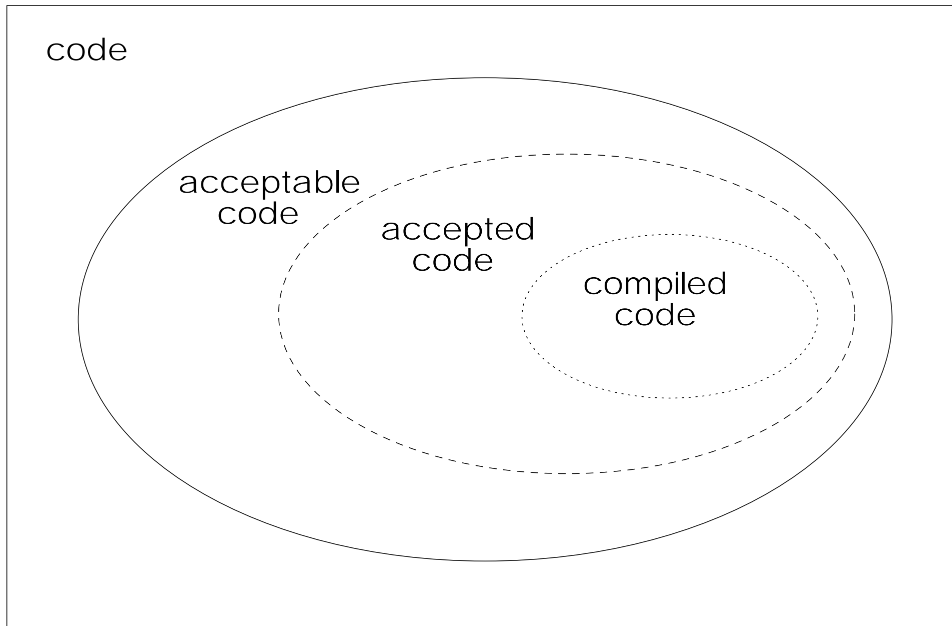

Figure 1: Characterizing bytecode verification

A desideratum is that any code produced by correct compilers is accepted by bytecode verification. It would be disappointing if, after successfully compiling a Java program, the JVM rejected the resulting class files. *Compiled* code (i.e., produced by compilers), is delimited by the smaller, dotted-line oval in Figure 1. So, the containment relationship among the ovals must be as indicated in the figure.

From a point of view, any choice of the accepted-code oval is fine, as long as it is contained in the acceptable-code oval and contains the compiled-code oval. Failure of either containment would cause type unsafety or rejection of compiled code.

While acceptable code can be characterized precisely based on the specification of instructions, compiled code depends on how compilers are implemented. Even if all current compilers used common compilation strategies, future compilers might use new strategies to generate better code. Therefore, the compiled-code oval is a moving target, and hence not very suitable to "universal" characterization.

The best approach is to give a precise characterization of accepted code. All implementors of the JVM should write bytecode verification algorithms that exactly recognize the specified subset. The specification of accepted code would also become a contract between developers of compilers and of the JVM: as long as a compiler produces code that falls inside the subset, that code is accepted by bytecode verification.

The definition of the accepted-code subset must embody an optimal trade-off between two criteria. The first criterion is that bytecode verification should be as simple and efficient as possible. The second criterion is that it should accept as much code as possible. Privileging the first could limit future compilers or reject code from current compilers; privileging the second could make implementations more susceptible to errors and hence to attacks exploiting the errors.

Based on the above considerations, the following are the necessary or desired improvements for JS, in my opinion. First of all, the notion of structural constraints should be purified of those forward references to the computable approximation about operand stack size (Section 2.3.1), uninitialized objects (Section 2.3.4), and subroutines (Section 2.3.5). Redundant constraints should also be eliminated. The contradiction regarding whether a constructor can store values into fields of an uninitialized object should be resolved, perhaps by disallowing that, because it simplifies verification [Cog01b, Cog01c]. Also, based on the arguments in Section 2.2.1, the term "structural constraints" could be replaced by something like "typing constraints".

In this way, these constraints would express all the type safety properties that acceptable code must satisfy. Note that these properties are "homogeneous" in some sense—see discussion in Section 2.2.7. While these can be derived from JS6, it may be useful to collect them all in one place of JS—currently, JS4.8.2.

More important, the algorithmic description in JS4.9 should be made much more complete and precise. It should be pointed out that the algorithm is an approximation of the undecidable constraints. The improved treatments of object initialization (Section 2.3.4), merging of reference types (Section 3.1.2), subtype relation (Section 3.1.3), and protected members (Section 3.1.4) should be incorporated. A solution to the problem, mentioned in Section 2.3.5, of compiled bytecode being rejected because of the treatment of subroutines, is given in [Cog01a]: this solution should be also incorporated. A formal description that fulfills all of these criteria is given in [Cog01b].

Finally, it should be made clear how bytecode verification cooperates with the other type safety mechanisms of the JVM. Issues include the use of class names in bytecode verification and the disambiguation of them to classes, as well as the generation of subtype constraints as opposed to loading classes.

4 Conclusion

The topic of Java bytecode verification has attracted the interest of several researchers. As a result, there has been a large number of publications on the subject [CG01, CGQ98, CL99, FC00, FC01, FM99a, FM99b, FM99c, Fre98, Gol98, HT98, Jon98, KN00, Nip01, O'C99, Pus99, PV98, Qia99, Qia00, Req00, RR98, SA99, SSB01, Yel99]. These works have greatly contributed to the clarification of key issues in bytecode verification, including pointing out some inadequacies in JS and proposing improvements.

To my knowledge, this paper is currently the only work to provide a comprehensive analysis of the official specification of bytecode verification and a comprehensive plan for improvement. Some details of the analysis and improvements (e.g., subroutines) are covered in other papers that are explicitly referred from this paper.

The need and desire to improve JS, in particular the description of class verification, "ideally to the point of constituting a formal specification", is explicitly stated in the Appendix of JS. This paper contributes to that goal.

References

[CG01] Alessandro Coglio and Allen Goldberg. Type safety in the JVM: Some problems in Java 2 SDK 1.2 and proposed solutions. *Concurrency—Practice and Experience*, 2001. To appear.

[CGQ98] Alessandro Coglio, Allen Goldberg, and Zhenyu Qian. Towards a provably-correct implementation of the JVM bytecode verifier. In *Proc. OOPSLA'98 Workshop on Formal Underpinnings of Java*, October 1998.

[CL99] Ludovic Casset and Jean Louis Lanet. A formal specification of the Java bytecode semantics using the B method. In *Proc. 1st ECOOP Workshop on Formal Techniques for Java Programs*, June 1999.

[Cog01a] Alessandro Coglio. Java bytecode subroutines demystified. Technical report, Kestrel Institute, 2001. Forthcoming at `http://www.kestrel.edu/java`.

[Cog01b] Alessandro Coglio. Java bytecode verification: A complete formalization. Technical report, Kestrel Institute, 2001. Forthcoming at `http://www.kestrel.edu/java`.

[Cog01c] Alessandro Coglio. Java bytecode verification: Another complete formalization. Technical report, Kestrel Institute, 2001. Forthcoming at `http://www.kestrel.edu/java`.

[DFW96] Drew Dean, Edward Felten, and Dan Wallach. Java security: From HotJava to Netscape and beyond. In *Proc. IEEE Symposium of Security and Privacy*, pages 190–200, May 1996.

[FC00] Philip Fong and Robert Cameron. Proof linking: Modular verification of mobile programs in the presence of lazy, dynamic linking. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 9(4):379–409, October 2000.

[FC01] Philip Fong and Robert Cameron. Proof linking: Distributed verification of Java classfiles in the presence of multiple classloaders. In *Proc. 1st Java Virtual Machine Research and Technology Symposium (JVM'01)*, pages 53–66. USENIX, 2001.

[FM99a] Stephen Freund and John Mitchell. A formal framework for the Java bytecode language and verifier. In *Proc. 14th ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'99)*, volume 34, number 10 of *ACM SIGPLAN Notices*, pages 147–166, October 1999.

[FM99b] Stephen Freund and John Mitchell. A type system for Java bytecode subroutines and exceptions. Technical Note STAN-CS-TN-99-91, Computer Science Department, Stanford University, August 1999.

[FM99c] Stephen Freund and John Mitchell. A type system for object initialization in the Java bytecode language. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 21(6):1196–1250, November 1999.

[Fre98] Stephen Freund. The costs and benefits of Java bytecode subroutines. In *Proc. OOPSLA'98 Workshop on Formal Underpinnings of Java*, October 1998.

[GJSB00] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java™ Language Specification*. Addison-Wesley, second edition, 2000.

[Gol98] Allen Goldberg. A specification of Java loading and bytecode verification. In *Proc. 5th ACM Conference on Computer and Communications Security (CCS'98)*, pages 49–58, November 1998.

[Gon99] Li Gong. *Inside Java™ 2 Platform Security*. Addison-Wesley, 1999.

[HT98] Masami Hagiya and Akihiko Tozawa. On a new method for dataflow analysis of Java Virtual Machine subroutines. In *Proc. 5th Static Analysis Symposium (SAS'98)*, volume 1503 of *Lecture Notes in Computer Science*, pages 17–32. Springer, September 1998.

[Jon98] Mark Jones. The functions of Java bytecode. In *Proc. OOPSLA'98 Workshop on Formal Underpinnings of Java*, October 1998.

[KN00] Gerwin Klein and Tobias Nipkow. Verified lightweight bytecode verification. In *Proc. 2nd ECOOP Workshop on Formal Techniques for Java Programs*, June 2000.

[LB98] Sheng Liang and Gilad Bracha. Dynamic class loading in the Java™ virtual machine. In *Proc. 13th ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'98)*, volume 33, number 10 of *ACM SIGPLAN Notices*, pages 36–44, October 1998.

[LY99] Tim Lindholm and Frank Yellin. *The Java™ Virtual Machine Specification*. Addison-Wesley, second edition, 1999.

[Nip01] Tobias Nipkow. Verified bytecode verifiers. In *Proc. 4th Conference on Foundations of Software Science and Computation Structures (FOSSACS'01)*, volume 2030 of *Lecture Notes in Computer Science*, pages 347–363. Springer, April 2001.

[NNH98] Flemming Nielson, Hanne Riis Nielson, and Chris Hankin. *Principles of Program Analysis*. Springer, 1998.

[O'C99] Robert O'Callahan. A simple, comprehensive type system for Java bytecode subroutines. In *Proc. 26th ACM Symposium on Principles of Programming Languages (POPL'99)*, pages 70–78, January 1999.

[Pus99] Cornelia Pusch. Proving the soundness of a Java bytecode verifier specification in Isabelle/HOL. In *Proc. 5th Conference on Tools and Algorithms for the Construction and Analysis of Systems TACAS'99*, volume 1579 of *Lecture Notes in Computer Science*, pages 89–103, March 1999.

[PV98] Joachim Posegga and Harald Vogt. Java bytecode verification using model checking. In *Proc. OOPSLA'98 Workshop on Formal Underpinnings of Java*, October 1998.

[QGC00] Zhenyu Qian, Allen Goldberg, and Alessandro Coglio. A formal specification of Java class loading. In *Proc. 15th ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'00)*, volume 35, number 10 of *ACM SIGPLAN Notices*, pages 325–336, October 2000. Long version available at `http://www.kestrel.edu/java`.

[Qia99] Zhenyu Qian. A formal specification of Java™ Virtual Machine instructions for objects, methods and subroutines. In Jim Alves-Foss, editor, *Formal Syntax and Semantics of Java™*, volume 1523 of *Lecture Notes in Computer Science*, pages 271–312. Springer, 1999.

[Qia00] Zhenyu Qian. Standard fixpoint iteration for Java bytecode verification. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 22(4):638–672, July 2000.

[Req00] Antoine Requet. A B model for ensuring soundness of a large subset of the Java Card virtual machine. In *Proc. 5th ERCIM Workshop on Formal Methods for Industrial Critical Systems (FMICS'00)*, pages 29–45, April 2000.

[RR98] Eva Rose and Kristoffer Rose. Lightweight bytecode verification. In *Proc. OOPSLA'98 Workshop on Formal Underpinnings of Java*, October 1998.

[SA99] Raymie Stata and Martín Abadi. A type system for Java bytecode subroutines. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 21(1):90–137, January 1999.

[Sar97] Vijay Saraswat. Java is not type-safe. Technical report, AT&T Research, 1997. Available at `http://www.research.att.com/vj/bug.html`.

[SSB01] Robert Stärk, Joachim Schmid, and Egon Börger. *Java™ and the Java™ Virtual Machine—Definition, Verification, Validation*. Springer, 2001.

[TH99] Akihiko Tozawa and Masami Hagiya. Careful analysis of type spoofing. In *Proc. Java-Informations-Tage 1999 (JIT'99)*, pages 290–296. Springer, September 1999.

[Yel99] Phillip Yelland. A compositional account of the Java Virtual Machine. In *Proc. 26th ACM Symposium on Principles of Programming Languages (POPL'99)*, pages 57–69, January 1999.