Formal Underpinnings of Java: Some Requirements

Richard M. Cohen
EDS Austin Technology Service Center
Austin, Texas USA
cohen@aus.edsr.eds.com

August 3, 1998

1 Why Formal Underpinnings for Java?

We would like to be able to execute a mixture of trusted and untrusted JVM classes with confidence that an untrusted class cannot take advantage of the authority/capabilities of a trusted class. This promise is one of the main reasons for the vast interest in Java.

We look to formal specifications and proofs to bolster our confidence and move us toward this goal.

Along the way toward gaining confidence, we might like to prove some security properties of a set of classes executing on a JVM. For example, we would like to know that private data of a class or instance cannot be altered or accessed by execution of any method outside the class. Type-safe execution may be a prerequisite of these properties, but is not sufficient. Type-safe dynamic linking may similarly be a prerequisite, but is not sufficient.

When attacking a large, complex, difficult, not-yet-understood problem, we often decompose it into a number of smaller, more tractable problems that (we hope) will provide steps to a combined solution to the larger problem. Many researchers have attacked subproblems of the "safe execution of untrusted classes on a JVM" problem. I want to make sure that we don't lose sight of the higher-level goal toward which these pieces build; and to make sure that we don't confuse primary safety properties of a system with secondary, or derived, properties of components. This is particularly easy to do in the world of Java, the JVM, and dynamic loading of class files, since the standard descriptions of these systems center around practical solutions used to make Java and the JVM appear to satisfy unstated safety properties similar to those above.

2 Some Necessary Pieces

A number of issues arise surrounding how to accomplish this overall goal. Consider the few key points below.

2.1 JVM vs. Java semantics

The Java language and the Java Virtual Machine are two different beasts. Clearly they have different levels of discourse, and some different fundamental operations. While the Java and JVM type systems are related, they are not the same.

We may implement Java programs on a JVM via a Java compiler, but the JVM program is clearly a translation of the Java program. The specification for the Java compiler is that it produce a JVM program that preserves the meaning of the Java program.

Since JVM programs are the medium of providing programs for execution, we should state their semantics – at the level of the JVM. While we would additionally like to show that a given JVM program preserves the meaning of a given Java program, that is a separate issue.

2.2 Type Safety in Java vs. Type Safety in the JVM

The type system of Java is not the same as that of the JVM. Clearly the statement of the types and consistency requirements for JVM program states is different from that for Java programs. The type system of the JVM must allow implementation of Java types consistent with the Java language semantics.

2.3 Static vs. Dynamic Typing Constraints

The static typing constraints defined for the JVM by Lindholm and Yellin are necessary because they describe an optimized JVM. [LY96] The Java language and the JVM have been cleverly designed to allow many run-time checks for type safety to be elided for programs satisfying certain static properties. Further, these static properties can be checked relatively efficiently. These two ideas are key to much of Java's success.

Optimized systems usually have more complex descriptions than their related unoptimized versions. This follows because the optimized systems rely on or impose some (possibly complex) system invariants. The standard JVM specification can elide certain run-time safety, but requires a complex description of legal JVM programs. This complexity is required because these optimizations are only safe under certain typing constraints.

It's more straightforward to justify the dynamic typing constraints of the JVM than the static typing constraints. The dynamic type constraints of an instruction are directly derived from the behavior of that instruction. For example, the dynamic type constraints on the `JSR` and `RET` instructions are quite simple, whereas the static type constraints are quite complex.

Not surprisingly, Drossopoulou and Eisenbach in their formalization of a subset of Java "discovered some rules [of the Java language description] to be more restrictive than necessary" [DE97].

The static typing constraints embodied in bytecode verification should be sufficient to assure safe execution, but are assuredly not necessary. We know that no bytecode verifier can accept all type-safe programs. We only want a bytecode verifier that will accept a large – and mostly predictable – class of type-safe programs.

Static typing constraints are defined only in service of efficient compilation and execution. The goal is type-safe execution.

The requirements for type-safe execution of JVM instructions are (relatively) straightforward, if somewhat more complex than might be expected. The requirement of a static typing system for JVM programs is that it assure type-safe execution, but the static typing system can be viewed as a secondary or derived property of the system. Indeed, the specification for the bytecode verifier is that it only accept programs whose dynamic behavior is type safe.

2.4 Data Integrity and Data Confidentiality

Our desire for safe execution of a JVM program serves another goal: that class and instance data can only be accessed or altered as "intended" (or, at least, according to the permissions explicit in the program).

For example, we might desire the following properties of JVM programs:

- data integrity – that private data of a class or instance cannot be altered except by a method of the class, and
- data confidentiality – that private data of a class of instance cannot directly affect the execution of a method of another class.

These informal statements clearly do not capture all we wish to say about data integrity and confidentiality, but they serve adequately as examples. These are properties of execution of a JVM program (or method) – not of a single instruction. Neither are they static properties of a program. They are derived from static properties of the JVM.

2.5 Implementation of Security Policies

A security policy states the circumstances under which certain actions may be taken or certain resources accessed. Current JVM implementations use a mechanism called "stack inspection" to enforce their security policy. Some particular policies enforcible by stack inspection have been formalized by Wallach and Felten. [WF98] A particular policy can be stated as a property of program execution, and should be amenable to formal proof given a formal model of JVM execution.

2.6 Keeping Formal Definitions Consistent

The Java language is fairly small by programming language standards. Yet the English description takes approximately 450 pages [GJS96]. A description one-tenth the size could capture the spirit, but not all of the details. Surely any formal definition of similar coverage and completeness would be at least as large. It is extremely difficult to keep such a large specification internally consistent. A mechanical tool for checking the consistency of the formal specification would be most useful. A mechanical tool for checking proofs using the formal specification would also be useful.

3 Overview of a Solid Foundation

Our goal of trusting execution of untrusted programs on a JVM requires solutions to a number of problems. Here is such a set of problems (motivated by the discussion above) whose solution would form a solid foundation for Java and the JVM. This is not the only decomposition nor the only plausible order. But a comprehensive approach must cover these points.

1. Define the behavior JVM execution.
2. Define safe execution on the JVM.
3. Define necessary and sufficient conditions for safe execution.
4. Define a recognizer for (a large set of) programs that will execute safely.
5. Prove that recognized programs execute safely.
6. Define an optimized JVM on which the recognized programs behave just as they would on the original JVM (except perhaps faster).
7. Define data integrity and confidentiality properties of the JVM.
8. Prove them.
9. Define a security-policy enforcement mechanism (e.g., privileged classes with stack inspection or "security passing").
10. Prove the mechanism can be implemented securely on the JVM. (There is also the separate question of what policies can be enforced with the chosen mechanism.)
11. Define the Java language.
12. Define a Java to JVML compiler.
13. Prove that the compiler produces a JVM program that preserves the behavior of the Java program.

The encouraging part is that we can envision doing most or all of this for Java and the JVM. But much remains before our formalisms can provide overall confidence about execution of untrusted programs on a JVM.

References

[DE97] Sophia Drossopoulou and Susan Eisenbach.
Java is Type Safe – Probably.
In *11th European Conference on Object Oriented Programming*, 1997.

[GJS96] James Gosling, Bill Joy, and Guy Steele.
The Java Language Specification.
Addison-Wesley, 1996.

[LY96] Tim Lindholm and Frank Yellin.
The Java Virtual Machine Specification.
Addison-Wesley, 1996.

[WF98] Dan S. Wallach and Edward W. Felten.
Understanding Java Stack Inspection.
In *IEEE Symposium on Security and Privacy*, May 1998.