Security and Dynamic Class Loading in Java: A Formalisation*

T. Jensen D. Le Métayer T. Thorn
IRISA/CNRS/INRIA†
Campus de Beaulieu
F-35042 Rennes Cedex, France

Email: {jensen,lemetayer,thorn}@irisa.fr

Abstract

We give a formal specification of the dynamic loading of classes in the Java Virtual Machine (JVM) and of the visibility of members of the loaded classes. This specification is obtained by identifying the part of the run-time state of the JVM that is relevant for dynamic loading and visibility and consists of a set of inference rules defining abstract operations for loading, linking and verification of classes. The formalisation of visibility includes an axiomatisation of the rules for membership of a class under inheritance, and of accessibility of a member in the presence of accessibility modifiers such as `private` *and* `protected`*. The contribution of the formalisation is twofold. First, it provides a clear and concise description of the loading process and the rules for member visibility compared to the informal definitions of the Java language and the JVM. Second, it is sufficiently simple to allow calculations of the effects of load operations in the JVM.*

1 Introduction

Security is one of the most important issues concerning the Java language, at least in the context of its application to the programming of mobile code. Java involves a unique combination of factors which makes the study of security issues especially challenging:

- The philosophy of the designers of the Java environment is that security mechanisms are language-based. Developers are encouraged to write code exclusively in Java, which implies that all security controls can be achieved through the Java abstract machine.
- Java was explicitly designed to be a small language including only well-understood features. Java is supposed to be a "type safe" language. This claim is still a matter of debate but even if this goal is not quite achieved yet, it is reasonable to believe that the fundamental choices underlying the language are sufficiently sound to make it possible to fix the remaining problems in the near future.

However, as shown by the various discussions triggered by the claims about the security of Java, the language still involves a number of innovations and subtleties which make its semantics far from obvious. The most striking features of the language in this respect are tied to the dynamic loading of classes:

- Verification of type soundness is carried out at four different times in Java: at compile time, at load time, at link time and at run time. The first one applies to source code relatively to a static environment, while the other ones apply to class files, the concrete representation of a compiled class definition, relatively to a set of dynamically loaded classes.
- The notion of class loader plays a critical role in the security of Java. Each class is associated with a specific class loader which corresponds to a specific name space in the virtual machine. The Java security model relies on name spaces to ensure that an untrusted applet cannot interfere with other Java programs [7].

Another important aspect of the security of the language is the visibility modifiers (*default, public, protected, private*) which are attached to classes and members in order to restrict the access to these. For instance, attributes with a *default* visibility are fully accessible only within their definition package.

This combination of features inevitably leads to complexities and uncertainties which can have unfortunate consequences in terms of security. This is manifested by the

* Published in the Proceedings of the 1998 IEEE International Conference on Computer Languages, Chicago, Illinois, May 14-16, 1998, p.4-15. Revised: Mars 12, 1999.

† Part of this work has been done in the context of Dyade (R&D joint venture between Bull and Inria).

fact that apparently harmless programs can lead to unexpected security flaws [13]. Furthermore some features of the informal specification of the language and the virtual machine are subject to different interpretations.

This situation is clearly not acceptable for a language in which security is such an important issue and we believe that the only way to tackle these problems is through an appropriate formalisation. But a formalisation is useful only if it really helps reasoning about security properties. It should be defined at the right level of abstraction, which means precise enough to include all the security critical features of the language (e.g. visibility rules, dynamic loading and class loaders) and abstract enough to focus on the significant issues and avoid all the irrelevant details (w.r.t. security) of the complete semantics.

Several formalisations of Java have been proposed in the literature. Each one has its own merits and sheds some light on the language, but we believe that none of them really satisfies the above criteria. At the language level, Drossopoulou and Eisenbach [4] have proposed an operational semantics and a type system for core Java, and proved that the type system is sound with respect to this semantics. However, this description does not deal with dynamic class loading at all. At the virtual machine level, the *tour de force* formalisation of the Java virtual machine by Bertelsen [1] seems too detailed to be used for practical calculations.

In this paper, we propose a formalisation of the visibility rules in the hierarchy of classes loaded into a virtual machine and their evolution through dynamic loading and linking. The formalisation is based on the informal specification of the visibility rules in the Java language definition [8]. We relate each of our rules to the explanations taken from [8], explaining our interpretation when appropriate. The formalisation of dynamic class loading is based on [9]. The two main contributions of the formalisation presented in this paper are the following:

- It provides a clear and concise description of the loading process and the rules for member visibility compared to the definitions of the Java language and the Java virtual machine (JVM). The informal definition of these rules and their interaction are quite complex, and the first benefit of the formalisation is a better understanding of those issues.

- It is sufficiently simple to allow calculations of the effects of load operations in the JVM. Indeed, we believe this specification to be at the right level of abstraction to help reasoning about security properties in Java because it is precise enough to include the security-critical features of the language (such as visibility rules, dynamic loading and class loaders) and abstract enough to avoid all the irrelevant details (w.r.t. security) of the complete semantics.

We abstract the state of the virtual machine into a set of relations describing the hierarchy of classes and members with their visibility rules. We do not consider computations on basic values here since they do not have any impact on the abstract state (and the security issues). As a consequence, we do not describe type checking either, considering it a complementary issue which can be tackled *e.g.*, by using the system presented by Qian in [12]. Then we show that this formalisation allows us to expose in a straightforward way a security problem that was discovered empirically in [13].

It is assumed that the reader is familiar with Java and its terminology concerning classes, interfaces, and visibility modifiers. The next section summarises our notation for these entities.

2 Syntax

Below we formalise the concepts from Java programs relevant to our point of view. These are names (of packages, classes, interfaces, fields, and methods), relations (subclassing and implements), types, and access modifiers. All these concepts together lead to the definition of a *class file*. The class file is the representation of a compiled Java class that can be loaded into the JVM.

2.1 Package, class, and interface names

Each class belongs to a package which defines the limits of visibility of the protected members.

$$\mathit{PackageName} ::= \mathit{Identifier} \mid \mathit{PackageName}\,'.'\,\mathit{Identifier}$$

Different classes and interfaces may appear with the same name in different packages. To disambiguate them, we use their *fully qualified names*, *i.e.*, their name paired with the package of their definition. The set *Class* is the set of valid class and interface names.

$$\mathit{Class} ::= \mathit{PackageName}\,'.'\,\mathit{Identifier}$$

Classes and interfaces declare a number of fields and methods. The namespaces of the two are disjoint. The fields of a class are class variables, instance variables, and constants, while only constants are allowed as interface fields. For our purpose, we do not need to distinguish between the different kinds of fields and just represent them all with *Field*, the set of field names paired with the class of their definition. As Java allows overloaded methods, the type of the arguments is needed to distinguish between different methods with the same name, thus *Method* is the set of methods (a pair of method name and argument signatures) paired with the class (or interface) of their definition.

We refer to the disjoint union[1] of these sets as *Member*. The set *Type* of argument types is defined in Section 2.3.

$$\begin{aligned} \mathit{FieldDesc} &= \mathit{Identifier} \\ \mathit{MethodDesc} &= \mathit{Identifier} \times \mathit{Type}^* \\ \mathit{MemberDesc} &= \mathit{FieldDesc} \uplus \mathit{MethodDesc} \\ \mathit{Field} &= \mathit{Class} \times \mathit{FieldDesc} \\ \mathit{Method} &= \mathit{Class} \times \mathit{MethodDesc} \\ \mathit{Member} &= \mathit{Field} \uplus \mathit{Method} = \mathit{Class} \times \mathit{MemberDesc} \end{aligned}$$

To ease the manipulation of the names from *Class* and *Member*, we define three selectors *package*, *origin*, and *descriptor*. The selector *package* gives the package name part of a class or interface name, *origin* the class or interface in which the member is declared, and *descriptor* the descriptor part of a member name:

$$\begin{aligned} \mathit{package} &: \mathit{Class} \to \mathit{PackageName} \\ \mathit{origin} &: \mathit{Member} \to \mathit{Class} \\ \mathit{descriptor} &: \mathit{Member} \to \mathit{MemberDesc} \end{aligned}$$

Since constructors play no particular rôle in this paper, we follow the operational reality and model the constructor of a class as a special method `<init>`, distinct from all user definable methods. The correspondence is simple:

```
new T(...)       ⟶  T.<init>(...)
new super(...)   ⟶  super.<init>(...)
this(...)        ⟶  this.<init>(...)
```

2.2 Access modifiers

Classes, interfaces, and class members can have their visibility changed through the use of a modifier from *Modifier*.[2]

$$\mathit{Modifier} = \{\texttt{public}, \texttt{protected}, \texttt{default}, \texttt{private}\}$$

The modifier only gives a *local* constraint. The overall access is determined by the combination of the restrictions on the package, the class, and the member itself. For classes and interfaces, only `public` and `default` apply. Assuming no other restrictions apply, `public` classes can be accessed everywhere, while the `default` visibility only grants access to classes from its package.

For members, `public` denotes that access to the entity is unconstrained. A `private` member is only accessible within its class of definition. The `default` accessibility of a member is restricted to classes within the same package as itself. Access of a `protected` member is slightly more complicated: access is generally granted in subclasses and in classes from the same package as the member. For interface members the visibility is always public. The precise semantics of modifiers in defined in Sections 4–5.

2.3 Types

The set of basic types such as `void`, `int` and `boolean` are represented by the set *BaseType*. Together all classes, interfaces, and arrays are known as reference types and *RefType* is defined as the smallest set satisfying

$$\begin{gathered} \mathit{Class} \subset \mathit{RefType} \\ \forall t \in \mathit{RefType} \cup \mathit{BaseType} \,.\, t[] \in \mathit{RefType} \end{gathered}$$

where an array with elements of type t is written $t[]$. The set of expressible types is the union of base types and reference types:

$$\mathit{Type} = \mathit{BaseType} \cup \mathit{RefType}$$

2.4 Class files

The class file is a concrete representation of a compiled Java class and is the unit for linking. We model the class file with an abstract type, *ClassFile* with corresponding operations:

$$\begin{aligned} \mathit{name} &: \mathit{ClassFile} \to \mathit{Class} \\ \mathit{super} &: \mathit{ClassFile} \hookrightarrow \mathit{Class} \\ \mathit{implement} &: \mathit{ClassFile} \to \mathcal{P}(\mathit{Class}) \\ \mathit{member} &: \mathit{ClassFile} \to \mathcal{P}(\mathit{Member}) \\ \mathit{member\text{-}type} &: \mathit{ClassFile} \times \mathit{Member} \hookrightarrow \mathit{Type} \\ \mathit{member\text{-}modifier} &: \mathit{ClassFile} \times \mathit{Member} \hookrightarrow \mathit{Modifier} \\ \mathit{class\text{-}modifier} &: \mathit{ClassFile} \to \mathit{Modifier} \\ \mathit{references} &: \mathit{ClassFile} \to \mathcal{P}(\mathit{Class}) \end{aligned}$$

where

- *name* gives the fully qualified name of the class implemented by the class file,
- *super* gives the name of the super class,
- *implement* gives the set (possibly empty) of implemented interfaces,
- *member* gives the set of members declared for the class,
- *member-type* gives the type (or return type for methods) of members,
- *member-modifier* gives the declared modifier of members, and

[1] We use disjoint union in order to preserve the information whether a member is a field or a method.

[2] There is no `default` keyword in Java, but the absense of a modifier implies it. To simplify, we assume that it is explicitly given.

- *class-modifier* gives the declared access modifier for the class itself.
- *references* gives the set of class names referenced from the class file, including those used by methods.

The functions *member-type* and *member-modifier* are only defined on members from the class file and are thus partial functions, denoted by $\hookrightarrow$. Similarly, *super* is a partial function since the class `java.lang.Object` does not have a super class. It follows from the definition that for a given class file *cf*,

$$\forall m \in \mathit{member}(cf) \ . \ \mathit{origin}(m) = \mathit{name}(cf)$$

3 Dynamic loading

Classes and interfaces are loaded into the JVM dynamically when they are either needed or explicitly demanded. The loading of classes is done by the methods `loadClass` of the `ClassLoader` classes. These methods accept as arguments a fully qualified name and determine where to search for a class file containing a class of that name. The return value is an object of class `Class`. There are two noticeable aspects about the relationship between class loaders and classes:

- To every class is associated the class loader object that loaded the class. It is this class loader that is invoked when the code in the class necessitates another class to be loaded.
- Once a class loader has loaded a class with name *n* and returned `Class` object *o*, all subsequent calls to that class loader with *n* as argument will return the same object *o*.[3]

The first aspect means that a loaded class is identified by specifying its class file and the class loader that loaded it. We define *ClassLoader* as an abstract set of class loaders with the system class loader, *scl*, as a distingushed element. The set of loaded classes is the set product of *ClassLoader* and *ClassFile*:

$$\mathit{LoadedClass} = \mathit{ClassLoader} \times \mathit{ClassFile}.$$

We use the notation $c.cf$ and $c.cl$ to denote the class file and class loader of a loaded class c.

The user is free to program his own class loader whose behaviour can be completely different from that of the system class loader, as long as it respects the second requirement above. The behaviour of a class loader can thus be modelled by a function

$$\mathcal{W} : \mathit{LoadedClass} \times \mathit{Class} \hookrightarrow \mathit{LoadedClass}$$

that for a given class loader *cl* (itself a loaded class) and a class name n returns a loaded class c_n. The function $\mathcal{W}$ satisfies:

$$\forall cl \in \mathit{LoadedClass}, n \in \mathit{Class} \ .$$
$$\mathcal{W}(cl, n) = c \ \Rightarrow \ \mathit{name}(c.cf) = n$$

Notice that the class loader attached to $\mathcal{W}(cl, n)$ may be different from *cl*, since *cl* may choose to delegate the loading to another class loader (*cf.* the example in Section 7).

The loading of a class c recursively triggers the loading of the superclass of c; thus after loading *c* with class loader *cl*, all classes between *c* and the root of the class tree have been loaded, but not necessarily by *cl*. If *cl* delegates the loading of certain classes to another class loader *cl'* then some of the superclasses could be loaded by *cl'*. The loading process will be formalised in Section 6.2.

3.1 Subclass and implements relation

A loaded class is an immediate subclass of another loaded class iff they are loaded by the same class loader and the declared super class of the former matches the name of the latter. We define the following relations relative to a set S of loaded classes since this will later be our representation of the state of the virtual machine (Section 6).

$$\text{sub} \ \subseteq \mathit{LoadedClass} \times \mathit{LoadedClass}$$

$$\frac{c, c' \in S \quad c' = \mathcal{W}(c.cl, \mathit{super}(c.cf))}{S \vdash c \ \text{sub} \ c'} \tag{1}$$

The subclass relation sub^* is the reflexive, transitive closure of sub .

$$\frac{}{S \vdash c \ \text{sub}^* \ c} \tag{2}$$

$$\frac{c, c', c'' \in S \quad S \vdash c \ \text{sub} \ c'' \quad S \vdash c'' \ \text{sub}^* \ c'}{S \vdash c \ \text{sub}^* \ c'} \tag{3}$$

Likewise we lift the notion that a class implements an interface to a relation between two instances of *LoadedClass* (where the latter represents an interface):

$$\text{impl} \ \subseteq \mathit{LoadedClass} \times \mathit{LoadedClass}$$

$$\frac{c, i \in S \quad \mathit{name}(i.cf) \in \mathit{implement}(c.cf) \quad i = \mathcal{W}(c.cl, \mathit{name}(i.cf))}{S \vdash c \ \text{impl} \ i} \tag{4}$$

[3] Although 2.16.2 in the JVM Specification [9] seems to indicate something else. The newer 1.1 implementation follows this rule, but we haven't been able to locate any documentation confirming this.

4 Membership and inheritance

The concepts of membership and inheritance are not explicit in the class files, but can be inferred from the syntactic information outlined in Section 2. Below we formalize such an inference system for the membership and inheritance based on their informal description in the Java specification[4]. In order to integrate this with the semantics of dynamic loading (Section 6) every rule is relative to a set of loaded classes S and a current class $cc \in$ *LoadedClass*.

4.1 Members

Throughout this section, §-references refer to [8]. §6.4.2 states the conditions for class membership (which are repeated in §8.2):

- Members inherited from its direct superclass (§8.1.3)
- Members inherited from any direct superinterfaces (§8.1.4)
- Members declared in the body of the class (§8.1.5)

§6.4.3 states the conditions for interfaces membership:

- Members inherited from any direct superinterfaces (§9.1.3)
- Members declared in the body of the interface (§9.1.4)

In the following, we combine these rules. We use m to denote a member from *Member* and c to denote a loaded class, represented by a *LoadedClass* and define 'member-of' as a relation between m and c.

$$\text{member-of} \subseteq \textit{Member} \times \textit{LoadedClass}$$

$$\frac{m \in \textit{member}(c.\textit{cf})}{S, cc \vdash m \text{ member-of } c} \tag{5}$$

$$\frac{S, cc \vdash c \text{ inherits } m}{S, cc \vdash m \text{ member-of } c} \tag{6}$$

The 'inherits' relation is defined in the next section.

4.2 Inheritance

Following [8] the rules for inheritance are divided into two parts. The first part gives an over-estimation, which is then constrained by the second. A general, but not sufficient, condition for inheritance is given by §8.2:

> "Members of a class that are declared private are not inherited by subclasses of that class. Only members of a class that are declared protected or public are inherited by subclasses declared in a package other than the one in which the class is declared."

[4]The definition is dependent on the concept of package access which in itself is system-dependent (file access etc). We do not model this aspect and assume that an occurence of a loaded class in one of the rules below implies that the class loader could obtain access to the package of the class.

We formulate this with the 'inheritable-by' relation between a member m and a loaded class c. The class name $\textit{origin}(m)$ associated with m is first translated into a loaded class c_m, and then the modifier associated with m in the class file of c_m is determined:

$$\text{inheritable-by} \subseteq \textit{Member} \times \textit{LoadedClass}$$

$$\frac{\begin{array}{l} c_m = \mathcal{W}(c.\textit{cl}, \textit{origin}(m)) \\ \textit{member-modifier}(c_m.\textit{cf}, m) \in \{\texttt{protected}, \texttt{public}\} \\ S \vdash c \text{ sub}^* c_m \end{array}}{S, cc \vdash m \text{ inheritable-by } c} \tag{7}$$

$$\frac{\begin{array}{l} c_m = \mathcal{W}(c.\textit{cl}, \textit{origin}(m)) \\ \textit{member-modifier}(c_m.\textit{cf}, m) = \texttt{default} \\ S \vdash c \text{ sub}^* c_m \\ \textit{package}(\textit{name}(c.\textit{cf})) = \textit{package}(\textit{origin}(m)) \end{array}}{S, cc \vdash m \text{ inheritable-by } c} \tag{8}$$

Other inheritance rules for members appear in two places in the specification. §8.3 deals with class fields and states:

> "A class inherits from its direct superclass all the fields of the superclass that are both accessible to code in the class and not hidden by a declaration in the class."

A field is *hidden* by declaring another field with the same name in a subclass. §8.4.6 deals with class methods and states:

> "A class inherits from its direct superclass all the methods of the superclass that are accessible to code in the class and are neither overridden nor hidden by a declaration in the class."

A class (resp., instance) method is said to be *hidden* (resp., *overridden*) if a subclass declares a class (resp., instance) method with the same name and signature *and* the former is visible in that subclass. Again, analogous inheritance rules are stated for interfaces in §9.2. We express the constraint imposed by "neither overridden nor hidden" with the relation 'undeclared-in' $\subseteq$ *Member* $\times$ *LoadedClass*.

$$\frac{\forall m' \in \textit{member}(c.\textit{cf}) \,.\, \textit{descriptor}(m) \neq \textit{descriptor}(m')}{S, cc \vdash m \text{ undeclared-in } c} \tag{9}$$

The relations 'inherits' between a member m and a loaded class c is defined based on the conditions above. Notice that the notion of accessibility here is the one given by 'inheritable-by' and not the one that follows in Section 5.1.

$$\text{inherits} \subseteq \textit{LoadedClass} \times \textit{Member}$$

$$\frac{S \vdash c \text{ sub } c' \quad S, cc \vdash m \text{ member-of } c' \quad S, cc \vdash m \text{ inheritable-by } c \quad S, cc \vdash m \text{ undeclared-in } c}{S, cc \vdash c \text{ inherits } m} \tag{10}$$

$$\frac{S \vdash c \text{ impl } c' \quad S, cc \vdash m \text{ member-of } c' \quad S, cc \vdash m \text{ undeclared-in } c}{S, cc \vdash c \text{ inherits } m} \tag{11}$$

Interface members are always public, so we leave out the 'inheritable-by' condition for interfaces.

5 Accessibility

Only member modifiers have been used to define the 'member-of' and 'inherits' relations. We show in this section how class and interface modifiers influence the accessibility of a member of one class from another class. This is formalised by the relation 'accessible' defined below.

$$\text{accessible} \subseteq \textit{LoadedClass} \times \textit{Member}$$

5.1 Class and interface accessibility

These rules determine when a class (or interface) c is accessible by methods within the current class *cc*. §6.6.1 states:

> "If a class or interface type is declared public, then it may be accessed by any Java code that can access the package in which it is declared. If a class or interface type is not declared public, then it may be accessed only from within the package in which it is declared."

The accessibility of packages is dependent on the class loader. As for the rules for membership (see footnote at the beginning of Section 4), the rules suppose that all packages needed could be accessed.

$$\text{type-acc} \subseteq \textit{LoadedClass}$$

$$\frac{\textit{class-modifier}(c.cf) = \texttt{public}}{S, cc \vdash c \text{ type-acc}} \tag{12}$$

$$\frac{\textit{class-modifier}(c.cf) = \texttt{default} \quad \textit{package}(\textit{name}(c.cf)) = \textit{package}(\textit{name}(cc.cf))}{S, cc \vdash c \text{ type-acc}} \tag{13}$$

5.2 Member accessibility

The 'accessible' relation determines the accessibility of a member m of a class c from the current class. The member m

- is defined in the class named *origin*(m), corresponding to the loaded class $\mathcal{W}(cc.cl, \textit{origin}(m))$ under the current class *cc*,
- belongs to an object o, which is an instance of class c, and
- is invoked in an instruction appearing in a method of the current class.

The specification of the Java language [8, §6.6.1] states:

> "A member (field or method) of a reference type or a constructor is accessible only if the type is accessible and the member or constructor is declared to permit access."

In particular, since classes and interfaces have their own accessibility modifiers, it is necessary to check whether the modifier allows access from the current class. However, it seems that this check is not performed in some implementations of the virtual machine; thus, if we want to model such machine behaviour, the check "c type-acc" should be left out of 'accessible' defined below.

$$\text{accessible} \subset \textit{LoadedClass} \times \textit{Member}$$

$$\frac{S, cc \vdash m \text{ member-of } c \quad S, cc \vdash c \text{ type-acc} \quad S, cc \vdash (c, m) \text{ permit-acc}}{S, cc \vdash (c, m) \text{ accessible}} \tag{14}$$

The modifier of member m is found in the class file $\mathcal{W}(cc.cl, \textit{origin}(m)).cf$. The rule 'permit-acc' details the extra conditions determined by the declared access modification. The case for protected members is treated separately below.

$$\text{permit-acc} \subseteq \textit{LoadedClass} \times \textit{Member}$$

$$\frac{\textit{member-modifier}(\mathcal{W}(cc.cl, \textit{origin}(m)).cf, m) = \texttt{public}}{S, cc \vdash (c, m) \text{ permit-acc}} \tag{15}$$

$$\frac{\textit{member-modifier}(\mathcal{W}(cc.cl, \textit{origin}(m)).cf, m) = \texttt{protected} \quad \textit{package}(\textit{name}(cc.cf)) = \textit{package}(\textit{origin}(m))}{S, cc \vdash (c, m) \text{ permit-acc}} \tag{16}$$

$$\frac{\textit{member-modifier}(\mathcal{W}(cc.cl, \textit{origin}(m)).cf, m) = \texttt{protected} \quad S \vdash cc \text{ sub}^* \mathcal{W}(cc.cl, \textit{origin}(m)) \quad S, cc \vdash (c, m) \text{ prot-acc}}{S, cc \vdash (c, m) \text{ permit-acc}} \tag{17}$$

$$\frac{\textit{member-modifier}(\mathcal{W}(cc.cl, \textit{origin}(m)).cf, m) = \texttt{default} \quad \textit{package}(\textit{name}(cc.cf)) = \textit{package}(\textit{origin}(m))}{S, cc \vdash (c, m) \text{ permit-acc}} \tag{18}$$

$$\frac{\textit{member-modifier}(\mathcal{W}(cc.cl, \textit{origin}(m)).cf, m) = \texttt{private} \quad \mathcal{W}(cc.cl, \textit{origin}(m)) = cc}{S, cc \vdash (c, m) \text{ permit-acc}} \tag{19}$$

For protected members, §6.6.2 specifies:

> "A protected member of an object may be accessed from outside the package in which it is declared only by code that is responsible for the implementation of that object."

$$\frac{\textit{descriptor}(m) \neq \texttt{<init>} \quad S \vdash c \text{ sub}^* \ cc}{S, cc \vdash (c, m) \text{ prot-acc}} \tag{20}$$

For protected access between different packages we thus have the requirement that the three classes involved form the hierarchy

$$c \text{ sub}^* \ cc \text{ sub}^* \ \mathcal{W}(cc.cl, \textit{origin}(m)).$$

6 The Java virtual machine

6.1 The JVM state

The part of the state of a JVM that is relevant for the operations below is given by the *current class cc* that caused their invocation and the set of classes loaded into the machine.

$$S \in \textit{State} = \mathcal{P}(\textit{LoadedClass})$$

The state determines a set of semantic relations on inheritance, accessibility and visibility of names in classes, as explained in the preceding sections. The purpose of this section is to describe how the instructions of the virtual machine [9] transform this state. This transformation is itself dependent on the visibility relation determined by the current state of the machine. In the following, we formalise two abstract operations *load* and *link* that model the behaviour of the JVM when loading and linking classes. The operations *load* or *link* receive as arguments the class *n* to be loaded or linked. We define a relation of the form $S, cc \vdash op(n) \rhd S'$, meaning that execution of operation *op* in state S leads to state S'.[5]

[5]The current class, *cc*, is unchanged after execution. It is straightforward to turn this into a transition relation of form *op(n):(S,cc)* $\rightarrow$ *(S',cc)*, but for expository reasons we have chosen the other format.

6.2 Loading

Some JVM instructions refer to names of classes, interfaces, methods or fields. Their execution provokes the loading of classes containing the definition of these names if these have not already been loaded. For example, executing the JVM instruction for calling a method requires the loading of the class in which the method is defined. Furthermore, loading of a class always entails the loading of its superclass; by recursion, the net result is that all classes between the class initially to be loaded and the top of the class hierarchy `java.lang.Object` are loaded. This loading is performed by the class loader associated with the current class (cf. Section 3). The following rule formalises the loading process.

$$\frac{\mathcal{W}(cc.cl, \texttt{java.lang.Object}) = c_n}{S, cc \vdash \textit{load}(\texttt{java.lang.Object}) \rhd S \cup \{c_n\}} \tag{21}$$

$$\frac{\mathcal{W}(cc.cl, n) = c_n \quad \textit{super}(c_n.cf) = n_s \quad S, cc \vdash \textit{load}(n_s) \rhd S'}{S, cc \vdash \textit{load}(n) \rhd S' \cup \{c_n\}} \tag{22}$$

6.3 Linking

Since some classes are only needed for verification purposes, the loading of a class does not automatically make its methods executable to the JVM ([9, §5.1.2.]). In order to invoke the methods of a class, the loaded class must be *linked*. The linking phase involves byte-code verification which in turn provokes the loading (but not linking) of classes containing the definition of entities named in the byte-code. We represent this verification process by an operation *verify* on a class. Furthermore, linking a class into the JVM automatically causes its superclass to be linked in.

$$\frac{S, cc \vdash \textit{load}(\texttt{java.lang.Object}) \rhd S' \quad \mathcal{W}(cc.cl, \texttt{java.lang.Object}) = c_n \quad S', cc \vdash \textit{verify}(c_n) \rhd S''}{S, cc \vdash \textit{link}(\texttt{java.lang.Object}) \rhd S''} \tag{23}$$

$$\frac{S, cc \vdash \textit{load}(n) \rhd S' \quad c_n = \mathcal{W}(cc.cl, n) \quad \textit{super}(c_n.cf) = n_s \quad S', cc \vdash \textit{link}(n_s) \rhd S'' \quad S'', cc \vdash \textit{verify}(c_n) \rhd S'''}{S, cc \vdash \textit{link}(n) \rhd S'''} \tag{24}$$

It should be noted that the operations can fail. The inference rules here and in the following only specify the treatment of errors implicitly; an error implies that no deduction is possible for a given operation.

The verification phase loads all classes referenced in the code being linked using the class being verified as the current class. It then type-checks the code based on the visibility relations determined by the loaded classes.

$$\frac{\begin{array}{l} C = \mathit{references}(c.\mathit{cf}) \\ S, c \vdash \mathit{load}^*(C) \rhd S' \\ S', \mathit{cc} \vdash \mathit{typecheck\text{-}class}(c) \end{array}}{S, \mathit{cc} \vdash \mathit{verify}(c) \rhd S'} \quad (25)$$

where *load** is *load* lifted to sets of classes.

The function *typecheck-class* verifies each instruction of a class using *typecheck* defined below. Type checking of JVM instruction is described in detail by Qian [12] and is not the aim of this paper. We give a brief presentation of the rules for type-checking two instructions that models method invocation and assignment (called *invokevirtual* and *putfield* after their JVM equivalents) in order to demonstrate how visibility and loading interacts.

Type-checking of JVM instructions in [12] operates with a simulated stack *ST* of types mirroring the types of the objects on the run-time stack. Invoking a method m defined in a class c_m from the object on top of the run-time stack, requires that m is visible from the class c_0 of that object:

$$\frac{\begin{array}{l} \mathit{ST} = c_0 :: \mathit{ST'} \\ S, \mathit{cc} \vdash (c_0, m) \text{ accessible}\mathit{cc} \end{array}}{S, \mathit{cc}, \mathit{ST} \vdash \mathit{typecheck}(\mathit{invokevirtual}(m))} \quad (26)$$

Similarly, the instruction *putfield*(cc, f) stores the object on top of the stack in the f field of the object just below the stack top, thus the type of f must be a superclass of the class on top of the simulated stack:

$$\frac{\begin{array}{l} \mathit{ST} = c_a :: c_m :: \mathit{ST'} \\ f \text{ member-of } c_m \\ f_c = \mathit{member\text{-}type}(\mathcal{W}(\mathit{cc.cl}, \mathit{origin}(f)).\mathit{cf}, f) \\ f_c \in \mathit{Class} \\ c_a \ \text{sub}^* \ \mathcal{W}(\mathit{cc.cl}, f_c) \end{array}}{S, \mathit{cc}, \mathit{ST} \vdash \mathit{typecheck}(\mathit{putfield}(f))} \quad (27)$$

6.4 Semantics of JVM instructions

JVM machine instructions reference classes and their fields and methods via the *constant pool* of the current class [9, Ch. 5]. Each class defines a constant pool whose entries contain attributes of the entity referenced through that entry. This indirect naming avoids the need for fully qualified names to appear in the bytecode but is irrelevant for our purposes. We shall instead work with an instruction set where instruction carry the information from the constant pool given as fully qualified names.

Instruction	Argument type
`checkcast`	class
`instanceof`	class
`anewarray`	class
`multianewarray`	class
`new`	class
`getfield`	field
`getstatic`	field
`putfield`	field
`putstatic`	field
`invokeinterface`	interface method
`invokespecial`	method
`invokestatic`	method
`invokevirtual`	method

Figure 1: Instructions causing name resolution

The JVM specification uses the term *name resolution* for the process that from a reference in an instruction loads and links the code necessary for the execution of the instruction. Whether the reference is to a class c or to a method of c the net effect is to link c into the machine. Instructions that involve name resolution, together with the required action, are listed in Table 1. All other instructions leave the state of loaded and linked classes unchanged.

The rule for *invokevirtual* reads

$$\frac{\begin{array}{l} \mathit{origin}(m) = n \\ S, \mathit{cc} \vdash \mathit{link}(n) \rhd S' \end{array}}{S, \mathit{cc} \vdash \mathit{invokevirtual}(m) \rhd S'} \quad (28)$$

and all method-referencing instructions are defined like this. Instructions with class references are treated similarly.

7 A bug in the ClassLoader mechanism

V. Saraswat has recently reported a bug in the way in which the JVM determines whether two classes are equivalent [13]. This bug can be exploited to cause *type confusion* between two classes with the well known consequences for the security such as providing access to `private` variables from other classes [10]. Consider the following four class files $\mathit{cf}_{\mathtt{R1}}$, $\mathit{cf}_{\mathtt{R2}}$, $\mathit{cf}_{\mathtt{RR}}$, and $\mathit{cf}_{\mathtt{RT}}$ in figure 2, all of whose superclass is `Object`.[6] Assume furthermore that we have two class loaders cl_1 and cl_2 satisfying

[6] To save space we will write `Object` when we actually mean `java.lang.Object`.

Class file : $cf_{\texttt{R1}}$

```
class R
  private int i=1
```

Class file : $cf_{\texttt{R2}}$

```
class R
  public int i=1
```

Class file : $cf_{\texttt{RR}}$

```
class RR
  R getR()
    return new R()
```

Class file : $cf_{\texttt{RT}}$

```
class RT
  private R r
  void test()
    RR rr=new RR()
    r=rr.getR()   (*)
```

Figure 2: Saraswats type confusion example

$$\mathcal{W}(cl_1, n) = \begin{cases} \mathcal{W}(scl, n) & \text{if } n \in \texttt{java.lang.*} \\ (cf_{\texttt{R1}}, cl_1) & \text{if } n = \texttt{R} \\ (cf_{\texttt{RR}}, cl_1) & \text{if } n = \texttt{RR} \end{cases}$$

$$\mathcal{W}(cl_2, n) = \begin{cases} \mathcal{W}(cl_1, n) & \text{if } n = \texttt{RR} \\ (cf_{\texttt{R2}}, cl_2) & \text{if } n = \texttt{R} \\ (cf_{\texttt{RT}}, cl_2) & \text{if } n = \texttt{RT} \end{cases}$$

We recall that a loaded class consists of a class file and a class loader. As an example we have $(cf_{\texttt{R2}}, cl_2)$, denoting the result of loading class file $cf_{\texttt{R2}}$ with class loader cl_2.

Informally, the problem can be explained as follows. The value returned by the expression `rr.getR()` in the statement $(*)$ will be an instance of the loaded class $(cf_{\texttt{R1}}, cl_1)$ since RR was loaded by cl_1 and resolution of R in the class RR is done wrt. cl_1. On the other hand, the R appearing in RT will be loaded by cl_2 and therefore yield $(cf_{\texttt{R2}}, cl_2)$. These two classes are not the same and the assignment should fail. However, if only the names of the classes are checked for equality then the assignment is deemed correct, leading to a situation where the once `private` field `i` of class R is now considered `public`. See the report by Saraswat [13] for an extensive discussion of this problem.

In this section we show how this problem and its solution can be become apparent by analysing the program using our formalism. The following deduction shows how the machine state evolves by linking RT when the current class cc has been loaded by class loader cl_2 *i.e.*, $cc.cl = cl_2$. We make the assumption that loading and linking the class `Object` with the system class loader *scl* does not affect the state, *i.e.*, that

$$\textit{If } cc.cl = scl \textit{ then } \begin{cases} S, cc \vdash \textit{load}(\texttt{Object}) \rhd S \textit{ and} \\ S, cc \vdash \textit{link}(\texttt{Object}) \rhd S. \end{cases}$$

Let now the current class be loaded by cl_2 *i.e.*, that $cc.cl = cl_2$. We can instantiate the rule defining linking as follows

$$\frac{\begin{array}{l} S, cc \vdash \textit{load}(\texttt{RT}) \rhd S' \\ \mathcal{W}(cc.cl, RT) = (cf_{\texttt{RT}}, cl_2) \\ \textit{super}(cf_{\texttt{RT}}) = \texttt{Object} \\ S', cc \vdash \textit{link}(\texttt{Object}) \rhd S' \\ S', cc \vdash \textit{verify}((cf_{\texttt{RT}}, cl_2)) \rhd S'' \end{array}}{S, cc \vdash \textit{link}(\texttt{RT}) \rhd S''}$$

and we aim at showing that setting

$$S' = S \cup \{(cf_{\texttt{RT}}, cl_2)\}$$

and

$$S'' = S' \cup \{(cf_{\texttt{RR}}, cl_1), (cf_{\texttt{R2}}, cl_2)\}$$

yields a provable instance.

The two premisses in the middle are immediately satisfied so we focus on the first and the last premiss. By the rule for *load*,

$$\frac{\begin{array}{l} cc.cl = cl_2 \\ \mathcal{W}(cc.cl, \texttt{RT}) = \mathcal{W}(cl_2, \texttt{RT}) = (cf_{\texttt{RT}}, cl_2) \\ \textit{super}(cf_{\texttt{RT}}) = \texttt{Object} \\ S, cc \vdash \textit{load}(\texttt{Object}) \rhd S \end{array}}{S, cc \vdash \textit{load}(\texttt{RT}) \rhd S \cup \{(cf_{\texttt{RT}}, cl_2)\}}$$

Similarly, the verification of the body of RT will cause the loading of all classes referenced in RT:

$$\frac{\begin{array}{l} \{\texttt{RR}, \texttt{R}\} = \textit{references}(cf_{\texttt{RT}}) \\ S', cc \vdash \textit{load}^*(\{\texttt{RR}, \texttt{R}\}) \rhd S' \cup \{(cf_{\texttt{RR}}, cl_1), (cf_{\texttt{R2}}, cl_2)\} \end{array}}{S', cc \vdash \textit{verify}((cf_{\texttt{RT}}, cl_2)) \rhd S' \cup \{(cf_{\texttt{RR}}, cl_1), (cf_{\texttt{R2}}, cl_2)\}}$$

The effect of loading R and RR can be found by the following deductions. Assume for the moment that classes are loaded in the order they appear in the text. Then

$$S', cc \vdash \textit{load}^*(\{\texttt{RR}, \texttt{R}\}) \rhd S' \cup \{(cf_{\texttt{RR}}, cl_1), (cf_{\texttt{R2}}, cl_2)\}$$

since

$$\frac{\begin{array}{l} \mathcal{W}(cc.cl, \texttt{RR}) = \mathcal{W}(cl_2, \texttt{RR}) = (cf_{\texttt{RR}}, cl_1) \\ \textit{super}(cf_{\texttt{RR}}) = \texttt{Object} \\ S', cc \vdash \textit{load}(\texttt{Object}) \rhd S' \end{array}}{S', cc \vdash \textit{load}(\texttt{RR}) \rhd S' \cup \{(cf_{\texttt{RR}}, cl_1)\}}$$

and

$$\frac{\begin{array}{l} \mathcal{W}(cc.cl, \texttt{R}) = \mathcal{W}(cl_2, \texttt{R}) = (cf_{\texttt{R2}}, cl_2) \\ \textit{super}(cf_{\texttt{R2}}) = \texttt{Object} \\ S' \cup \{(cf_{\texttt{RR}}, cl_1)\}, cc \vdash \textit{load}(\texttt{Object}) \rhd \\ \qquad S' \cup \{(cf_{\texttt{RR}}, cl_1)\} \end{array}}{S' \cup \{(cf_{\texttt{RR}}, cl_1)\}, cc \vdash \textit{load}(\texttt{R}) \rhd S' \cup \{(cf_{\texttt{RR}}, cl_1), (cf_{\texttt{R2}}, cl_2)\}}$$

Executing the code of RT in state S'', cc leads to executing the instruction

$$\mathit{invokevirtual}((\mathtt{RR}, \mathtt{getR()}))$$

whose effect on the state of the machine can be determined according to the rule from Section 6.4 defining this instruction:

$$\frac{\begin{array}{c}\mathit{origin}(\mathtt{RR.getR}()) = \mathtt{RR} \\ S'', cc \vdash \mathit{link}(\mathtt{RR}) \rhd S'' \cup \{(cf_{\mathtt{R1}}, cl_1)\}\end{array}}{S'', cc \vdash \mathit{invokevirtual}((\mathtt{RR}, \mathtt{getR()})) \rhd S'' \cup \{(cf_{\mathtt{R1}}, cl_1)\}}$$

which leads to analysing the linking of RR. The class loader of the current class has already loaded RR; we omit the formal proof that re-executing *load*(RR) in state S'', cc does not affect state S''. However, the verification of RR leads to the loading of all classes mentioned in its methods.

$$\frac{\begin{array}{c}S'', cc \vdash \mathit{load}(\mathtt{RR}) \rhd S'' \\ (cf_{\mathtt{RR}}, cl_1) = \mathcal{W}(cc.cl, \mathtt{RR}) \\ \mathit{super}(cf_{\mathtt{RR}}) = \mathtt{Object} \\ S'', cc \vdash \mathit{link}(\mathtt{Object}) \rhd S''\end{array}}{S'', cc \vdash \mathit{verify}((cf_{\mathtt{RR}}, cl_1)) \rhd S'' \cup \{(cf_{\mathtt{R1}}, cl_1)\}}$$

since the verification of RR satisfies

$$\frac{\begin{array}{c}\{\mathtt{R}\} = \mathit{references}(cf_{\mathtt{RR}}) \\ S'' \vdash \mathit{load}((cf_{\mathtt{RR}}, cl_1), \mathtt{R}) \rhd S'' \cup \{(cf_{\mathtt{R1}}, cl_1)\}\end{array}}{S'', cc \vdash \mathit{verify}((cf_{\mathtt{RR}}, cl_1)) \rhd S'' \cup \{(cf_{\mathtt{R1}}, cl_1)\}}$$

where the second premiss is deduced as follows:

$$\frac{\begin{array}{c}\mathcal{W}(cl_1, \mathtt{R}) = (cf_{\mathtt{R1}}, cl_1) \\ \mathit{super}(cf_{\mathtt{R1}}) = \mathtt{Object} \\ S'', cc \vdash \mathit{load}(\mathtt{Object}) \rhd S''\end{array}}{S'' \vdash \mathit{load}((cf_{\mathtt{RR}}, cl_1), \mathtt{R}) \rhd S'' \cup \{(cf_{\mathtt{R1}}, cl_1)\}}$$

The state after invoking the method getR thus looks as follows:

$$S \cup \{(cf_{\mathtt{RT}}, cl_2), (cf_{\mathtt{RR}}, cl_1), (cf_{\mathtt{R2}}, cl_2), (cf_{\mathtt{R1}}, cl_1)\}$$

and the last two loaded classes satisfy

$$\mathit{name}((cf_{\mathtt{R2}}, cl_2).cf) = \mathit{name}((cf_{\mathtt{R1}}, cl_1).cf)$$

from which the type confusion arises. More precisely, the verification should find an error as follows. The faulty statement (marked with an asterisk in class file $cf_{\mathtt{RT}}$) is implemented by a call to *invokevirtual* with argument RR.getR() followed by an assignment operation *putfield* to RT.r. When type checking the latter instruction, the simulated run-time stack will contain the class of the result of RR.getR() (*i.e.*, $(cf_{\mathtt{R1}}, cl_1)$) on top of the class of the invoking object (*i.e.*, $(cf_{\mathtt{RT}}, cl_2)$), and the following deduction would be required to hold.

$$\frac{\begin{array}{l}ST = (cf_{\mathtt{R1}}, cl_1) :: (cf_{\mathtt{RT}}, cl_2) :: ST\text{'} \\ r = (\mathtt{RT}, \mathtt{r}) \\ r \text{ member-of } (cf_{\mathtt{RT}}, cl_2) \\ r_c = \mathit{member\text{-}type}(\mathcal{W}(cc.cl, \mathit{origin}(r)).cf, r) \\ \quad = \mathit{member\text{-}type}(\mathcal{W}(cl_2, \mathtt{RT}).cf, r) \\ \quad = \mathit{member\text{-}type}((cf_{\mathtt{RT}}, cl_2).cf, r) = \mathtt{R} \\ r_c \in \mathit{Class} \\ (cf_{\mathtt{R1}}, cl_1) \quad \text{sub}^* \quad \mathcal{W}(cc.cl, \mathtt{R}) \\ \qquad\qquad\qquad\quad = \mathcal{W}(cl_2, \mathtt{R}) \\ \qquad\qquad\qquad\quad = (cf_{\mathtt{R2}}, cl_2)\end{array}}{S, cc, ST \vdash \mathit{typecheck}(\mathit{putfield}((\mathtt{RT}, \mathtt{r})))}$$

What has become apparent from the formalisation is that this deduction is not valid: the predicate $(cf_{\mathtt{R1}}, cl_1)$ sub* $(cf_{\mathtt{R2}}, cl_2)$ does not hold because the class loaders are different. The error committed by the JVM is to implement the sub* relation by using only the name equivalence mentioned above without checking that the class loaders are the same.

8 Related work and avenues for further research

The linking process has not received much attention from the semantics community so far. Its importance has been recognised recently by Cardelli [2] who proposes a formalisation of linking as a process of combining program fragments. The main goal in [2] is to provide a formal basis for the design of programming languages and module systems and it considers only static linking. A formal study of dynamic linking and its interaction with static typing is reported by Dean [3]. The paper presents a model of dynamic linking which is closely related to Java and proposes a restriction on the dynamic linking operation in order to ensure its soundness with respect to the static type checking. The proof has been carried out in PVS and Dean provides only a sketch of its structure. This restriction on dynamic loading imposes that the (class types) context is monotonically increasing (a new context must always be a consistent extension of the previous one). The practical consequence for implementors is that no class definition should be loaded more than once for each class loader, which is also required and implemented in the lastest versions of JVM. In contrast to this work, we detail the rôle of the class loader identity in the relationship between types and classes and the interaction between dynamic loading and visibility rules rather than static typing.

The formalisations of the semantics of Java published so far deal exclusively with either the type system [4, 14] of the compiler or the reduction rules of the virtual machine [1, 12]. Drossopoulou and Eisenbach [4] presents a formalisation of the type system of a substantial subset of Java (including arrays, dynamic method binding, object creation, exception handling, ...). The type inference system annotates Java programs with types and the operational semantics of annotated programs is specified in terms of a rewrite system defining a relation between configurations (tuples of terms and states). A subject reduction property is then established expressing the fact that well-typed programs rewrite either to an exception or to another well-typed term of the same type (up to the subclass/subinterface relationship). This shows a form of soundness of the type system for this subset of Java. A variation of this formalisation has been expressed and mechanically verified by Syme using a higher-order logic proof system called *Declare* [14]. The formalisation of the type system of Java is extremely useful both for a better understanding of the language and for discovering mistakes or omissions in the informal specifications. The cited work however does not consider dynamic loading and visibility rules (modifiers) which play a significant role in the security of Java. These aspects are precisely the core of our contribution. On the other hand, we do not deal with type checking ourselves, considering it an orthogonal issue. So, the results presented in this paper are complementary to [4, 14]. This complementarity goes in both directions: visibility rules should be taken into account within type checking and a form of type checking should be included in the "load time" verifications of the abstract machine. This integration is a natural avenue for further work. Note however that the formalisation proposed by Drossopoulou and Eisenbach would need to be reformulated to be used in this context because it applies to source code whereas we need verification rules of the byte code. To the best of our knowledge, the only attempt at defining byte code verifications formally has been carried out by Qian [12] who includes a proof of soundness of a bytecode verifier with respect to a simplified specification of the Java virtual machine. However, as opposed to the work described here, Qian starts with the "closed world assumption" (all classes have been previously loaded into the system by a single class loader) and does not take visibility rules into account.

Bertelsen [1] presents a very precise definition of the virtual machine based on the informal specification in [9]. In addition to the heap and the environment, the state of the machine includes a stack of frames, each one corresponding to a method invocation. A frame consists of a program counter, an operand stack, a set of local variables and a current method identifier. The report by Bertelsen contains a comprehensive definition of the instruction set of the Java Virtual Machine (only some instructions cannot be described precisely because they require concrete representations of the state). This specification was precise enough to reveal a number of ambiguities and errors in [9]. Because of its low level and exhaustive nature, this specification is in fact close to a formal reference manual and it is not evident whether it can easily be exploited to support formal reasoning on Java programs. In contrast to Bertelsen, we have focussed here on one specific aspect of the semantics (the visibility rules, their use in the virtual machine and their dynamic evolution through dynamic loading) in order to highlight the choices made in the specification of the Java language and their impact on security.

Further work is still needed to relate our inference systems to that of Bertelsen. We believe however that the work described here suggests another, more promising, approach: rather than providing the full semantics of a complex programming language from scratch, it would be more tractable (and enlightening) to define different views of the semantics and then show their integration and consistentcy. This notion of view is thus close to the *facets* in Action Semantics[7] [11]. It should be clear that all the contributions reviewed here (except [1] which aims at comprehensiveness) focus on complementary aspects and shed some light on the language. Determining what the most interesting views of the semantics should be and how they interfere is certainly one major avenue for future research in this area.

Another important extension of the existing results is the formalisation of specific security policies for Java applications [5, 6, 7] and the study of the impact of the safety properties studied so far on security issues. The soundness of the type system and its interaction with dynamic loading is definitely a prerequisite to enforce security (because type errors can be exploited to breach security rules) but security goes beyond mere type checking. What we would like to ensure ultimately is confidentiality and integrity properties. A short review of existing works in this direction can be found in [15].

References

[1] P. Bertelsen. Semantics of Java byte code. Technical report, Department of Information Technology, Technical University of Denmark, March 1997.

[2] L. Cardelli. Program fragments, linking and modularization. In *Proc. of 24th ACM Symposium on Principles of Programming Languages*, pages 266–277. ACM Press, 1997.

[7] The motivations for facets in Action Semantics are very much along the lines of the above discussion (tackling the complexity of the semantics of real programming language and providing a better insight on the main design choices through a separation of concerns) but they represent independent kinds of information (like functionality, communications, declarations).

[3] D. Dean. The security of static typing with dynamic linking. In *Proc. of 4th ACM Conf. on Computer and Communications Security*, pages 18–27. ACM Press, 1997.

[4] S. Drossopoulou and S. Eisenbach. Java is type safe – probably. In *11th European Conference on Object Oriented Programming*, June 1997.

[5] L. Gong. Going beyond the sandbox: An overview of the new security architecture in the Java development kit 1.2. In *Proc. of USENIX Symposium on Internet Technologies and Systems*, December 1997.

[6] L. Gong. Java security: Present and near future. *IEEE Micro*, May-June:14–19, 1997.

[7] L. Gong. New security architectural directions for Java. In *Proc. of IEEE COMPCON*, pages 97–102, 1997.

[8] J. Gosling, B. Joy, and G. Steele. *The Java Language Specification*. Addison-Wesley, 1996.

[9] T. Lindholm and F. Yelling. *The Java Virtual Machine Specification*. Addison-Wesley, 1997.

[10] G. McGraw and E. Felten. *Java Security: hostile applets, holes and antidotes*. J. Wiley & Sons, 1997.

[11] P. Mosses. *Action Semantics*, volume 26 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 1992.

[12] Z. Qian. A formal specification of Java virtual machine instructions. Technical report, Universität Bremen, 1997.

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

[14] D. Syme. Proving Java type soundness. Technical Report 427, Cambridge University, June 1997.

[15] T. Thorn. Programming Languages for Mobile Code. *ACM Computing Surveys*, September 1997.