A Formal Specification of Java™ Class Loading

Zhenyu Qian Allen Goldberg* Alessandro Coglio

Kestrel Institute, 3260 Hillview Avenue, Palo Alto, CA 94304
{qian, goldberg, coglio}@kestrel.edu

ABSTRACT

The Java Virtual Machine (JVM) has a novel and powerful mechanism to support lazy, dynamic class loading according to user-definable policies. Class loading directly impacts type safety, on which the security of Java applications is based. Conceptual bugs in the loading mechanism were found in earlier versions of the JVM that lead to type violations. A deeper understanding of the class loading mechanism, through such means as formal analysis, will improve our confidence that no additional bugs are present.

The work presented in this paper provides a formal specification of (the relevant aspects of) class loading in the JVM and proves its type safety. Our approach to proving type safety is different from the usual ones since classes are dynamically loaded and full type information may not be statically available. In addition, we propose an improvement in the interaction between class loading and bytecode verification, which is cleaner and enables lazier loading.

1. INTRODUCTION

The Java Virtual Machine (JVM) has a novel and powerful class loading mechanism. Class loading is the process of obtaining a representation of a class (declaration), called a *class file*, and installing that representation within the JVM. The JVM allows lazy, dynamic loading of classes, user-definable loading policies, and a form of name space separation using loaders. According to both the Java and JVM specifications [13, 16] distinct loaded classes may have the same name, and within an executing JVM each loaded class is identified by its name plus the class loader that has loaded it.

One of the key properties of the JVM, from both a security and software engineering perspective, is *type safety*. If the JVM confuses distinct classes, type safety problems can result. Ensuring type safety requires a fairly sophisticated mechanism. The reason is that on one hand it is impossible to determine, prior to execution, the actual loader of a class because loaders can delegate class loading to each other according to the program logic of user-written code. On the other hand, loading a class before it is required for execution is undesirable. Saraswat first publicly reported [20] name spoofing problems due to deficiencies in the mechanisms employed by earlier versions of the JVM.

The Sun solution [15] to such problems employs a constraint mechanism. Constraints on the disambiguation of names to ensure type safety are posted when necessary. As classes are loaded the constraints are checked.

The main result of this paper is to provide formal arguments that the Sun approach is sufficient to prove type safety. We formalize the operational semantics of a simplified JVM that uses this approach. The operational semantics models class loading, resolution, bytecode verification, and execution of some selected instructions.

Our model of the JVM departs from Sun's regarding how the bytecode verifier checks subtype relationships. Sun's bytecode verifier performs the checks by loading certain referenced classes, while our verifier uniformly posts constraints. These subtype constraints are checked when classes are loaded, analogously to the constraints mentioned above. The advantage of our approach is lazier class loading and clearer interaction between verification and class loading.

Because a class loader is a runtime object, a reference to a class (in a class file) is just a name and cannot include the loader. Disambiguation of such a name is needed, but can only occur when the referenced class has been loaded. This implies that traditional approaches to proving type safety, in which a static type semantics is extracted from the source code and related to the dynamic semantics, cannot be applied. Instead, our type safety proof relates the dynamic semantics with static type information, currently loaded classes, and currently posted constraints (which express requirements on not-yet-loaded classes).

Our analysis led us to identify bugs in the current Sun implementation of the JVM, some of which relate to inadequacies in the JVM specification [16][1]. The bugs result from the fail-

[*] A. Goldberg's current affiliation and address: Shoulders Corp., 800 West El Camino Real, Mountain View, CA 94040.

[1] Some of these bugs were independently discovered and re-

ure of the bytecode verifier to properly disambiguate names. For a full description of these bugs see [4], where it is also shown how the bugs are avoided by having the verifier post subtype constraints.

The remainder of the paper is organized as follows. In the next section we present relevant JVM concepts and examples. Section 3 gives an overview of our approach. Section 4 describes the key points of our formalization and proof. Section 5 discusses related work, and section 6 states conclusions.

2. BACKGROUND

2.1 Class objects

A class file is typically produced by compiling a Java class (declaration): the class file contains essentially the same information, except that the code of each of its methods is compiled to *bytecode*, i.e. an assembly-like language that uses an operand stack and a register array, both local to the method. The registers are also referred to as *local variables*. The JVM executes bytecode.

The internal representations of classes in an executing JVM are objects of the system class named `Class`, and they are called *class objects*. We will use *Obj*, *ClLd* and *Cls* to denote the class objects for the system classes named `Object`, `ClassLoader` and `Class`. For convenience, we will not distinguish between a class and its corresponding class object in the informal discussion below.

2.2 Resolution and loading

Bytecode instructions use (fully qualified) names to reference classes. *Class resolution* is the process of replacing these symbolic references with (pointers to) actual class objects in the executing JVM. Resolution causes loading of the class and checking of the access control modifiers of the loaded class.

Bytecode instructions reference fields and methods by name, including the name of the class where they are expected to be declared. *Field* and *method resolution* is the process of replacing these symbolic references with (pointers to) actual fields and methods. It requires first resolving the class in which the field or method is declared, then checking the presence of the field or method in the resolved class, and finally checking its access control modifiers.

(Class) loaders are objects of subclasses of class *ClLd*. By overriding certain methods of class *ClLd*, user-defined class loaders can implement arbitrary loading policies.

Class *ClLd* contains a `defineClass` method. It is a `final` method and thus cannot be overridden in subclasses of class *ClLd*. This method takes a class file (in the form of a byte array) as argument, and returns a newly created class object, unless the class file has an invalid format – in which case an exception is thrown. The creation of the new class requires resolution of all its superclasses. If the `defineClass` method is invoked on a loader and a class object is returned, then the loader is called the *defining loader* of the resulting class.

Class *ClLd* contains a `loadClass` method, which may be overridden in subclasses. This method takes a class name (in the form of a string) as argument, and returns a class object (or throws an exception). The user's code in this method can implement arbitrary loading policies. Typically the code will fetch a class file in some way (e.g., from the local file system, or a network connection) and then invoke `defineClass` with the resulting class file as argument. However, user's code can delegate loading by calling `loadClass` upon another loader. If the `loadClass` method is invoked by the JVM on a loader and a class object is returned, then the loader is called an *initiating loader* of the resulting class. The defining loader of a loaded class is also regarded as an initiating loader of the class.

Therefore, a loaded class may have many initiating loaders but only one defining loader.

When a class name needs to be resolved within an executing class, the defining loader of the executing class is used as initiating loader for the class (name) to be resolved.

2.3 A simple example

Let us use an example to explain some basic concepts and issues regarding classes, loaders, and type safety.

Assume that we have a subclass *MyLd* of class *ClLd*, two distinct objects l_1 and l_2 of class *MyLd*, and five classes C, D, E, T_1 and T_2, satisfying the following:

- Classes C, D, E, T_1 and T_2 have names `C`, `D`, `E`, `T` and `T`, respectively.
- Classes C, E and T_1 have defining loader l_1, and classes D and T_2 have defining loader l_2.
- Using loader l_1 as initiating loader for names `D` and `T` yields the classes D and T_1, respectively, while using loader l_2 as initiating loader for names `E` and `T` yields classes E and T_2, respectively. This means that l_1 delegates to l_2 the loading of a class named `D`, and that l_2 delegates to l_1 the loading of a class named `E`.

Figure 1 shows classes C, D, E, T_1 and T_2. We consider execution starting with the `m()` method. Initially, only class C is loaded. The others are loaded when they are needed, i.e. at their first active use (see below)[2].

Starting at the `m()` method, the JVM first resolves names `D` and `T`, loading classes D and T_1 using l_1 as initiating loader for names `D` and `T`, respectively. Then it creates objects of these classes. The `n(T t)` method is then invoked with an object of class T_1 as argument.

Class D has defining loader l_2. Within the `n(T t)` method, name `E` is resolved, loading class E using l_2 as initiating loader for name `E`. Then the `k(T t)` method is invoked with the object of class T_1 as argument.

ported in [21].

[2] The JVM specification allows loading of a class to happen at any time no later than its first active use. In this example, we assume that loading of a class happens exactly at its first active use.

```
public class C {              // Class $C$ with defining loader $l_1$.
  public void m() {
    (new D()).n(new T());     // Load classes $D$ and $T_1$ and pass a $T_1$ object as argument.
  }
}
public class D {              // Class $D$ with defining loader $l_2$.
  public void n(T t) {        // Receive a $T_1$ object as argument.
    (new E()).k(t);           // Load class $E$ and pass the $T_1$ object as argument.
    int i = t.f;              // Load class $T_2$ and fail since $T_1 \neq T_2$!
  }
}
public class E {              // Class $E$ with defining loader $l_1$.
  public void k(T t) {        // Receive a $T_1$ object as argument.
    Object o = t.f;           // Operation succeeds.
  }
}
public class T {              // Class $T_1$ with defining loader $l_1$.
  Object f;
}
public class T {              // Class $T_2$ with defining loader $l_2$.
  int f;
}
```

Figure 1: A simple example

Class E has defining loader l_1. Within the `k(T t)` method, the execution of the field access `t.f` causes resolution of class name `T`, using l_1 as initiating loader. The resulting class is T_1. The field access succeeds because the resolved class of the formal parameter `t` is the same as that of the actual value it holds.

Returning back to the `n(T t)` method, the execution of the field access `t.f` causes resolution of class name `T`, using l_2 as initiating loader. The resulting class is T_2. Now execution will throw an exception (and prevent field access) because the JVM detects that the class T_1 of the actual value in the formal parameter `t` does not match the class T_2 of the formal parameter `t`.

We observe that in the execution just described no exception is thrown until the field access `t.f` in the `n(T t)` method is attempted. The reason is that the JVM cannot check whether the class T_2 of the formal parameter `t` matches the class T_1 of the value it holds until the class T_2 is loaded. In fact, the JVM would not throw an exception if the field access `t.f` in the `n(T t)` method were not present.

The usual approach to proving type safety is to show that during execution values stored in variables always conform to the types statically assigned to these variables. In our example, this means showing that, after the `n(T t)` method is invoked, the T_1 object passed as argument conforms to "the type statically assigned to formal parameter `t` of `n(T t)`". However, the JVM cannot determine what the class the name `T` denotes, until the name `T` is actually resolved in class D. In fact, if the class were loaded at the time when the `n(T t)` method is invoked, the JVM would be able to detect that the value does not conform to the loaded class.

The bug reported by Saraswat [20] led to type violations in earlier versions of the JVM. In reference to the example in Figure 1, the earlier versions of the JVM did not check the equality of classes T_1 and T_2 when executing the field access `t.f` in the `n(T t)` method. The result of such field access was unpredictable, since it operated on an object of the wrong type.

The Sun solution to Saraswat's bug, which checks the equality of classes T_1 and T_2 when executing the field access `t.f` in the `n(T t)` method in Figure 1, introduce two internal data structures into the JVM called *loaded class cache* and *loading constraints* [15].

2.4 Loaded class caches and loading constraints

The JVM resolves referenced class names to loaded classes. The results of the loading process are stored in a JVM data structure called the *loaded class cache*. The loaded class cache maps an initiating loader and a class name to a loaded class. Recall that a reference to a class is resolved by using the defining loader of the current class as the initiating loader of the referenced class; so the loaded class cache records how a class name is disambiguated.

When a class name needs to be resolved, the JVM, before loading with a given initiating loader, checks the loaded class cache. If a class has been loaded for that initiating loader and that class name, the JVM returns the already loaded class as the result. Thus resolution will always give consistent results. If the loaded class cache has no entry for the initiating loader and class name then loading is carried out. If a class is returned, the loaded class cache is updated with a new entry for the just loaded class. If instead an exception is thrown, resolution fails.

In the sequel, we call a pair $\langle l, cn\rangle$ of an initiating loader l and a class name cn a *loading request.*

A loaded class cache is a finite map from loading requests to loaded classes:

$$\{\langle l_1, cn_1\rangle \mapsto c_1, \cdots, \langle l_n, cn_n\rangle \mapsto c_n\}.$$

A *loading constraint* is a triple $\langle l, l', cn\rangle$ consisting of two loaders l and l' and a class name cn. A loading constraint as above expresses the requirement that using l and l' as the initiating loaders for name cn, must yield the same (loaded) class if they both succeed. Loading constraints are generated by the JVM when fields and methods are resolved. Such constraints enforce that classes exchanging objects (through field access and method invocation) agree on the actual classes (and not only the names) of the exchanged objects.

An executing JVM maintains a loaded class cache and a set of loading constraints. These two data structures are checked for mutual consistency whenever either one is updated. Whenever an update causes a violation of a loading constraint w.r.t. the loaded class cache, an exception is thrown, thus causing a failure of the operation that triggered the update.

In the example in Figure 1, a loading constraint $\langle l_1, l_2, \mathtt{T}\rangle$ is generated when the `n(T t)` method is resolved (from C) during execution of `(new D()).n(new T())`. The loader l_1 is the defining loader for class C, which is used as an initiating loader for name `T` within the `m()` method. The loader l_2 is the defining loader for class D, which will be used as an initiating loader for name `T` within the `n(T t)` method, when `T` is resolved from D. Class T_1 is loaded for the loading request $\langle l_1, \mathtt{T}\rangle$ in the execution of the instruction `new T()`. The loading constraint is checked only when class T_2 is loaded for the loading request $\langle l_2, \mathtt{T}\rangle$, during the execution of the field access `t.f` within the `n(T t)` method.

2.5 Bytecode verification

Type-safe execution requires that each instruction operates on data with appropriate types. In order to reduce the number of runtime checks and thus to improve efficiency, most type-safety requirements are checked statically. In the JVM, the code of each loaded class is verified prior to execution of any of its methods, by a JVM component called the *bytecode verifier.* Bytecode verification should assign types to operand stack elements and local variables for each instruction, consistently with the types required by the instructions in the method. However, in order to avoid premature loading, the bytecode verifier only assigns class names, instead of class types, to local variables and operand stack elements. Therefore, the bytecode verifier only makes "partial" checks on class types; such checks are in fact complemented by the loading constraint mechanisms at runtime.

The following is the bytecode of the `m()`, `n(T t)` and `k(T t)` methods in Figure 1. The comments on the right give the class names assigned by the bytecode verifier to some elements in the operand stack at some program points.

```
method m()
 p1: new(D)
 p2: new(T)                      //  [··· ,D]
 p3: invokevirtual(D,n,T,void)   //  [··· ,D,T]
    ...
method n(T)
    ...
 r1: getfield(T,f,int)           //  [··· ,T]
 r2: ...                         //  [··· ,int]
    ...
method k(T)
    ...
 q1: getfield(T,f,Object)        //  [··· ,T]
 q2: ...                         //  [··· ,Object]
    ...
```

Intuitively, the instruction `new(D)` creates a new object of class D and pushes (a reference to) it onto the operand stack[3]. Thus, bytecode verification assigns the name `D` to the top element in the operand stack at program point `p2` after that instruction. The instruction `new(T)` at program point `p2` works in a similar way.

The instruction `invokevirtual(D,n,T,void)` dynamically selects the `n(T)` method based on the class of the second top object on the operand stack, and invokes the method with the top element in the operand stack as argument. The instruction contains a class name `D` indicating the class in which the method is to be found, and a class name `T` indicating the class of the argument. Bytecode verification uses this information to check the consistency of the class names assigned to the top stack positions. In this case, it checks whether the class names `D` and `T` are the ones assigned to the top two elements in the operand stack. Note that this static consistency check is not sufficient to guarantee type safety: this is the reason why the executing JVM will generate a loading constraint $\langle l_1, l_2, \mathtt{T}\rangle$, as explained in Section 2.4.

The instruction `getfield(T,f,int)` fetches the `int` value of the field `f` in the top object of the operand stack and pushes it onto the operand stack. The instruction contains a class name `T` indicating the class in which the field is to be found. Again bytecode verification uses this information to check the consistency of the class name assigned to the top stack position. In this case, it checks whether class name `T` is the one assigned to the top element in the operand stack. At the next program point `r2`, bytecode verification assigns type `int` to the top element of the operand stack.

Bytecode verification analyzes `getfield(T,f,Object)` in a similar way as `getfield(T,f,int)`. The only difference is that the class name `Object` is assigned to the top element of the operand stack at program point `q2`.

2.6 Another example

Let us now consider an example involving subtypes. This example provides some motivation for the subtype constraints that will be introduced and explained in the next sections.

[3] For simplicity, we ignore the issue of object initialization in this example. In other words, we assume that the bytecode instruction `new` creates a fully initialized object.

We assume that we have a subclass *MyLd* of class *ClLd*, two distinct objects l_1 and l_2 of class *MyLd*, and five classes C, D, T_1, T_2 and S, satisfying the following:

- Classes C, D, T_1, T_2 and S have names `C`, `D`, `T`, `T` and `S` respectively.
- Classes C, D and T_1 have defining loader l_1, and classes T_2 and S have defining loader l_2.
- Using loader l_1 as initiating loader for names `D`, `T` and `S` yields the classes D, T_1 and S, respectively, while using loader l_2 as initiating loader for name `T` yields class T_2, respectively. This means that l_1 delegates to l_2 the loading of a class named `S`.

Note that S is a subclass of T_2, and not of T_1, because resolving name `T` with l_2 (which is the defining loader of S) as initiating loader yields T_2.

Figure 2 shows classes C, D, T_1, T_2 and S. We assume that initially, only class C is loaded, and consider execution starting with the `m()` method.

Based on analogies with the previous example, one might expect execution to happen as follows. First, classes D and S are loaded. According to the JVM specification, when a class is loaded, all its superclasses are also loaded. Therefore, when S is loaded T_2 is also loaded.

Next, an object of class D and one of class S are created, and method `n(T t)` is invoked upon the D object, passing the S object as argument. The field access `t.f` causes resolution of name `T` from class D, which results in class T_1. At this point, field access should fail because S is not a subclass of T_1.

However, that is not quite what happens in the JVM. Before method `m()` is executed, it must go through bytecode verification. Let us examine the bytecode for this method, together with the class names assigned by the bytecode verifier to some elements of the operand stack:

```
method m()
 p1: new(D)
 p2: new(S)                     //  [⋯,D]
 p3: invokevirtual(D,n,T,void)  //  [⋯,D,S]
     ...
```

The difference between this and the `m()` method in the previous example is that the name `T` assigned to the top element in the operand stack at program point `p3` is now replaced with the name `S`. Since the `invokevirtual` instruction references name `T`, the JVM specification actually requires bytecode verification to resolve both names `T` and `S` from class C (i.e., using l_1 as initiating loader), and to check that the appropriate subclass relationship holds.

The check fails, because the loading request $\langle l_1, \mathtt{T}\rangle$ yields T_1, which is not a superclass of S. Therefore, bytecode verification throws an exception, and method `m()` is not executed at all.

Of course, the same would happen even if the field access `t.f` were not present in method `n(T t)`. In fact, when class C is being verified, class D has not been loaded.

This suggests that resolving class names during bytecode verification in order to check subclass relationships, does not lead to the laziest possible loading strategy. In the next sections, we will show how subtype constraints allow lazier loading, and how the behavior of the JVM in this example would match what one would probably expect.

3. OVERVIEW OF OUR APPROACH

The main objective of our work is to raise the assurance that the JVM as specified and implemented by Sun is safe. Because the mechanisms that enforce safety in the JVM are inherently complex, it is not straightforward to assess their correctness, as demonstrated by the discovery of bugs that lead to type safety violations [20, 21, 4]. In order to raise assurance, we construct a formal specification, which is consistent with the JVM English specification and the Sun implementation, and we prove results about that specification. Actually, our specification intentionally differs from Sun's in one aspect, namely the use of subtype constraints for bytecode verification, which enable a cleaner design and lazier loading policy.

We present an operational semantics for an abstraction of the JVM and prove a type safety result. The operational semantics differs from usual ones (e.g. [1]) that just specify state transitions for the JVM bytecode instructions. Our semantics includes transitions for "macro operations," e.g. resolution and class loading, that are performed by the "core" of the JVM. The core of the JVM maintains data structures, such as a loaded class cache, which are updated by these macro operations. The interaction between the core of the machine and user code is complex: there is a mutual recursion between user code executing in a thread and the core machine. The state transition formalism we use reflects this with the use of "nested" rules.

We start by introducing the formal entities (sets, operations, etc.) used in our formalization. We then define states and transitions for a state machine representing an abstraction of the JVM. Transitions fire when certain conditions are satisfied. Some conditions correspond to runtime checks actually performed by the executing JVM; their failure raises exceptions in the JVM. Other conditions serve to ensure type-safe operations in our formalization; there are no corresponding runtime checks in the JVM. If these type safety conditions are not satisfied, then the behavior of the JVM is undefined[4]. We finally introduce a notion of *validity* of states, and present theorems stating that transitions from valid states lead to valid states, and that transitions from valid states are always possible unless either there is no more

[4] In practice, the behavior of an actual JVM when such conditions are not satisfied is determined by the implementation of the machine. Knowledge of the implementation can be maliciously exploited to compromise the security of the JVM, if type safety can be circumvented. For example, typical implementations access fields through offsets added to the address of an object. In such implementations, a type-unsafe operation could access arbitrary data within objects, regardless of field layout (see example in the previous section).

```
public class C {               // Class C with defining loader l1.
  public void m() {
    (new D()).n(new S());
  }
}
public class D {               // Class D with defining loader l1.
  public void n(T t) {
    Object o = t.f;
  }
}
public class T {               // Class T1 with defining loader l1.
  Object f;
}
public class T {               // Class T2 with defining loader l2.
  int f;
}
public class S extends T {     // Class S with defining loader l2.
}
```

Figure 2: Another example

code to execute or a condition corresponding to an actual check in the JVM is violated. In other words, in a valid state, the failure of a type safety condition alone never causes the machine to halt. This means that if the JVM starts in a state corresponding to a valid state in our formalization, each operation in the JVM will be always type-safe, and no checks corresponding to the type safety conditions are needed.

Our formalization makes many simplifications to the JVM. Most important is that we ignore concurrency and exceptions. One could easily imagine a concurrency bug leading to inconsistent data structures; such bugs are not addressed by our specification. The JVM, when presented with a program that is inconsistent, throws an exception to report the error. For example, if a loading constraint is violated, a linking exception is thrown. Since we do not model exceptions, in our formalization the machine simply halts. We also do not treat static fields or methods, primitive types, interfaces, arrays, object initialization, subroutines, and field or method modifiers. We believe these features are orthogonal to the issues raised by class loading. We treat a few bytecode instructions, namely those to access fields, call methods, return results, create new objects, and load/store from/to local variables. In a strong sense the essence of the Sun approach is captured by our formalization.

Our formalization includes bytecode verification. Bytecode verification assigns class names to memory locations (in the operand stack and for all local variables) at each program point of a method based on the instructions in the method, as shown in Section 2.5. For example, a `getfield` instruction requires a certain class name (for the target object) to be assigned to the top of the stack at the program point where the instruction is, and requires another class name (for the field value) to be assigned at the top of the stack at the next program point.

In our formalization the state of an executing JVM contains a global state consisting of a loaded class cache, a set of loading and subtype constraints and a heap for storing objects. In addition, the state includes a component that models the execution stack of a single thread (as per the restriction above) storing frames. Each frame is a tuple consisting of a class, a method of the class, a program point within the method, and a state of the local memory (the operand stack and all local variables).

We identify a subset of execution states called *valid states* that satisfy certain constraints. Many of the constraints are straightforward. For example, one of such constraints requires that all classes in a frame be in (the range of) the loaded class cache. However, the key constraint is the *conformance* condition, which relies on loading and subtype constraints. The problem is that some classes may never get loaded or only get loaded at run time and that the bytecode verifier assigns only type names, not full type information, to memory locations, to avoid premature loading. Loading and subtype constraints are used to state requirements for not-yet-loaded classes denoted by class names. The conformance relation takes into account not only loaded classes, but also loading and subtype constraints.

To further explain the above point, consider again the example in Figure 1. The type of the object passed to the `n(T t)` method is T_1. Until name `T` is resolved from D, the formal parameter `t` of method `n(T t)` has no full type assigned to it (only name `T`). Therefore, we cannot state that the passed object matches the type of the formal parameter, simply because there is no full type information. However, upon resolution of method `n(T t)` from C, the loading constraint $\langle l_1, l_2, \mathtt{T}\rangle$ is introduced. Such a constraint requires equality of the class loaded by l_1 for name `T` (i.e., T_1) and the class that will be possibly loaded by l_2 for the same name `T`. By taking this constraint into account, the conformance relation captures the fact that the already loaded type T_1 "matches" the not-yet-loaded type for the formal parameter of the method. If T_2 were never loaded (e.g., if the assignment `i = t.f` were not present in the method), the constraint would never be violated and the conformance relation would still hold.

Let us now explain subtype constraints. Suppose that, during bytecode verification, a class name cn is required (e.g., by a **getfield**$(cn, \ldots)$ instruction) at the top of the stack, where a name cn', with $cn' \neq cn$, has instead been assigned. The field access is correct as long as cn' is a subclass of cn. More precisely, since cn and cn' are just names, the requirement is that if and when the loading requests $\langle l, cn' \rangle$ and $\langle l, cn \rangle$, where l is the defining loader of the class whose method is being verified, yield two loaded classes C' and C (respectively), then C' must be a subclass of C. According to Sun's specification and implementation, and as described in the previous section for the example in Figure 2, the bytecode verifier checks this subtype relation eagerly, by resolving names cn and cn'.

In our formalization, we check subtype relations lazily: the bytecode verifier just posts subtype constraints of the form $\langle l, cn, cn' \rangle$. Such a constraint expresses exactly the requirement that if and when the loading requests $\langle l, cn' \rangle$ and $\langle l, cn \rangle$ yield two classes C' and C, then C' must be a subclass of C. These subtype constraints are handled analogously to the loading constraints. Each time a class is loaded, subtype constraints are checked for violation. Each time a new subtype constraint is introduced, it is checked for violation, too. In other words, loading constraints, subtype constraints, and the loaded class cache are constantly maintained in a mutually consistent state. The primary advantage of this approach is lazier loading, because no class needs to be loaded for verification purposes. Another advantage is that the interaction between the bytecode verifier and the rest of the core JVM is simpler and clearer: the verifier is in fact just a functional component that takes a class as argument and returns a yes/no answer plus a set of subtype constraints as result.

To further illustrate subtype constraints, let us consider the example in Figure 2 using subtype constraints. Again, we start with only class C loaded. Before method `m()` is executed, it must be verified. Bytecode verification successfully verifies the method and posts the subtype constraint $\langle l_1, \mathtt{S}, \mathtt{T} \rangle$ (without loading any class). Method `m()` starts executing. Classes D and S are loaded. Since T_1 has not been loaded yet, the subtype constraint $\langle l_1, \mathtt{S}, \mathtt{T} \rangle$ is not violated yet. The newly created S object is passed as an argument to method `n(T t)` invoked upon the newly created D object. When the field access `t.f` is about to be executed, class name `T` is now resolved with l_1 (which is the defining loader of D) as initiating loader. This yields class T_1; since it is not a superclass of S, now the subtype constraint $\langle l_1, \mathtt{S}, \mathtt{T} \rangle$ is violated. Therefore, an exception is thrown and field access is prevented.

4. FORMALIZATION

A full presentation of our formalization and its properties would exceed the space allowed to this paper. Therefore, in this section we just give highlights, making some simplifications for the sake of brevity. A full presentation can be found in [19].

4.1 Transition systems

A transition system is a pair $\langle X, \tau \rangle$ where X is a set of states and $\tau \subseteq X \times X$ is a transition relation between states ($x \tau x'$ means that from state x we can move to state x'). In our formalization we make use of a family of labeled transition systems $\langle \mathit{GStt}, \overset{lab}{\Longrightarrow} \rangle$, where GStt formalizes all (execution) states in the "core" of the JVM, and lab indicates the current "macro operation" (as mentioned in Section 3). Furthermore, we make use of an unlabeled transition system $\langle \mathit{Stt}, \Longrightarrow \rangle$, where Stt formalizes all (execution) states in the JVM. In substance, the labeled transition systems capture activities taking place in the core of the machine, while the unlabeled one captures the execution of JVM instructions.

The state sets GStt and Stt are incrementally built starting from abstract sets whose elements represent the entities present in the JVM. In particular, we have sets O, C, L, and M consisting of (all possible) objects, classes, loaders, and methods. These sets are abstract in the sense that we do not define their exact structure, but just postulate the existence of functions operating on them which capture the properties we are interested in. This approach of using abstract sets makes the formalization simple and general.

Four of those functions operating on the abstract sets are as follows:

$$\begin{aligned} &cl : O \to C, \\ &cd : M \to I^+, \\ &sup : C \to \mathit{CN} \uplus \{\mathsf{nil}\}, \\ &ld : C \to L. \end{aligned}$$

The function cl returns the class of a given object. The function cd returns the code of a given method, where the set I formalizes all instructions we are interested in, and the code consists of a finite non-empty sequence of such instructions. The function sup gives the name of the superclass of a given class, where the set CN consists of all class names, and nil indicates that the given class is the system class Obj, and thus has no superclass. The function ld yields the defining loader of a given class.

Formally, the state sets GStt and Stt are defined as follows:

$$\mathit{GStt} = \mathit{Hp} \times \mathit{LC} \times \mathit{Ct} \qquad \text{and} \qquad \mathit{Stt} = \mathit{Stk} \times \mathit{GStt}.$$

The set Hp formalizes heaps (where objects are stored).

The set LC formalizes *loaded class caches*:

$$\mathit{LC} = L \times \mathit{CN} \xrightarrow{\mathrm{f}} C.$$

A loaded class cache is a finite map from loading requests (i.e. pairs of loaders and class names) to classes.

Given $lc \in \mathit{LC}$, we define the *subclass* relation determined by all loaded classes as the least relation $\preceq_{lc} : C \times C \to \mathbf{B}$ satisfying

$$\begin{aligned} &c \preceq_{lc} c' \;\Leftrightarrow \\ &\quad c = c' \;\vee \\ &\quad (\langle ld(c), sup(c) \rangle \in \mathcal{D}om(lc) \;\wedge\; lc(ld(c), sup(c)) \preceq_{lc} c'). \end{aligned}$$

In the definition, the loading request $\langle ld(c), sup(c) \rangle$ means that the defining loader of the class c is used as an initiating loader to load the direct superclass of the class c. The condition $\langle ld(c), sup(c)\rangle \in \mathcal{D}om(lc)$ means that the loaded class cache lc records that the direct superclass of the class c has been loaded. Note that if $\langle \cdots, sup(c)\rangle \in \mathcal{D}om(lc)$, then $sup(c) \neq \mathsf{nil}$, since $\mathsf{nil} \notin CN$.

The set Ct formalizes *loading* and *subtype constraints*:

$$Ct = \mathcal{P}_\omega((L \times L \times CN) \cup (L \times CN \times CN)).$$

A loading constraint is a triple $\langle l, l', cn\rangle \in L \times L \times CN$, expressing that if loaders l and l' load classes for class name cn, then loaded classes must be the same. A subtype constraint is a triple $\langle l, cn, cn'\rangle \in L \times CN \times CN$, expressing that if loader l loads a class for class name cn, and another class for class name cn', then the loaded class for class name cn must be a subclass of loaded class for class name cn'.

A state in $GStt$ is called a *global* state. A state in Stt consists of a global state plus a *call stack*, which is a finite sequence of *frames*:

$$Stk = Frm^* \qquad \text{where} \qquad Frm = C \times M \times \mathbf{N} \times OS \times LV.$$

A frame is a tuple $\langle c, m, p, os, lv\rangle$, where c denotes a current class c, m denotes a current method in the class c, p is a natural number denoting the program point of the instruction (in the method code $cd(m)$) about to be executed, and os and lv constitute the current contents of the operand stack and local variables:

$$OS = V^* \quad \text{and} \quad LV = V^* \quad \text{where} \quad V = O \cup \{\mathsf{null}\}.$$

Transitions are defined by means of rules such as those shown in Figure 3. Each rule contains premises and a conclusion (respectively above and below the line), and expresses that if all premises hold, then the conclusion holds. Conclusions assert transitions between states. The labeled and unlabeled transition relations are defined as the smallest relations that satisfy all the rules. For brevity, not all the rules are shown in Figure 3, and not all the functions used in the rules are formally defined in this paper.

Rule (IV) formalizes method invocation. Its first premise requires that the instruction $cd(m)|_p$ at program point p is an `invokevirtual` instruction, where mn is a method name, cn and cn_j $(0 \leq j \leq n)$ are class names, and $n \geq 0$. The second premise requires that the method is resolved: method resolution is captured by a labeled transition that updates the global state. The third premise is a type-safety condition, but does not correspond to any runtime checks in the JVM (see Section 3). The type-safety condition requires that the class of the object o upon which the method is invoked, is a subclass of the class resulting from resolving name cn from the defining loader $ld(c)$ of the current class c. Note that the method resolution step in the second premise ensures that the loaded class cache lc' contains a loaded class for the loading request $\langle ld(c), cn\rangle$ (see below). The last premise serves to select the method to be executed based on the class of the target object o (i.e., dynamic dispatch): $mthSel_{lc'}(cl(o), \ldots)$ returns the closest class above $cl(o)$ in the hierarchy (possibly $cl(o)$ itself, otherwise a superclass) that contains a method with the given descriptor – the method is also returned. Such a method will always exist because of the previous two premises. The conclusion of the rule expresses that a new frame is added to the call stack. Its operand stack is initially empty, while its first $n+1$ local variables are initialized to the target object o of method invocation and the actual arguments $v_1, \ldots, v_n$. The other variables are initialized to null for simplicity. The notation $nv(m')$ denotes the number of local variables used in m', and $[a]^k$ a sequence containing k occurrences of a.

Rule (RM) formalizes method resolution. Its first premise requires class name cn to be resolved, while the second premise ensures there is a method with the required descriptor in the resolved class or one of its superclasses. The third premise requires that the extended set of constraints is consistent w.r.t. the loaded class cache, i.e., that no constraints are violated. Note that the predicate $css_{lc'}$ checks consistency of the given set of constraints w.r.t. lc' (see below). The conclusion asserts that the set of constraints is extended with loading constraints for the class names appearing in the method descriptor (as described in [15]).

Rules (RC1) and (RC2) formalize class resolution. The former expresses that if the loaded class cache already contains an entry for the defining loader $ld(c)$ of c and name cn (i.e., a loading request $\langle ld(c), cn\rangle$) in its domain $\mathcal{D}om(lc)$ then no loading is necessary and the global state does not change. If that is not the case (first premise of rule (RC2)), then the class must be loaded, where the defining loader $ld(c)$ of the class c is used as the initiating loader, and the method `loadClass` is selected for the class of the loader $ld(c)$, as expressed by the second premise. The third and fourth premises ensure that the invocation of `loadClass` eventually terminates with a class at the top of the operand stack – the result of loading. (We use $\Longrightarrow^*$ to denote zero or more transition steps.) The function str returns a string (i.e., an object of system class `String`) corresponding to the given class name. The conclusion asserts that the loaded class cache is updated with a new entry, provided that the new entry does not lead to the violation of any constraints (as required by the last premise of the rule). During execution of `loadClass` other methods can be invoked, and therefore other methods and classes can be resolved. The rules capture the mutual recursion between execution of user's code and activities carried out by the core of the JVM.

4.2 Bytecode verification and subtype constraints

Verifying a method m of a class c amounts to assigning two sequences of type names to each program point in the code $cd(m)$. The two sequences constitute the names of the types of the values in the operand stack and local variables at that program point, for all possible executions of the code. The assigned type names must be "consistent" with the instructions: an instruction at a program point may require that certain type names be assigned to certain positions of the operand stack and certain local variables at the current and other related program points. For example, if $cd(m)|_p = \mathsf{invokevirtual}(cn, mn, [cn_1, \ldots, cn_n], cn_0)$, then the type names assigned to the top $n+1$ stack positions at program point p must denote subclasses of the classes denoted by class names $cn, cn_1, \ldots, cn_n$, respectively. At program point $p+1$, those top $n+1$ positions must be replaced by one with class name cn_0, while the type names for local variables are unmodified. See also Section 2.5.

$$\frac{\begin{array}{l} cd(m)|_p = \mathsf{invokevirtual}(cn, mn, [cn_1, \ldots, cn_n], cn_0) \\ \langle hp, lc, ct\rangle \overset{\mathrm{RM}(c,cn,mn,[cn_1,\ldots,cn_n],cn_0)}{\Longrightarrow} \langle hp', lc', ct'\rangle \\ cl(o) \preceq_{lc'} lc'(ld(c), cn) \\ mthSel_{lc'}(cl(o), mn, [cn_1, \ldots, cn_n], cn_0) = \langle c', m'\rangle \end{array}}{\begin{array}{l} \langle stk + \langle c, m, p, os + o + [v_1, \ldots, v_n], lv\rangle, hp, lc, ct\rangle \Longrightarrow \\ \langle stk + \langle c, m, p+1, os, lv\rangle + \langle c', m', 0, [\,], [o, v_1, \ldots, v_n] + [\mathsf{null}]^{nv(m')-n-1}\rangle, hp', lc', ct'\rangle \end{array}} \tag{IV}$$

$$\frac{\begin{array}{l} \langle hp, lc, ct\rangle \overset{\mathrm{RC}(c,cn)}{\Longrightarrow} \langle hp', lc', ct'\rangle \\ mthSel_{lc'}(lc'(ld(c), cn), mn, [cn_1, \ldots, cn_n], cn_0) = \langle c', m\rangle \\ css_{lc'}(ct' \cup \{\langle ld(c), ld(c'), cn_j\rangle \mid 0 \le j \le n\}) \end{array}}{\langle hp, lc, ct\rangle \overset{\mathrm{RM}(c,cn,mn,[cn_1,\ldots,cn_n],cn_0)}{\Longrightarrow} \langle hp', lc', ct' \cup \{\langle ld(c), ld(c'), cn_j\rangle \mid 0 \le j \le n\}\rangle} \tag{RM}$$

$$\frac{\langle ld(c), cn\rangle \in \mathcal{D}om(lc)}{\langle hp, lc, ct\rangle \overset{\mathrm{RC}(c,cn)}{\Longrightarrow} \langle hp, lc, ct\rangle} \tag{RC1}$$

$$\frac{\begin{array}{l} \langle ld(c), cn\rangle \notin \mathcal{D}om(lc) \\ mthSel_{lc}(cl(ld(c)), \mathtt{loadClass}, [\mathtt{String}], \mathtt{Class}) = \langle c, m\rangle \\ \langle [\langle c, m, 0, [\,], [ld(c), str(cn)] + [\mathsf{null}]^{nv(m)-2}\rangle], hp, lc, ct\rangle \Longrightarrow^* \\ \quad \langle [\langle c, m, p, os + c', lv\rangle], hp', lc', ct'\rangle \\ cd(m)|_p = \mathsf{areturn} \\ css_{lc'\{\langle ld(c), cn\rangle \mapsto c'\}}(ct') \end{array}}{\langle hp, lc, ct\rangle \overset{\mathrm{RC}(c,cn)}{\Longrightarrow} \langle hp', lc'\{\langle ld(c), cn\rangle \mapsto c'\}, ct'\rangle} \tag{RC2}$$

Figure 3: Some transition rules

Within method code, classes are referenced by name only. In order to verify if a class named cn is a subclass of a class named cn', the bytecode verification of the current JVM [16] resolves cn and cn' (using the defining loader $ld(c)$ of the current class), and checks the subtype relation. In general, if there is a branch in the control flow of the program, then at the branch's target that is reachable from more than one preceding program point, the types denoted by the assigned names must be obtained by "merging" the types denoted by the assigned names at the preceding program points[5]. The current JVM resolves the relevant class names assigned at the preceding program points, and if successful, all resulting loaded classes are merged. The behavior just described implies that classes that may not be used in execution, get loaded during verification.

In our formalization, bytecode verification never resolves (or loads) any classes, thus avoiding premature loading. In order to deal with the "merging" operation, finite sets of type names are assigned by bytecode verification, and single class names cn are represented as singleton sets $\{cn\}$ [5, 11, 18, 4]. In that case, the merging operation is simply set union. In order to ensure that the class denoted by a set $\{cn_1, \ldots, cn_n\}$ is a subclass of the class denoted by a class name cn, bytecode verification produces subtype constraints $\langle ld(c), cn_j, cn\rangle$ for $1 \le j \le n$, where class c is the current class. These subtype constraints will then be added to the global state of the executing JVM, and checked only when the relevant classes are loaded.

In our formalization, bytecode verification is captured by a function

$$bcv : C \times M \to (StTy \times SR) \cup \{fail\}$$

where

$$\begin{array}{l} StTy = ((\mathcal{P}_\omega(CN))^* \times (\mathcal{P}_\omega(CN))^*)^* \text{ and} \\ SR = \mathcal{P}_\omega(CN \times CN). \end{array}$$

The result of this function is determined by solving a constraint problem over semilattices as in e.g. [5, 18, 17]. If the problem has no solution, *fail* is returned. Otherwise, the solution contains a static type assignment (i.e., an element of $StTy$) plus a finite set of *subtype requirements* (i.e., pairs of class names). A static type assignment is a finite sequence of pairs. Each pair consists of two finite sequences of finite sets of class names: it gives sets of class names (as discussed above) for operand stack and local variables at a program point.

In our transition system, bytecode verification takes place when a new class c is introduced into the state of the JVM (as formalized by a rule not shown here). Each subtype requirement $\langle cn, cn'\rangle$ returned by bcv (if successful) is added

[5] "Merging" two classes means to find their first common superclass.

as a subtype constraint $\langle ld(c), cn, cn'\rangle$ to the global state of the machine, provided the resulting set of loading and subtype constraints satisfies the predicate css_{lc} with respect to the current loaded class cache lc.

As mentioned before, the predicate $css_{lc}(ct)$ checks whether any loading or subtype constraint in the set ct is violated with respect to the current loaded class cache lc. Concretely, we define for each $ct \in Ct$ an equivalence relation $\simeq_{ct}$ induced by all loading constraints in ct as the smallest equivalence relation over $(L \times CN) \times (L \times CN)$ satisfying

$$\langle l, l', cn\rangle \in ct \;\Rightarrow\; \langle l, cn\rangle \;\simeq_{ct}\; \langle l', cn\rangle,$$

and a subtype relation $\preceq_{ct}$ induced by all subtype constraints in ct as the smallest transitive relation over $(L \times CN) \times (L \times CN)$ satisfying

$$\begin{array}{l} \langle l, cn\rangle \;\simeq_{ct}\; \langle l', cn'\rangle \Rightarrow \langle l, cn\rangle \preceq_{ct} \langle l', cn'\rangle \text{ and} \\ \langle l, cn, cn'\rangle \in ct \;\Rightarrow\; \langle l, cn\rangle \preceq_{ct} \langle l, cn'\rangle. \end{array}$$

Then the predicate $css_{lc}(ct)$ actually checks whether the equivalence relation $\simeq_{ct}$ coincides with the identity relation with respect to the loaded class cache lc, and whether the subtype relation $\preceq_{ct}$, modulo the equivalence relation $\simeq_{ct}$, coincides with the subclass relation $\preceq_{lc}$ induced by the loaded class cache lc. For example, for $\langle l, l', cn\rangle$, $\langle l', cn, cn'\rangle$, $\langle l', l'', cn'\rangle \in ct$, if $lc(l, cn) \not\preceq_{lc} lc(l'', cn')$, then $css_{lc}(ct)$ is false, even when $\langle l', cn\rangle, \langle l', cn'\rangle \notin \mathcal{D}om(lc)$, i.e., even when no classes have been loaded for the loading requests $\langle l', cn\rangle$ and $\langle l', cn'\rangle$.

4.3 Type safety

As mentioned in Section 3, our transition systems formalize a defensive JVM, i.e., one that ensures type safety by checking conditions during execution. Our main theorems state that validity is preserved by state transitions, and that from a valid state it is always possible to move to another valid state, unless either there is no more code to execute or some of the listed conditions that correspond to runtime checks in the actual JVM fail.

Let us consider, for example, rule (IV) above. The second premise in the rule requires method resolution to succeed. This condition corresponds to a check also performed by the actual JVM – in fact, an exception is thrown if method resolution fails. The third premise, however, is a type-safety check that the real JVM does not perform – in typical implementations it just accesses a method table at the offset determined by resolution. Our theorem proves that if our defensive machine halts, the reason may be that method resolution fails, but never that the type safety check fails. This means that the type safety check is unnecessary, and that the real JVM will always invoke methods in a way that is consistent with the type of the invocation targets. Note that since we do not consider exceptions, in the event that method resolution fails our machine just halts. We could add exceptions to our model without significant conceptual modifications.

The notion of *validity* of states captures a well-formedness and a *(type) conformance* property. Validity is proved to be preserved by all transitions: any transition from a valid state leads to a valid state. Given that the machine starts in a valid state (which is true for any reasonable initial state), validity is an invariant of the execution.

The well-formedness property includes simple requirements, e.g. that the class $cl(o)$ of each object o in the heap must be in (the range of) the loaded class cache (i.e., $cl(o) \in \mathcal{R}(lc)$), and some consistency relations, e.g. the relation $css_{lc}(ct)$ for the current loaded class cache lc and set of loading and subtype constraints ct.

The conformance property is more complex, since it depends on a subtype relation $\preceq_{lc,ct}$ defined by all loaded classes in lc and all loading and subtype constraints in ct on not-yet-loaded classes. Formally, the relation $\preceq_{lc,ct}$, for each $lc \in LC$ and $ct \in Ct$, is defined as the smallest transitive relation satisfying

$$\begin{array}{l} \langle l, cn\rangle \preceq_{ct} \langle l', cn'\rangle \;\Rightarrow\; \langle l, cn\rangle \preceq_{lc,ct} \langle l', cn'\rangle \text{ and} \\ \langle l, cn\rangle, \langle l', cn'\rangle \in \mathcal{D}om(lc) \;\wedge\; lc(l, cn) \preceq_{lc} lc(l', cn') \;\Rightarrow \\ \qquad \langle l, cn\rangle \preceq_{lc,ct} \langle l', cn'\rangle. \end{array}$$

The key idea behind $\preceq_{lc,ct}$ is that it relates loading requests, instead of loaded classes.

At any time in execution, the loaded class cache may or may not contain a loaded class for a given loading request. The relation $\preceq_{lc,ct}$ on loading requests considers not only the actual subclass relation $\preceq_{lc}$, but also the constraints through $\preceq_{ct}$. So, even if some classes are not (yet) loaded, requirements for such (future) classes are captured by suitable constraints.

Note that $\preceq_{lc}$, $\preceq_{ct}$ and $\preceq_{lc,ct}$ are distinct relations.

The conformance property is captured by a *conformance* relation $cfm_{lc,ct}$ over $V \times L \times CN$, defined for each $lc \in LC$ and $ct \in Ct$ as

$$\begin{array}{l} cfm_{lc,ct}(v, l, cn) \;\Leftrightarrow \\ (v \neq \mathsf{null} \;\Rightarrow\; \langle ld(cl(v)), nm(cl(v))\rangle \preceq_{lc,ct} \langle l, cn\rangle), \end{array}$$

This relation specifies the notion of a value v "conforming" to the class for a loading request $\langle l, cn\rangle$ in case that class is loaded. If the value is null, then the relation is (trivially) satisfied (because null is a valid element of every type). Otherwise, the defining loader $ld(cl(v))$ and name $nm(cl(v))$ of the class of the value (object) are computed, and the relation

$$\langle ld(cl(v)), nm(cl(v))\rangle \preceq_{lc,ct} \langle l, cn\rangle$$

must hold. It means that if $\langle l, cn\rangle \in \mathcal{D}om(lc)$, then

$$lc(ld(cl(v)), nm(cl(v))) \preceq_{lc} lc(l, cn)$$

The subtle point here is the condition $\langle l, cn\rangle \in \mathcal{D}om(lc)$. In other words, if no class has been loaded for the loading request $\langle l, cn\rangle$, the value v can be of any class.

5. RELATED WORK

The requirement that loaders in addition to class names are needed to uniquely identify class objects is mentioned in the Java language specification [13]. But it was not completely understood what the concrete mechanism should be. The bugs reported by Saraswat [20], Tozawa and Hagiya [21] and ourselves [4] are evidence of this. Saraswat proposed a solution where method overriding is based on full types instead of names only. However his solution may cause counter-intuitive dynamic dispatch and requires a modification to the class loaders' API to avoid premature class loading.

Dean [6] presented probably the first formal model for Java class loading, focusing on the static typing, using the PVS verification system [9]. The model is clean and abstract, but does not consider loading and subtype constraints.

Jensen, LeMetayer and Thorn [14] proposed an abstract formalization of Java class loading, and showed how the Saraswat bug is disallowed by the formalization. Their formalization pre-dates Sun's response to the Saraswat bug and differs in some aspects from the official semantics of the JVM [2].

Goldberg [11] formalized a way to integrate some aspects of class loading into bytecode verification. The idea is that the bytecode verifier does not load classes to insure type safety, but generates constraints that are checked when the referenced classes are loaded. He did not consider multiple loaders. Our approach includes and generalizes this idea.

Liang and Bracha [15] introduced the concepts of loading constraints and loaded class cache. Their setting is informal, and thus it is hard to verify their claims, or see where the problems are and how they are solved. This solution is also documented in [16]. A good informal account of class loading can be found in [12]. However, that account often does not distinguish between the JDK 1.2 implementation and the JVM specification.

Fong and Cameron [8] proposed a general, modular architecture for mobile-code loading and verification, and discussed a possible instantiation for Java loading and bytecode verification. In particular, their concept of proof obligations corresponds to our concepts of constraints. However, they do not consider multiple loaders.

Börger and Schulte [1] presented a fairly complete operational semantics of the JVM which includes loading. The work follows the well-established approach of Abstract State Machines and can take advantage of existing machine-supported simulation tools. Their model of loading considers exceptions, but does not consider the loaded class cache and loading and subtype constraints.

Tozawa and Hagiya's formal model for a type safe JVM [22] is closely related to ours, although their model and ours were developed independently. Some of the components considered in their model are also considered in ours. For example, their loaded class caches and loading constraints are very similar to ours, and their concept of widening conversion roughly corresponds to our concept of subtype constraints. There are two main conceptual differences between their model and ours. The first is that they do not consider subtype constraints: their widening conversion is checked at verification time, which means that the involved classes must be loaded. This corresponds to Sun's eager loading strategy for verification. The second is that they do not consider execution of code in user-defined loaders. Furthermore, they take a different overall approach. They define an operational semantics that depends on an "environment", which consists of loaded classes, objects in the heap, etc. They define judgements (such as widening conversion) holding in an environment, and they prove some monotonicity properties for judgements w.r.t. extending environments (e.g., as per loading of new classes). In our approach, instead, the information about loaded classes, heap, etc. is part of the state, and there are explicit state transitions that load new classes, create new objects, and so on. Therefore, we believe our model is more intuitive and closer to the JVM specification and implementations.

Drossopoulou [7] developed a model for Java class loading, focusing at an abstract level on the interface between execution, loading and verification. Her work introduces and analyzes a high-level language, which is closer to Java than the JVM. Her model does not consider multiple loaders, while ours does, but her work handles exceptions, while ours does not.

Type-safe loading has also been studied in static settings both for high-level modules (e.g. [3]) and for assembly languages (e.g. [10]). In these settings, the major difference between the dynamic and static loading is that the final steps of loading and the formation of the "real" executable are delayed until loading-time [10]. The mechanism for dynamic loading in Java is more complex, since it is possible that a class that is used in the static type inference never gets loaded, and that a loader itself is a user-defined program and needs to be loaded at run time.

6. CONCLUSION

The mechanisms for Java class loading and bytecode verification are complex. It is hard to ascertain properties such as type safety using an informal specification. We have presented a formalization of class loading and formally proved type safety.

Our formalization addresses most key issues mentioned in [13, 16, 15] and in Sun's current implementation. In addition, we have introduced subtype constraints so that the bytecode verifier does not need to resolve any class, thus avoiding premature loading and having a cleaner interaction with the rest of the machine.

Our formalization models a simplified JVM. It includes essential internal data structures such as the loaded class cache, loading and subtype constraints. It supports selected language features such as classes, objects, methods, and dynamic and lazy loading, but not interfaces, arrays, primitive types, access control, exception handling, garbage collection and multi-threading. So far we have not seen any fundamental problems to extending our formalization to address the missing data structures and features.

Acknowledgements

The research has been partially supported by DARPA contracts F30602-96-C0363 and F30602-99-C-0091. We sincerely thank Gilad Bracha for numerous discussions, clarifications and comments on early versions of the paper. We also thank Sheng Liang for several useful discussions, and Stephen Fitzpatrick for comments on the paper.

7. REFERENCES

[1] E. Börger and W. Schulte. Modular design for the Java virtual machine architecture. `ftp://ftp.di.unipi.it/pub/Papers/boerger/jvmarch.ps`, 1999.

[2] G. Bracha. A critique of 'Security and dynamic loading in Java: A formalisation'. `http://java.sun.com/people/gbracha/critique-jmt.html`, 1999.

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

[4] A. Coglio and A. Goldberg. Type safety in the JVM: Some problems in JDK 1.2.2 and proposed solutions. In *Proc. ECOOP Workshop on Formal Techniques for Java Programs*, 2000. Long version available at `http://www.kestrel.edu/java`.

[5] A. Coglio, A. Goldberg, and Z. Qian. Towards a provably-correct implementation of the JVM bytecode verifier. In *Proc. OOPSLA'98 Workshop Formal Underpinnings of Java*, 1998.

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

[7] S. Drossopoulou. Towards an abstract model of Java dynamic linking and verification. Department of Computing, Imperial College, London, UK.

[8] P. Fong and R. Cameron. Proof linking: An architecture for modular verification of dynamically-linked mobile code. In *Proc. 6th ACM SIGSOFT Int. Symp. on the Foundations of Software Engineering (FSE'98)*, 1998.

[9] Formal Methods Program - SRI Computer Science Laboratory. The PVS specification and verification system. `http://pvs.csl.sri.com/`, 1999.

[10] N. Glew and G. Morrisett. Type-safe linking and modular assembly language. In *Proc. 26th ACM Symp. Principles of Programming Languages*, 1999.

[11] A. Goldberg. A specification of Java loading and bytecode verification. In *Proc. 5th ACM Conference on Computer and Communications Security*, 1998.

[12] L. Gong. *Inside Java 2 Platform Security: Architecture, API Design, and Implementation.* Addison-Wesley, 1999.

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

[14] T. Jensen, D. LeMetayer, and T. Thorn. Security and dynamic class loading in Java: a formalisation. In *Proc. IEEE Int. Conference on Computer Languages*, 1998.

[15] S. Liang and G. Bracha. Dynamic class loading in the Java™ virtual machine. In *Proc. Conf. on Object-Oriented Programming, Systems, Languages, and Applications*, pages 36–44. ACM Press, 1998.

[16] T. Lindholm and F. Yellin. *The Java™ Virtual Machine Specification - 2nd edition.* Addison-Wesley, 1999.

[17] Z. Qian. Standard fixpoint iteration for Java bytecode verification. *ACM TOPLAS.* To appear.

[18] Z. Qian. A formal specification of Java^{tm} virtual machine instructions for objects, methods and subroutines. In J. Alves-Foss, editor, *Formal Syntax and Semantics of Java™.* Springer Verlag LNCS 1523, 1998.

[19] Z. Qian, A. Goldberg, and A. Coglio. A formal specification of Java™ class loading. Long version. `http://www.kestrel.edu/java`, 2000.

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

[21] A. Tozawa and M. Hagiya. Careful analysis of type spoofing. In *JIT'99 Java-Informations-Tage 1999, Clemens H. Cap, Hrsg., Informatik aktuell*, pages 290–296. Springer Verlag, 1999.

[22] A. Tozawa and M. Hagiya. New formalization of the JVM. `http://nicosia.is.s.u-tokyo.ac.jp/members/miles/papers/cl-99.ps`, 1999.