Proof Linking: Modular Verification of Mobile Programs in the Presence of Lazy, Dynamic Linking

Philip W. L. Fong and Robert D. Cameron
School of Computing Science
Simon Fraser University, B.C., Canada
{pwfong, cameron}@cs.sfu.ca

Although mobile code systems typically employ link-time code verifiers to protect host computers from potentially malicious code, implementation flaws in the verifiers may still leave the host system vulnerable to attack. Compounding the inherent complexity of the verification algorithms themselves, the need to support lazy, dynamic linking in mobile code systems typically leads to architectures that exhibit strong interdependencies between the loader, the verifier and the linker. To simplify verifier construction and provide improved assurances of verifier integrity, we propose a modular architecture based on the concept of proof linking. This architecture encapsulates the verification process and removes dependencies between the loader, the verifier, and the linker. We also formally model the process of proof linking and establish properties to which correct implementations must conform. As an example, we instantiate our architecture for the problem of Java bytecode verification and assess the correctness of this instantiation. Finally, we briefly discuss alternative mobile code verification architectures enabled by the proof linking concept.

Categories and Subject Descriptors: D.2.4 [**Software Engineering**]: Software/Program Verification—*Correctness proofs, Formal methods*; D.2.11 [**Software Engineering**]: Software Architectures—*Domain-Specific architectures*; D.3.4 [**Programming Languages**]: Processors—*Run-time environments*

General Terms: Security, Verification, Languages, Design

Additional Key Words and Phrases: Correctness conditions, dynamic linking, Java, mobile code, modularity, proof linking, safety, verification protocol, virtual machine architecture

1. INTRODUCTION

Recent years have witnessed a significant growth of interest in mobile code, particularly in the form of active contents (web-browser applets) and code-on-demand [Carzaniga et al. 1997]. A key factor in this growth has been the development of suitable security models for the protection of host computer systems against the potential dangers of executing unknown code. In particular, the Java programming language [Gosling et al. 1996] and its associated support technology have achieved considerable success through a strong security model implemented within the Java Virtual Machine (JVM). As Java bytecode is downloaded from an untrusted origin, the JVM subjects it to a verification step [Lindholm and Yellin 1999, Chapter 5] in order to ensure that it cannot affect the host machine in an undesirable way. A major function of the verification procedure is to ensure that untrusted mobile code units are type safe. As some authors have pointed out, Java's access control mechanism, namely, the security manager, is protected by the type system [Dean et al. 1996]. As long as downloaded bytecode obeys the typing rules of Java, the security manager should be tamperproof.

Relying on a link-time verifier to protect a host computer system has the problem that the verifier itself may be flawed. If so, designers of malicious code may well be able to exploit the flaw to bypass security checks. In fact, several security breaches have been discovered in major Java implementations [Kimera Team ; Princeton Secure Internet Programming Group ; JavaSoft]. These flaws may be attributed, in part, to the inherent complexity of bytecode verification, involving both data flow analysis and type checking.

Additional complexity in verifier implementation may arise through the combination of verification in an incremental process with lazy, dynamic linking. This complexity become manifest in two problematic architectural features of Sun's JVM:

The research was funded in part by a scholarship and an operating grant from the Natural Sciences and Engineering Research Council of Canada.
An earlier version of this paper appeared in the *Proceedings of the ACM SIGSOFT Sixth International Symposium on the Foundations of Software Engineering* (Nov. 3–5, 1998) pp 222–230.
Name: Philip W. L. Fong
Address: School of Computing Science, Simon Fraser University, Burnaby, BC V5A 1S6, Canada
Address: Tel: +1-604-291-4277
Address: Fax: +1-604-291-3045
Name: Robert D. Cameron
Address: School of Computing Science, Simon Fraser University, Burnaby, BC V5A 1S6, Canada
Address: Tel: +1-604-291-3241
Address: Fax: +1-604-291-3045

(1) **Interleaved logic.** Sun's implementation interleaves bytecode verification and loading. Java programs are composed of classes, each being loaded into the JVM separately. In the middle of verifying a class X, a new class Y may need to be loaded in order to provide enough information for the verification of X to proceed. For example, in order for the verifier to make sure that a method may throw an "`ArithmeticException`", it must check whether "`ArithmeticException`" is a subclass of the class "`Throwable`". As a result, the loader has to be invoked to bring in "`ArithmeticException`" and all its superclasses. Moreover, since the loader cannot trust the bytecode of "`ArithmeticException`" (and its superclasses) to be well-formed, part of the verification work must be carried out by the loader. As a result, verification and loading logic are interleaved in Sun's JVM.

(2) **Delocalized implementation.** Sun's bytecode verifier has a four-pass architecture. Pass one is the verification logic performed by the loader, as we have briefly mentioned. Passes two and three, performed by the bytecode verifier at link time, check for the well-formedness of bytecode files and carry out data flow analysis to type check methods of the underlying classes. Pass four is invoked at run time, whenever symbolic references need to be resolved. Consequently, security checks are scattered throughout the run-time system, again adding complexity to the task of analyzing the verification logic.

In the program understanding literature, it is well known that interleaving and delocalized program plans lead to programs that are difficult to comprehend [Rugaber et al. 1996; Letovsky and Soloway 1986]. This so-called "scattershot security" [McGraw and Felten 1997] adds considerable complexity to the task of implementing, validating and maintaining a reliable verifier.

Nevertheless, one may understand the rationale for current JVM architectures by considering the need to accommodate a lazy, dynamic linking strategy. Such a strategy seeks to defer expensive computations that may never be needed. For example, a class may be parsed but not further analyzed when only its interface is needed (pass one). Subsequently, its internal structure may be checked when code is linked in (passes two and three), but external references may be left unresolved in the event they are not needed. Finally, these external dependencies may be resolved individually as necessary at run time (pass four). Although such a strategy is not required by the JVM specification, the performance advantages should be easy to understand, particularly for classes with strong static coupling but weak dynamic coupling.

The above analysis reveals a software engineering challenge that is common to all dynamically-linked languages with both security and efficiency concerns. In particular, for mobile code systems which incorporate a security system based on link-time verification, one has to determine how loading, verification, and linking interact with each other so that the following goals are achieved simultaneously.

(1) **Laziness**: loading, verification, and linking can be deferred as long as possible.
(2) **Safety**: all necessary verification checks are performed before any code is executed.
(3) **Comprehensibility**: the resulting system architecture can easily be understood and thus verified.

As described previously, an *ad hoc* implementation of *laziness* dramatically increases the interleaving and delocalization of program plans within the system. This degrades *comprehensibility*, which may in turn lead to the loss of *safety*. We argue that a well-designed mobile code architecture should achieve the goals of safety and comprehensibility by localizing all the security-related code into a stand-alone verifier module free of loading and linking logic. In particular, it should allow one to *specify*, *craft*, *understand*, and *evaluate* the mobile code verifier as an individual engineering component, independent of the loading and linking procedures.

In this paper, we propose a language-independent infrastructure for building dynamically-linked mobile code systems. Our design results in a run-time environment that localizes the verification logic in a stand-alone module completely decoupled from loading and linking, while preserving laziness in dynamic linking. To achieve this, the verifier eschews the loading of classes to validate external dependencies. Instead, it converts each dependency into a *proof obligation*, which forms a safety precondition for endorsing the class or method being verified. Proof obligations are submitted to a *proof linker*, which is responsible for remembering and discharging them when the required external information becomes available as a result of class loading. We use the term *proof linking* to refer to the overall architecture for formulating and discharging verification obligations.

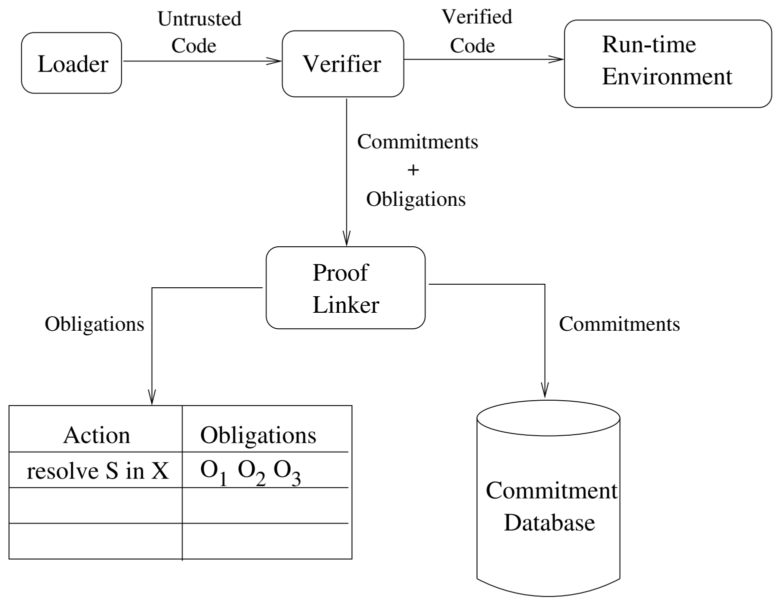


Fig. 1. Modular Verification

In support of the proof linking architecture, we also develop a formal model of proof linking and determine the conditions sufficient for proof linking to behave correctly. The correctness conditions are expressed in terms of a partial ordering of linking events, and as properties of a deductive database data model. We demonstrate how one can formulate Java type checking in our framework, and we also establish the correctness of this formulation. The correctness results have been formally checked by a theorem prover.

The architecture for modular verification is described in Section 2. Section 3 develops a theoretical framework in which we can articulate the correctness of modular verification in the presence of lazy, dynamic linking. Section 4 applies the modularization to Java bytecode verification, and demonstrates how the correctness of modular verification can be established. In Section 5, we briefly discuss alternative mobile code verification architectures enabled by the proof linking concept. The paper concludes with a discussion of related work and potential extensions. In the appendix, we discuss our experience with using a specification and verification tool to check the correctness proof presented in section 4.

2. A DYNAMIC-LINKING ARCHITECTURE

We assume that a program is composed of one or more *code units* (modules, classes and so on), each of which may contain externally visible *members* (functions, methods, variables, and so on). Code units and their members are identified by symbolic names. These names may be used to refer to the units or their members from within other units. When a program is executed, its code units are loaded, verified, and the symbolic references are incrementally replaced by actual machine pointers.

In the dynamic-linking architecture presented here, loading, verification, and linking are performed by three separate modules. No module attempts to invoke any other during its processing, nor will one recursively invoke itself. This poses the following challenge: *Verification requires knowledge of other code units which might not have been loaded yet. How do we remove such dependencies while maintaining the integrity of the verification process?* The problem is addressed by decomposition of verification into two subtasks: *modular verification* and *proof linking*.

2.1 Modular Verification

Figure 1 depicts the setup for modular verification. Untrusted code units are subjected to static verification after loading. The verifier might need the knowledge of another code unit in order to decide if the current code unit should be endorsed. Instead of recursively verifying (or even loading) the other code unit, the

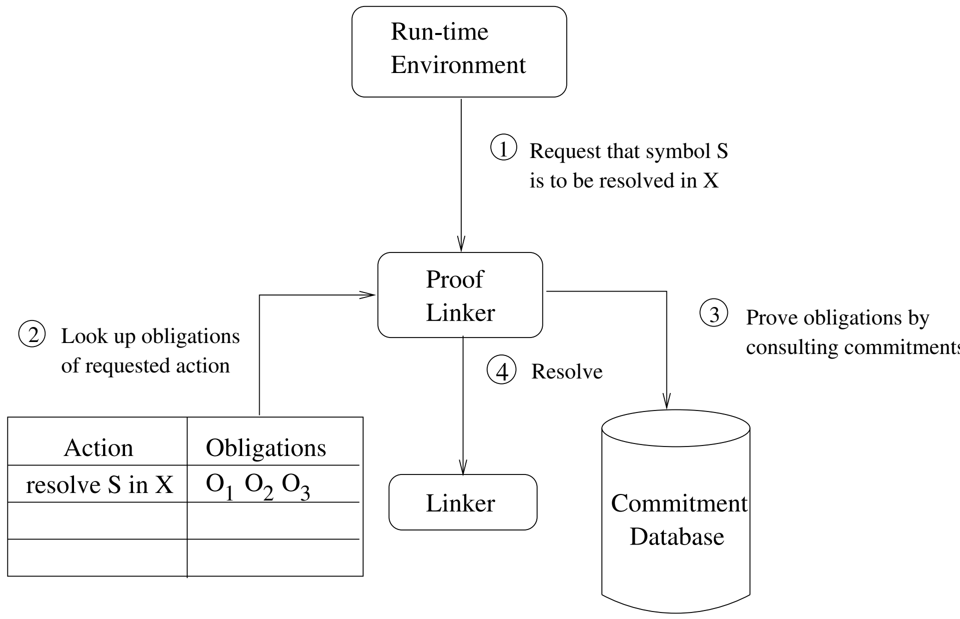


Fig. 2. Proof Linking

verifier computes a conservative *safety precondition* that will guarantee the safety of the code unit. The safety precondition is represented as a conjunctive set of database queries. For example, during the verification of a Java classfile, we might find out that an exception of class `ArithmeticException` can be raised by the code in the classfile. Since the classfile is safe only if `ArithmeticException` is a subclass of the Java class `Throwable`, the verifier formulates the query[1] `?subclass(ArithmeticException, Throwable)`. The Java verifier may end up generating many such queries. The conjunctive set of all queries formulated by a verification session becomes the safety precondition for endorsing the classfile being considered. More specifically, each of the queries describes a safety precondition of a certain linking action. For example, the query `?subclass(ArithmeticException, Throwable)` is a safety precondition for the action "resolving `ArithmeticException` in class `X`"[2]. Such queries are said to be the *proof obligations* for the associated actions, representing conditions that must be met if the run-time system attempts to safely perform the corresponding actions in the future. We say that a proof obligation is *attached* to its associated action. Obligations generated by the verifier are collected by the *proof linker*, which records in a global *obligation table* the mapping from linking actions to their attached obligations.

In order for the run-time system to discharge proof obligations, the verifier also computes, for each code unit, a set of clauses called *commitments*. The commitments are ground facts that describe the interface properties of the code unit. For example, during the verification of the Java classfile `ArithmeticException`, the verifier generates a commitment `!extends(ArithmeticException, Exception)` to indicate that `Exception` is the immediate superclass of `ArithmeticException`. The generated commitments are collected by the proof linker, and subsequently asserted into a global *commitment database*. When proof obligations are to be checked, the commitment database provides the set of facts against which the query can be evaluated.

2.2 Proof Linking

The process by which the run-time system cross-validates the results of verifying different code units is called *proof linking*. Figure 2 depicts the setup for proof linking. When the run-time system needs to resolve a symbolic reference to a machine pointer, it sends the request to the proof linker. The proof linker looks up

[1]To differentiate the various roles played by a predicate symbol, we prefix a query by a question mark ("?") and an assertion by an exclamation mark ("!").
[2]As later sections will point out, the query `?subclass(ArithmeticException, Throwable)` is actually associated with an action other than the one being mentioned here. We describe a simplified scenario just to illustrate our point.

the obligations that have been attached to the request, and then posts them to the commitment database as deductive queries. If the queries are satisfied, the requested action is performed. Otherwise, a linking exception is raised to signal failure to endorse the consistency of the code units.

To make proof linking more expressive, arbitrary logic programs can be provided as an initial theory in the commitment database. For example, recursive definitions of the following form can be supplied to capture the transitive closure of the subclassing relationship:

```
subclass(X, X).
subclass(X, Y) :- extends(X, Z), subclass(Z, Y).
```

If the verifier asserts commitments

```
!extends(ArithmeticException, Exception)
```

and

```
!extends(Exception, Throwable)
```

then the obligation

```
?subclass(ArithmeticException, Throwable)
```

can be deduced.

2.3 Implementation

Although we have used a deductive database model as a means of representing obligations and commitments, we do not require that an actual system be implemented this way. It may be that such an implementation would be unacceptably slow, because loading and linking in a mobile code system occurs frequently. Given queries and commitments of fixed signatures, and given a fixed initial theory, appropriate data structures and algorithms can be designed for the efficient assertion of commitments and discharging of obligations. For example, the commitments collectively defining the the `extends` relationship can be represented in a space-efficient manner using an appropriate graph data structure, while the logic of the `subclass` relationship (transitive closure) may be implemented using a graph traversal algorithm.

There are two reasons to model proof linking as a series of database updates and queries. First of all, the database model provides an abstract framework to describe the general notion of proof linking, without getting into the idiosyncrasies of individual mobile code systems. Secondly, and more importantly, it allows us to define a formal model of proof linking and its correctness conditions, a topic to which we now turn.

3. CORRECTNESS OF INCREMENTAL PROOF LINKING

In this section, we consider the following three correctness conditions for proof linking.

(1) **Safety:** All obligations relevant to the safe execution of a code unit are generated and checked before that unit is executed.
(2) **Monotonicity:** Once an obligation is satisfied, no subsequently asserted commitment will contradict it.
(3) **Completion:** All commitments that may be needed to satisfy an obligation are generated before the obligation is checked.

Note the parallel between the complete generation of obligations required by the safety property and the complete generation of commitments required by the completion property. There is also an interesting parallel between monotonicity and completion. The latter may be rephrased to state that once an obligation fails, no subsequently asserted commitment will enable it.

In summary, the properties of safety, monotonicity and completion are intended to ensure that the checking of obligations and the enabling of code unit execution are deterministic processes even though the lazy, dynamic linking procedure is not. In essence, we characterize the correctness of proof linking as the correct scheduling of static verification steps over time.

The remainder of this section formalizes these notions as follows. Section 3.1 models dynamic linking processes as partially-ordered sequences of primitive linking actions. Section 3.2 presents a simple proof linking algorithm as an operational definition of commitment and obligation processing. Section 3.3 then goes on to formalize the safety, monotonicity and completion properties in terms of the terminology developed in the previous two subsections.

3.1 A Model for Lazy Dynamic Linking

The fundamental simplification achieved by our dynamic linking architecture is that loading, verification and linking may be decomposed into independent primitive actions. That is, although concurrent execution of the primitive actions is allowed, each step of loading or verifying a particular code unit, or resolving a particular external reference may be considered as a self-contained action independent of any other. Thus, we model actions by *linking primitives*, each of which can be executed at most once during the life time of the run-time environment. Although the precise set of primitives that are used in a particular system may vary, we assume that the following minimal set exists for each code unit X:

load X**:** acquire code unit X.
verify X**:** verify code unit X.
resolve S **in** X**:** replace symbolic reference S in code unit X with an actual machine pointer.
use S **in** X**:** symbolic reference S in code unit X is used for the first time.

Associated with each linking primitive p are two *linking events*, namely, "**begin** p" and "**end** p", which respectively represent the initiation and termination of the primitive p. These events occur asynchronously as the run-time system executes various linking primitives. We assume that events are then queued up in some synchronized event queue, waiting to be examined by the proof linker. Intuitively, when the run-time system requests that a linking primitive p be authorized to execute, "**begin** p" will be generated. Similarly, the run-time system generates "**end** p" to inform the proof linker that p is properly terminated. The sequence of linking events that enters the event queue from the beginning of an execution session to some point of execution is said to be an *execution trace* of the run-time system in that period of time. We further assume that, event "**end** p" can occur in an execution trace only if there is a corresponding event "**begin** p" occurring strictly before it.

Given a set P of linking primitives, a linking strategy $\sigma = (P, <)$ is a strict partial ordering of the linking primitives in P. Every implementation of a mobile code run-time system defines a linking strategy. The strategy expresses the order in which linking events may be processed by the run-time system. More precisely, an execution trace τ is σ-conforming if the following hold: (1) all linking events in τ initiate or terminate primitives from P, and (2) for all p, $q \in P$ such that $p <_\sigma q$, if "**begin** q" occurs in τ then, "**end** p" occurs in τ before "**begin** q". To say that a run-time system implements a linking strategy σ is to say that the run-time system guarantees that all possible execution traces are σ-conforming. Notice that this definition of linking strategy allows primitives to be executed concurrently as long as the strategy does not explicitly order them.

A strategy is *admissible* if the following properties hold: Given any code units X and Y, and a symbol S imported by X from Y, we have

(1) **Natural Progression Property:**

$$\textbf{load } X < \textbf{verify } X < \textbf{resolve } S \textbf{ in } X$$

(2) **Import-Checked Property:**

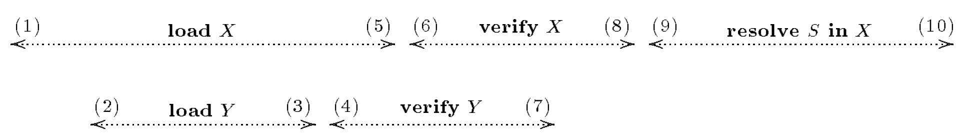
$$\textbf{verify } Y < \textbf{resolve } S \textbf{ in } X < \textbf{use } S \textbf{ in } X$$

We shall only consider admissible strategies hereafter.

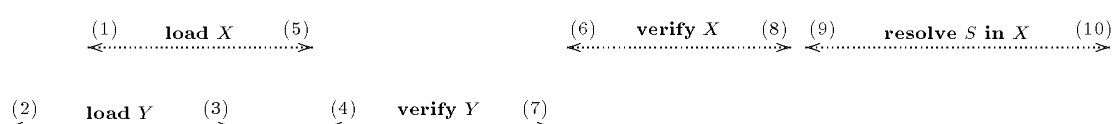
For example, consider the minimal strategy imposing only the Natural Progression and Import-Checked Properties as ordering constraints. Assuming that code unit X imports symbol S from code unit Y, the following execution trace conforms to the strategy:

(1) **begin** "**load** X"
(2) **begin** "**load** Y"
(3) **end** "**load** Y"
(4) **begin** "**verify** Y"
(5) **end** "**load** X"
(6) **begin** "**verify** X"
(7) **end** "**verify** Y"
(8) **end** "**verify** X"
(9) **begin** "**resolve** S **in** X"
(10) **end** "**resolve** S **in** X"

The ordering of events corresponds to the following timeline:



Switching the relative ordering of events (1) and (2) results in a new execution trace that still conforms to the strategy. Further switches of (4) with (5) and (6) with (7) also maintain conformance and lead to an execution trace illustrated by the following timeline diagram:



Now, if we further switch the position of (8) with (9) then the resulting execution trace would violate the **Natural Progression Property**. Similarly, moving event (7) after (9) would violate the **Import-Checked Property**.

3.2 A Model Proof Linking Algorithm

We model proof linking with an algorithm in which every linking primitive may generate both commitments and obligations. Commitments are the facts describing the information collected as a result of executing a primitive. Obligations are the queries that are attached as preconditions to subsequent primitives, called *targets*. An obligation-target pair is called an *attachment*. Notice that obligations can be attached to targets other than the **resolve** primitive.

Figure 3 presents a model proof-linking algorithm in which linking primitives are consumed from a global event queue. The proof linker maintains two global data structures, namely, a commitment database (`DB`) and an obligation table (`Obligations[]`). The commitment database is a decidable first-order theory containing both facts and rules. The obligation table maps each linking primitive to a set of database queries. Initially, the commitment database contains an initial theory (`Initial-Theory`), and the obligation table is empty (line 1). The proof linker consumes linking events in the order specified by the linking strategy (line 4). When the **begin** event of a linking primitive is removed from the event queue (line 6), its associated obligations are retrieved from the obligation table (line 8). The verification of these obligations is then attempted

```
algorithm ProofLinker(Initial-Theory):

01: DB ← Initial-Theory; Obligations[] ← ∅;
02: Ready ← ∅; Satisfied ← ∅; Failed ← ∅;
03: while (¬ run-time-env-terminated()) do
04:   e ← get-next-event();
05:   switch e of
06:   case "begin p":
07:     All-Obligations-Satisfied ← true;
08:     for all o ∈ Obligations[p] do
09:       if (DB ⊢ o) then
10:         Satisfied ← Satisfied ∪ { o };
11:       else
12:         Failed ← Failed ∪ { o };
13:         All-Obligations-Satisfied ← false;
14:       end if
15:     end for
16:     if (All-Obligations-Satisfied) then
17:       Ready ← Ready ∪ { p };
18:       authorize the execution of p;
19:     else
20:       deny request to execute p;
21:     endif
22:   case "end p":
23:     DB ← DB ∪ get-commitments(p);
24:     for all ⟨o, t⟩ ∈ get-attachments(p) do
25:       Obligations[t] ← Obligations[t] ∪ {o};
26:     end for
27:   end switch
28: end while
```

Fig. 3. The Proof-Linker Model Algorithm

against the logic program in the commitment database (line 9). If all obligations are satisfied (line 16), then the linking primitive will be allowed to execute (line 18); otherwise, its execution will be denied (line 20). Alternatively, when the **end** event of a primitive is removed from the event queue (line 22), the commitments and attachments for the primitive are collected. The commitments are added to the commitment database (line 23). The attachments are incorporated into the obligation table (lines 24–26). The proof linker repeats this process until the run-time environment terminates (line 3).

3.3 Formalization of Correctness Conditions

To formalize the correctness conditions of the proof linker, we have introduced three auxiliary variables into the listing in Figure 3. "`Satisfied`" (lines 2 and 10) denotes the set of obligations that have already been checked at line 9, while "`Failed`" (line 2 and 12) collects obligations that have failed to check. "`Ready`" (lines 2 and 17) is the set of primitives that are ready for execution.

Given a fixed, admissible linking strategy σ, the proof linker is correct if the following conditions hold:

(1) **Safety:** Before any primitive is executed, all obligations that may potentially be attached to it are generated and checked. For any linking primitives x and t, if x may introduce the attachment $\langle o, t\rangle$, then we require that $x <_\sigma t$. As a result, the following invariant will hold at all times:

$$\forall p \in \texttt{Ready} \ . \ \forall o \in \texttt{Obligations}[p] \ . \ o \in \texttt{Satisfied}$$

(2) **Monotonicity:** Obligations may not be contradicted by subsequently asserted commitments. The monotonicity requirement may be summarized by asserting that the following invariant holds at all times:

$$\forall o \in \texttt{Satisfied} \ . \ \texttt{DB} \vdash o$$

In our deductive database model, monotonicity results naturally from the application of Horn clause logic [Lloyd 1987]. The initial theory and generated commitments are required to be Horn clauses (definite clauses), thus ensuring that no contradiction is possible. Obligations should be formulated only as definite queries, thus ensuring that subsequent commitments do not contradict conclusions reached through negation-by-failure. For a brief discussion of how monotonicity can be guaranteed when other form of logic is used, consult appendix A.4.

(3) **Completion:** A commitment c is said to *support* an obligation o if there is a proof of o in which c is a necessary premise. If a linking primitive p may assert a commitment that supports o, and if o may be attached to linking primitive q, then we require that $p <_\sigma q$. Thus, if an obligation o of primitive q is eventually provable, then generation of the commitments necessary for its proof must be complete when "**begin** q" is processed. Conversely, obligation failure may not be subsequently contradicted by asserted commitments:

$$\forall o \in \texttt{Failed} \,.\, \texttt{DB} \not\vdash o$$

In general, the correctness of proof linking depends on (1) the linking strategy σ, (2) the kind of logic being used, and (3) the specific commitments and obligations returned by each linking primitive. In particular, the safety and completion conditions constrain the linking strategy to ensure that an obligation is checked neither too late nor too early.

Note that the correctness conditions do not impose a strict policy on the linking strategy. Either eager linking (linking every code unit at once) or lazy linking (linking a code unit only when its code is being executed)—or indeed any intermediate strategy—can be tailored to satisfy the correctness conditions. To maximize the opportunities for laziness, however, we prefer strategies with fewer ordering constraints so long as the correctness conditions hold.

4. AN EXAMPLE: JAVA BYTECODE VERIFICATION

This section describes an instantiation of our modular verification framework. Specifically, we use Java bytecode verification as an example to illustrate various concepts discussed in previous sections. In particular, we develop an admissible strategy for Java proof linking and prove that any system implementing this strategy satisfies the the correctness conditions for proof linking.

4.1 The Java Linking Model

In Java, the code units of our model are classes. The standard Java classloading semantics uses multiple classloaders to implement namespace partitioning. A loaded class is identified by both its classname and its defining classloader [Liang and Bracha 1998]. Since the focus of this research is on analyzing how the deferred discharging of obligations interacts with linking strategies, we consider a simplified model in which there is a single classloader. As a result of this simplification, classnames are sufficient for identifying code units. Interested readers are referred to section 4.5 for a brief discussion of how our model may be generalized to one involving multiple classloaders.

A symbolic reference in Java may refer to either a class[3] or a member of a class. Class references are simply classnames. Member references refer to either fields or methods. As a class may contain multiple members with the same identifier, both the identifier and the descriptor (type signature) of a member are generally needed to uniquely denote the member within a class. The descriptor of a field specifies its type, while that of a method specifies the type of both its formal parameters and its return value. A member M of a class X with descriptor S has a symbolic reference of the form $X::M(S)$.

Additional linking primitives are introduced. Since class symbols and member symbols are resolved separately, we denote the linking primitive that resolves method $Y::M(S)$ in class X by "**resolve** $Y::M(S)$ **in**

[3]Although our scheme assumes a programming model in which all classes are *top-level* classes, it is sufficient for handling Java *inner classes*, which were introduced in Java 1.1 so that programmers can define classes as members of other classes, locally within a block of statements, or within an expression[JavaSoft 1997]. However, inner classes are strictly source language constructs implemented entirely by a Java source compiler which transforms all Java 1.1 inner classes into Java 1.0 bytecode. Consequently, the presence of inner classes requires no change to our scheme, which assumes a standard JVM bytecode semantics.

X", and we reserve the simpler syntax of "**resolve** Y **in** X" for resolution of classes. We also introduce auxiliary primitives "**endorse** Y" and "**endorse** $Y{::}M(S)$", with the intuitive semantics of declaring the corresponding objects to be ready for linking. These primitives serve as place holders to which one may attach obligations that should be checked before a symbol is resolved.

We articulate an admissible strategy for Java linking. We first modify the Natural Progression Property and the Import-Checked Property to accommodate the introduction of new primitives, and then add further properties to capture the linking dependencies peculiar to Java.

(1) **Natural Progression Property:**

load X < **verify** X < **endorse** X < **resolve** Y **in** X < **resolve** $Y{::}M(S)$ **in** X

(2) **Import-Checked Property:**

endorse Y < **resolve** Y **in** X < **use** Y **in** X

and also

endorse Y < **endorse** $Y{::}M(S)$ < **resolve** $Y{::}M(S)$ **in** X < **use** $Y{::}M(S)$ **in** X

(3) **Subtype Dependency Property:** To establish an obligation concerning a class, type information about its superclasses and superinterfaces might be needed. For example, to show that a class is properly defined, one has to show that none of its superclasses is declared final, and that all of its superinterfaces are properly defined interfaces. To address this need, we require, for all superclasses and superinterfaces Y of X,

verify Y < **endorse** X

(4) **Referential Dependency Property:** Sometimes, verification of a class Y is needed before we can safely endorse a method $X{::}M(S)$. For example, if method $X{::}M(S)$ assigns a reference of type Y to a variable of type Z, then Java type rules require Z to be either a superclass or a superinterface of Y. Unless Y is a superclass of X, it is entirely possible that the superclasses and superinterfaces of Y are not verified yet. Consequently, the required supports for the obligation are not necessarily present at the time the obligation is checked, a violation of the completion property. In such a case, we say that Y is *relevant* to the endorsing of $X{::}M(S)$. We assume that, statically associated with the bytecode of each method $X{::}M(S)$ is a set of relevant classes Y, and we require that

endorse Y < **endorse** $X{::}M(S)$

That is, we want to verify all relevant classes (plus their superclasses and superinterfaces) before we check the obligations attached to "**endorse** $X{::}M(S)$". Since the notion of relevance is statically defined, an implementation may enforce the above ordering constraint by identifying the relevant classes when "**verify** X" scans the bytecode of X. Equipped with this information, the run-time system can consistently endorse all relevant classes before endorsing a method of X.

4.2 Commitments, Obligations, and Initial Theory for Java Type Checking

In Java, only the "**verify** X" primitive generates commitments and obligations. Figure 4 describes the commitments generated by "**verify** X". Figure 5 describes the obligations generated by "**verify** X", together with the primitives to which the generated obligations are attached. Figure 6 shows the clauses in the initial theory. The commitments, obligations, and initial theory described here captures Java verification passes 1 through 3, now integrated within a single verification primitive[4]. Further improvement could be achieved by formulating commitments and obligations related to the checking of resolution errors— the fourth pass of verification [Lindholm and Yellin 1999, chapter 5]. However, because this exercise is a conceptually straightforward extension, and because it has less of an impact on enabling other verification protocols (see Sections 5.2 and 5.3), we omit it to facilitate our presentation.

[4]Because pass 4 (resolution of constant pool entries) is not included in the modelling, and because some of the verification checks are performed entirely within the **verify** primitive, it is not necessary to capture all typing information (e.g. whether a member is abstract) in Figures 4 and 5. The specified commitments and obligations reflect the dependencies between the loader and the verifier.

`!class(`X`)` : *X is a non-interface class.*
`!interface(`X`)` : *X is an interface class.*
`!non_final(`X`)` : *X is not declared to be final.*
`!package(`P`,` X`)` : *X is defined in package P.*
`!extends(`X`,` Y`)` : *Y is a direct superclass of X.*
`!implements(`X`,` L`)` : *L is the list of all direct superinterfaces of X.*
`!member(`X`,` M`,` S`)` : *$M(S)$ is a member of X.*
`!public_member(`X`,` M`,` S`)` : *$M(S)$ is a public member of X.*
`!protected_member(`X`,` M`,` S`)` : *$M(S)$ is a protected member of X.*
`!default_member(`X`,` M`,` S`)` : *The member $M(S)$ has default access in X.*
`!private_member(`X`,` M`,` S`)` : *$M(S)$ is a private member of X.*

Fig. 4. Commitments that may be asserted by **verify** X

`?subclassable(`Y`)`
Target: **endorse** X
Intention: Direct superclass Y of X can be subclassed.

`?implementable(`L`)`
Target: **endorse** X
Intention: The list L of direct superinterfaces for X are properly defined interfaces.

`?class(`Y`)`
Target: **resolve** Y **in** X
Intention: Y should be a non-interface class.

`?interface(`Y`)`
Target: **resolve** Y **in** X
Intention: Y should be an interface.

`?member(`Y`,` M`,` S`)`
Target: **resolve** Y::$M(S)$ **in** X
Intention: $M(S)$ is a member of Y.

`?throwable(`Y`)`
Target: **endorse** X::$M(S)$
Intention: Identified as relevant to X::$M(S)$, class Y is throwable.

`?subclass(`Y`,` Z`)`
Target: **endorse** X::$M(S)$
Intention: Identified as relevant to X::$M(S)$, Y is a subclass of Z.

`?subclass(`X`,` Y`)`
Target: **resolve** Y::$M(S)$ **in** X
Intention: X is a subclass of Y.

`?superinterface(`Z`,` Y`)`
Target: **endorse** X::$M(S)$
Intention: Identified as relevant to X::$M(S)$, Y has Z as a superinterface.

`?assignment_compatible(`Y`,` Z`)`
Target: **endorse** X::$M(S)$
Intention: Identified as relevant to X::$M(S)$, Y is assignment compatible to Z.

`?accessible_member(`Y`,` M`,` S`,` X`,` Z`)`
Target: **resolve** Y::$M(S)$ **in** X
Intention: Asserted when a method $N(T)$ of X is verified. It requires that, Z being relevant to X::$N(T)$, a reference of type Z can be used to access the member $M(S)$ of Y.

Fig. 5. Obligations that may be asserted by **verify** X

```
;; implementable(List) : Are all interfaces in List properly defined?

implementable([]).
implementable([X|T]) :- interface(X), implements(X, I), implementable(I),
                        implementable(T).

;; subclassable(X) : Is X a properly-defined super class?

subclassable('java/lang/Object').
subclassable(X) :- class(X), non_final(X), implements(X, I), implementable(I),
                   extends(X, Y), subclassable(Y).

;; subclass(X, Y) : Is X a subclass of Y?

subclass(X, X).
subclass(X, Z) :- extends(X, Y), subclass(Y, Z).

;; throwable(X) : Can X be thrown as an exception?

throwable(X) :- subclass(X, 'java/lang/Throwable').

;; transitively_implements(X, Y) : Transitive closure of implements

transitively_implements(X, Y) :- implements(X, I), member(Y, I).
transitively_implements(X, Z) :- implements(X, I), member(Y, I),
                                 transitively_implements(Y, Z).

;; superinterface(Y, X) : Is Y a superinterface of X?

superinterface(X, X) :- interface(X).
superinterface(Z, X) :- subclass(X, Y), transitively_implements(Y, Z).

;; assignment_compatible(X, Y) : Can X be assigned to Y?

assignment_compatible(X, Y) :- subclass(X, Y).
assignment_compatible(X, Y) :- superinterface(Y, X).

;; accessible_member(Y, M, S, X, Z) :
;;  Can one apply getfield/putfield/invokevirtual/invokespecial in a
;;  class X to a reference type Z in order to access the member Y::M(S)

accessible_member(Y, M, S, _, _) :- public_member(Y, M, S).
accessible_member(Y, M, S, X, _) :- protected_member(Y, M, S),
                                    package(P, X), package(P, Y).
accessible_member(Y, M, S, X, Z) :- protected_member(Y, M, S),
                                    subclass(X, Y), subclass(Z, X).
accessible_member(Y, M, S, _, _) :- default_member(Y, M, S).
accessible_member(Y, M, S, _, _) :- private_member(Y, M, S).
```

Fig. 6. The Initial Theory

4.3 Correctness of Java Proof Linking — A Proof Sketch

We now assume the construction of a Java proof linker that generates commitments and obligations as described in section 4.2 and that enforces the ordering constraints of section 4.1. We prove that such an implementation satisfies the correctness conditions for proof linking.

(1) **Safety:** Only "**verify** X" generates obligations. As shown in Figure 5, the generated obligations are only attached to "**endorse** X", "**endorse** X::$M(S)$", "**resolve** Y **in** X", and "**resolve** Y::$M(S)$ **in** X. In each case, however, the proof linker processes these primitives only after the appropriate "**verify** X" primitive, in accord with the Natural Progression and Import-Checked properties.

(2) **Monotonicity:** The initial theory and the commitments are all in Horn clause form. Obligations are expressed as definite queries. Monotonicity is thereby assured.

(3) **Completion:** Consider the obligation `?subclassable(`Y`)` attached to "**endorse** X" by "**verify** X". Supports of this obligation are asserted by all primitives of the form "**verify** Z", where Z is either Y or one of its superclasses or superinterfaces. It then suffices to show that "**verify** Z" < "**endorse** X". According to Figure 5, the obligation is imposed only when Y is declared to be a direct superclass of X. The Subtype Dependency Property guarantees that Y and all its superclasses and superinterfaces are verified before X is endorsed. Therefore, the obligation is consistently established.
Consider now the obligation `?assignment_compatible(`Y`, `Z`)` attached to "**endorse** X::$M(S)$". Supports of the obligation are asserted by all primitives of the form "**verify** W", where W is Y or a superclass or superinterface thereof. As shown in Figure 5, Y is relevant to X::$M(S)$ if the the obligation is to be asserted. It then follows from the Referential Dependency Property that the superclasses and superinterfaces of Y are already verified when the obligation is tested. Thus, all the supports are already present, and the obligation can be consistently established.
Using similar arguments, one can establish that completion holds for all obligations.

We have formalized and checked the above reasoning using the PVS specification and verification tool [SRI Computer Science Laboratory]. The details are presented in the appendix.

4.4 Implementing Commitment and Obligation Generation

To debug our formulation, we have implemented a stand-alone Java bytecode verifier that performs modular verification and proof linking. As in Sun's `javaverify` application, our system involves only one classloader. Given a target class X, the verification system performs the following steps.

(1) The classfile for X is loaded ("**load** X").

(2) The bytecode for X is verified (**begin** "**verify** X").

(3) The commitments and obligations for the **verify** primitive are generated in accord with the specifications of Figures 4 and 5 (**end** "**verify** X"). All relevant classes are also identified.

(4) All **endorse** and **resolve** primitives to which obligations are attached are scheduled for execution, thereby checking the obligations according to the logic prescribed in Figure 6.

In accordance with the linking strategy described in Section 4.1, all superclasses, superinterfaces and relevant classes of X will be appropriately loaded, verified or endorsed as a result of the last step.

Several implementation details are worth noting.

Meet computation. Java type checking involves data flow analysis. In Sun's implementation of the data flow analyzer, the meet of two classes Z_1 and Z_2 is their most specific common superclass [Lindholm and Yellin 1999, Chapter 4]. To compute this superclass at verification time, Sun's JVM immediately loads all superclasses of Z_1 and Z_2. Our implementation defers recursive loading by representing the meet symbolically as $Z_1 \sqcap Z_2$, the semantics of which is that the reference could either be that of Z_1 or Z_2, and thus any operation on such a reference should be supported by the type interfaces of both Z_1

and Z_2. Obligations are then formulated in terms of these *meet expressions*.[5] Specifically, when the obligation $P(Z_1 \sqcap Z_2)$ is to be imposed, the verifier will generate both $P(Z_1)$ and $P(Z_2)$. For example, if `?subclass`$(Y,\ Z_1 \sqcap Z_2)$ is found to be a safety precondition, then the two obligations `?subclass`$(Y,\ Z_1)$ and `?subclass`$(Y,\ Z_2)$ will be generated. This arrangement is more relaxed than Sun's current implementation, but it is strong enough for preserving type safety.

Arrays. Java arrays are classes, and there are special type rules for handling them. However, these rules are ultimately formulated in terms of type rules of ordinary classes. For example, an array of A is assignment compatible to `java.lang.Object` and `java.lang.Cloneable`, and an array of A is assignment compatible to an array of B only if A is assignment compatible to B. As a result, the modular verifier can always translate an obligation involving array types into one or more (conjunctive) obligations that are free of array type. Therefore, our existing verification system can still manage arrays correctly.

Optimization. Notice that our Java proof linking strategy is more or less fixed, except for the ordering imposed by the Referential Dependency Property, which is defined in terms of the relevance relation. Specifically, if a class is not identified to be relevant to a method, less ordering, and thus more laziness, results. To optimize the linking process, a modular verifier may choose not to identify a class as relevant if it is redundant to do so. For example, if the modular verifier attempts to attach `?subclass(`X`, `Y`)` to "**endorse** Z", normally, X will need to be identified as relevant. But in the case when $X = Y$, one knows that `?subclass(`X`, `Y`)` is trivially true. So, the obligation and its corresponding relevance annotation need not be generated in this case. Similar optimizations can be adopted for obligations that involve other special case combinations of rules in the initial theory.

An implementation such as the above may be used as an infrastructure for the work of Devanbu, Fong and Stubblebine [Devanbu et al. 1998]. An online demonstration of modular verification and commitment/obligation generation can be accessed at:

`http://www.cs.sfu.ca/~pwfong/personal/Research/Verifier`.

4.5 Limitations

As we have mentioned, both our model and our implementation assumes that there is only a single classloader. In real Java platforms, multiple namespaces can be created by defining multiple classloaders. A Java class is then identified not only by its name, but by both its name and the classloader in which the class is defined. Formally, when a Java application attempts to load a class C with a given name X by a classloader L_i, *the initiating classloader* of C, L_i may delegate the classloading to another classloader, which, in turn, might delegate the task to yet another classloader. The classloader L_d that eventually loads and defines C is said to be its the *defining classloader*. C is uniquely identified by the pair $\langle X, L_d\rangle$. We also write $X^{L_i} = \langle X, L_d\rangle$ to indicate the fact that L_i initiates the loading of $\langle N, L_d\rangle$. See [Lindholm and Yellin 1999, Chapter 5] and [Liang and Bracha 1998] for more details.

A simple attempt to account for the complexity introduced by multiple classloaders would be to replace each classname X in commitments and obligations by a classname-classloader pair $\langle X, L_0\rangle$. The problem with this approach is that, at commitment-assertion time or obligation-attachment time, the defining classloader L_0 of the classes involved might not be known (because the class $\langle X, L_0\rangle$ has not been loaded yet). However, the initiating classloader L_1 for such a class is already known. To see this, note that commitments and obligations are introduced only by a **verify** primitive. By the Natural Progression Property, the defining classloader L_1 of the verified class is already known. Java semantics dictates that this L_1 will be used as the initiating classloader for all the classes referenced in the verified class. That is, all the classes appearing in the commitments and obligations introduced by the **verify** primitive have L_1 as initiating classloader. Consequently, class references occurring in commitments and obligations should be represented by the classnames and their corresponding initiating classloaders. The following extensions to our scheme are required:

(1) The binding of a class $\langle X, L_d\rangle$ to a name X^{L_i} in an initiating class loader L_i should be modelled explicitly as a linking primitive, "**bind** $\langle X, L_d\rangle$ **to** X^{L_i}".

[5] Such a symbolic representation does not affect the termination of the data flow analysis because the number of class symbols that may appear in a method is finite, and the underlying lattice is consequently bounded [Kam and Ullman 1977].

(2) The above primitive should assert the commitment $X^{L_i} \mapsto \langle X, L_d \rangle$.
(3) The rules in the initial theory should be updated to make use of the binding commitments ($\mapsto$) for explicitly resolving the occurrences of X^{L_i} in commitments and obligations to $\langle X, L_d \rangle$.

With appropriate refinement to our linking strategy, we expect that the above scheme will account for multiple classloaders. This remains an area for further work.

5. RETHINKING VERIFICATION

The proof linking concept invites several natural extensions to mobile code verification.

5.1 Replaceable Verifier Module

It is generally desirable for the mobile code verification technology to evolve independently of the mobile code hosting technology. In the context of Java applet verification, this would mean that the bytecode verifier is manufactured as a replaceable component that can be "plugged" into any browser that supports pluggable verification engine. Third party vendors can then specialize in producing highly secure verifier modules, while JVM vendors can concentrate their efforts on producing faster virtual machines. As a result, installation of a browser of one brand does not preclude the adoption of a Java bytecode verifier of another brand. This business model should yield higher quality and more secure mobile code hosting environments.

Our proof linking architecture is a framework for identifying and reducing the coupling between a mobile code hosting environment and its verification component. Consequently, it may represent a good basis for further work in developing replaceable verifier modules.

5.2 Remote Verification

Modularization makes it feasible for mobile code verification to be performed remotely. The example in section 4 only requires that the **verify** primitive correctly generates all commitments and obligations. It does not specify how such commitments and obligations are generated. Therefore, a remote Java bytecode verifier can analyze a classfile, generate the corresponding commitments, obligations and relevance annotations, and digitally sign the entire package. Upon acquiring the package, a browser can perform a special **verify** primitive that (1) authenticates the signature of the package, and (2) processes the commitments and obligations as if they were generated locally. To the proof linker, this special **verify** primitive looks no different than a normal **verify** primitive, and will proof-link the remotely-verified classfile correctly. Had we not modularized verification, remote verification would not be possible, because the verification of one classfile will require the knowledge of other classfiles, which may not be accessible at the remote verifier's site. Combining modular verification with key-management technologies, and by employing a physically secure coprocessor to perform verification, Devanbu, Fong, and Stubblebine [Devanbu et al. 1998] produce a distributed mobile code verification architecture that has various security-related and configuration-management-related benefits. In this way, verification becomes a service that may be offered by third-party providers.

5.3 Interoperability of Verification Protocols

Modularization of verification can also provide interoperability among various *verification protocols*. A verification protocol specifies how various concerned parties cooperate to carry out an overall verification process. There are at least three known verification protocols in the mobile code literature:

(1) **Proof-on-demand**: The existing implementations of Java bytecode verification follow this protocol. Verification is performed dynamically whenever a classfile is linked into the run-time environment. The protocol introduces link-time overhead, but it allows dynamically-generated code to be verified properly.
(2) **Proof-carrying code [Necula 1997; Colby et al. 2000]**: Verification is performed at the originating site. A safety proof is attached to a code module when it is shipped. Upon arrival at the execution site, the safety proof is checked before execution is allowed. Since proof checking may be substantially easier than proof generation, this protocol can involve less link-time overhead than proof-on-demand. Furthermore, since proof generation may now be performed once and for all at compile time, one can afford to consider difficult-to-prove safety properties, including those that have to be verified with human assistance.

(3) **Proof delegation [Devanbu et al. 1998]**: Code is passed to a trusted program analyzer, which certifies the safety of the code, and then digitally signs it. Upon arrival at the execution site, verification is replaced by signature checking. This protocol is potentially the most efficient when the safety properties can be mechanically established.

Since each of these protocols has merit, it is worthwhile to avoid premature commitment to a single one. For example, a mobile program may consist of some uncertified modules, some proof-carrying modules, and some remotely-certified modules, each from a different source. To make this possible, we need to be able to combine the results of verification produced by multiple protocols. Proof linking provides an infrastructure for such interoperability. In particular, each protocol is handled by a specialized **verify** primitive. In the case of proof-on-demand, the corresponding **verify** primitive will verify an untrusted code unit and generate obligations and commitments as usual. In the cases of proof-carrying code and proof delegation, an untrusted code unit will carry commitments and obligations generated by a remote verifier. Upon acquiring the code unit, the corresponding **verify** primitives perform either proof checking or signature authentication, and assert the attached obligations and commitments as if they were generated locally. Proof linking thus proceeds normally even in the presence of multiple verification protocols.

6. RELATED WORK

Neither dynamic linking nor modular verification is new. Dynamic linking is, of course, common in operating systems like UNIX and Windows. The notion of local certifiability is well known in software engineering [Weide and Hollingsworth 1992] and the composability of security features was studied as early as the 1980's [McCullough 1987; McCullough 1988]. The principal contribution of this paper is the definition of a concept and an architecture that permits modular verification in the presence of lazy, dynamic linking. In particular, this paper advocates the following two ideas:

—**Modular verification.** Whole-program verification problems can be modularized by the formulation of appropriate proof obligations.

—**Proof linking.** Incremental discharging of proof obligations in the presence of lazy, dynamic linking can be made safe and complete by careful scheduling of linking events in the form of a linking strategy.

Safe, dynamic linking of mobile code units as exemplified in the Java platform [Liang and Bracha 1998] has recently been a topic of considerable interest. Dean pioneered the study of static typing with dynamically-linked mobile code [Dean 1997]. Using the PVS specification language, he defines an abstract implementation of Java's classloading mechanism which models the behavior of multiple, user-definable classloaders. He proved that his implementation does not produce type confusion, that is, classloaders always return the same class for a given classname. Essentially, he has proven that his implementation satisfies a property called consistent extension of typing environments, that is, previously checked type judgments remain valid in any extended type environment that may be created due to dynamic linking. In our work, consistency of type extension is achieved by the monotonicity condition. However, in order to focus on the issue of linking strategies, we do not consider the scenario of multiple classloaders.

Jensen *et al* [Jensen et al. 1998] defined a formal model for reasoning about the dynamic states of class loading in a JVM. Despite some modelling inaccuracies [Bracha], the formalism is comprehensive enough to model a well-known security hole in Java's type system [Saraswat 1997]. There are many points of comparison between their model and ours as defined in the appendix. Jensen *et al* model the dynamic state of the JVM as a set of loaded classes relating to each other by a web of typing relations. In contrast, we attempt to model the *epistemological* state of a proof linker as assertions about loaded classes (commitments) and pending dependency checks (obligations). Our formalization attempts to show that the epistemological state of a proof linker is always consistent with a given, static subtype hierarchy (i.e., the intended model) when a carefully designed linking strategy is followed. This work could be made more realistic by replacing the static intended model by a dynamic classloading model similar to that of Jensen *et al.* The exploration of how the correctness conditions can be re-established in the dynamic model is an area for future work.

Cardelli [Cardelli 1997] defined a formal model for type-safe, *static* linking in a simply-typed lambda calculus. Linking is characterized as a series of substitutions that preserve type safety invariants. Our

verify primitive corresponds to Cardelli's *intra-checking*, while our **endorse** and **resolve** primitives could be seen as an incremental version of *inter-checking*. Our approach differs substantially in the treatment of typing environments. In particular, we replace the notion of an import environment as an input to intra-checking by our notion of obligations produced as output. In essence, obligations represent a logical specification of all allowable typing environments for which a module intra-checks. This technique is key to our implementation of lazy, dynamic linking. The second distinction in the treatment of typing environments is that we replace the notion of export environments by the set of commitments produced during module verification. In this case, however, the replacement is essentially a direct encoding of the typing environment in logical form. To mention a last point of comparison, the correctness of Cardelli's model is dependent, though implicitly, on a specific ordering of substitution steps [Cardelli 1997, Lemma 3-3 & Section 6]. In our framework, the interaction between verification correctness and relative ordering of linking events are formalized explicitly as the three correctness conditions stated in section 3.

Building on the work of Cardelli, Glew and Morriset [Glew and Morrisett 1999] proposed the typed object file as an extension to Typed Assembly Language (TAL) [Morrisett et al. 1999]. TAL object programs are annotated with type information, to be type checked at link time. The typed object file provides a means for safe, modular type checking of separately compiled code units. As the authors note, however, this approach does not naturally extend to lazy, dynamic linking.

Another related line of research is the work of Drossopoulou *et al* on formalizing the Java notion of binary compatibility [Drossopoulou et al. 1998; Drossopoulou et al. 1998a; Drossopoulou et al. 1998b]. Binary compatibility prescribes conditions under which revisions of classes can be safely linked with code that has been checked against the original versions. Although the notion has bearing on type-safe, static linking, it does not directly address the issue of lazy, dynamic linking.

7. CONCLUSION

We have proposed a modular architecture for language environments in which a static verification phase is used in conjunction with lazy, dynamic linking. Our architecture preserves laziness while encapsulating the verifier. Consequently, we avoid the problems of current commercial architectures in which verification logic is delocalized and interleaved with the logic of loading and linking. The verifier becomes a standalone component that can be understood and validated separately.

Our architecture achieves modularity through the notion of proof linking. Rather than performing recursive loading and verification to deal with intermodule dependencies, the verifier formulates dependency checks as proof obligations that may be discharged later. These obligations are attached to specific linking events and represent preconditions to enable those events, if and when they become necessary. We have formally characterized the conditions for the correct implementation of proof linking, and demonstrated the application of these conditions to an implementation of Java proof linking.

Our modularization could potentially be used to support several alternative architectures for mobile-code verification. These include the use of third-party verifier "plug-ins", remote verification service providers, and heterogeneous systems relying on the interoperability of verification protocols.

ACKNOWLEDGMENTS

We thank Prem Devanbu for introducing us to the wonderful world of Java bytecode verification, Andrew Walenstein for many helpful discussions, and the anonymous referees of FSE-6 and this journal for their very constructive comments on earlier versions of this paper.

APPENDIX

A. FORMAL VERIFICATION OF JAVA PROOF LINKING

The proof of safety, monotonicity, and completion as presented informally in section 4.3 can be formally verified using a theorem prover. In particular, we have formally established the above properties using the PVS specification and verification system [SRI Computer Science Laboratory]. This appendix reports on the exercise, in order (1) to demonstrate that the verification of safety, monotonicity, and completion can be performed rigorously with the help of a theorem prover, (2) to illustrate the specification and proof techniques that are found to be helpful in such an endeavor, and (3) to highlight the improved understanding of proof linking that may be gained as a result.

A.1 Intended Model

Before one can specify and prove theorems about the correctness of proof linking, one has to define an *intended model* for the first-order theory that is used in proof linking. Specifically, one has to define the intended meaning for predicate symbols "`extends`", "`subclass`", and so on. To this end, one must give a specification of the class hierarchy in a way that captures not only properties that the modular verifier enforces, but also the potential anomalies that could arise if proof linking is not performed properly.

For instance, a modular verifier can guarantee that the class `java/lang/Object` has no immediate superclass, that interfaces have `java/lang/Object` as their only direct superclass, and that all other classes have a unique direct superclass. However, confined to examine one code unit at a time, a modular verifier cannot rule out the possibility of circular subclassing (i.e., two classes being subclasses of each other) and subclassing from an interface class. Such anomalies must be made possible in our specification of the intended model. To capture these, we specify the following:

—`class` is a non-empty type.

—`java_lang_object` is a distinguished object of type `class`. All other `class` objects have type `(non_root_class?)`.

—The set `(non_root_class?)` is further partitioned into two subsets, `(interface?)` and `(proper_class?)`.

—We postulate that there is a function `extends : [(proper_class?) -> class]` that maps a non-interface class to its unique, direct superclass. Notice that circularity and subclassing from an interface class is thus allowed.

—As an "extension" to the function `extends`, we define a predicate `direct_super_class? : [class -> class -> bool]` to capture the facts that `java/lang/Object` has no superclass and that interfaces extend `java/lang/Object`.

—A predicate `super_class? : [class -> class -> bool]` is defined to be the transitive closure of `direct_super_class?`.

Other notions like subinterfacing, subclassability, and so on are completely specified according to what the modular verifier enforces and allows. This part of the specification documents the capabilities and limitations of the modular verifier.

A.2 Strategy

To specify strategies, we define an abstract datatype `primitive`. A PVS datatype declaration introduces constructors and accessors for each of the subtypes (e.g., `PRIM_verify(X)` is a constructor for the primitive "**verify** X"). We then define a binary relation `before` over `primitive` to represent the ordering as specified in section 4.1. Notice that the specification defines the linking strategy in terms of the intended model. For example, the Subtype Dependency Property requires `before(PRIM_verify(Y), PRIM_endorse_class(X))` to hold if `super_class?(X)(Y)` is true.

For the sake of clarity and specification economy, `before` is specified in the following manner. We capture each of Natural Progression Property, Import-Checked Property, Subtype Dependency Property and Referential Dependency Property in a separate relation. We then define another binary relation `Precede` as a union of the four. Also, we only specify the immediate precedence of primitives, and then define the binary relation `before` as a transitive closure of `Precede`.

When a new strategy is defined, it is imperative to check that it actually defines a strict partial ordering over the set of primitives. To illustrate this necessity, consider an alternative formulation of the Subtype Dependency Property, in which we require

$$\textbf{endorse}\ Y < \textbf{endorse}\ X$$

for any class X and its *direct* superclass or *direct* super interface Y. Despite the subtle difference, this formulation appears to have achieved everything we want a Subtype Dependency Property to achieve, namely, forcing all superclasses and superinterfaces of X to be verified before X is endorsed.

However, such a formulation also introduces an inconsistency: the resulting strategy is not a strict partial ordering. Recall that the modular verifier cannot rule out circular subclassing. In the case when X and Y are subclasses of each other, the above formulation places "**endorse** X" and "**endorse** Y" before each other, an impossibility if `before` is to be a strict partial ordering. It is through this articulation process that we have come to adopt our current formulation of Subtype Dependency Property instead of the alternative shown here.

To prove that `before` is a strict partial ordering, one has to show that it is transitive (which is trivial since `before` is defined as a transitive closure of `Precede`) and irreflexive. The latter can be shown by, firstly, assigning an (integer) ordinal number[6] to each primitive and, secondly, showing that the ordering of ordinals preserves the ordering of `before`. Irreflexivity follows since no integer is less than itself.

A.3 Database

An abstract datatype `predicate` is defined to capture the signature of the predicate symbols used in proof linking. A `database` is represented as a `set` of `predicate`. We also define a mapping `model : [predicate -> bool]` that correlates a predicate to the relation it denotes. For example, `model(PRED_extends(`X`, `Y`))` maps to the value of `direct_super_class?(`X`)(`Y`)`.

We encode Figure 5 by a relation `may_attach?(`q`)(`o`)(`p`)` that evaluates to true when `primitive` q could potentially attach `predicate` o to `primitive` p as an obligation. We also define the shorthand `potential_obligation?(`p`)(`o`)` to mean $(\exists\ (q) :$ `may_attach?`$(q)(o)(p))$. In addition, we represent Figure 4 by a relation `commit?(`p`)(`c`)` that evaluates to true when `primitive` p asserts `predicate` c as a commitment. Both `may_attach?` and `commit?` are defined in terms of the intended model. For example, `commit?(PRIM_verify(`X`))(PRED_extends(`X`, `Y`))` is true iff the relation `direct_super_class?(`X`)(`Y`)` holds in the intended model. This formally captures the condition under which the modular verifier generates a specific commitment or obligation. One can also sanity-check the definition of `commit?` by proving the following challenge using case analysis:

$$\begin{array}{l}\text{CONSISTENT\_COMMITMENT}: \textsc{lemma}\\ \quad (\forall\ (p: \text{primitive}, c: \text{predicate}): \text{commit?}(p)(c) \Rightarrow \text{model}(c))\end{array}$$

We define a `state` to be a `set` of `primitive`. Intuitively, a `state` describes the set of primitives that are already terminated at a certain point of the proof linking process. We then define `state_database` as a mapping from a `state` to a `database` that contains all the predicates committed by members of the given `state`. We also define `STATE_before : [primitive -> state]` as a mapping from a primitive to a state containing all the primitives that are guaranteed to have terminated before the initiation of the given primitive. As a result, the expression `state_database(STATE_before(`p`))` gives the database containing all commitments that are guaranteed to be available prior to the execution of a `primitive` p.

Query evaluation and the initial theory are captured by an inductively defined relation `provable? : [[predicate, database] -> bool]`, which captures whether a `predicate` is provable in a given `database`. For non-recursive queries, `provable?` simply checks if the predicate is an element of the database. For recursive queries, `provable?` unfolds the query inductively. For instance, `provable?(PRED_subclass(`X`, `Y`), `DB`)` is true iff the following is true:

$$\begin{array}{l}X = Y \vee \\ \quad (\exists\ (Z : \text{class}) : \\ \qquad \text{provable?}(\text{PRED\_extends}(X,\ Z), \text{db}) \wedge \\ \qquad \text{provable?}(\text{PRED\_subclass}(Z,\ Y), \text{db}))\end{array}$$

[6] PVS automatically defines an ordinal number for members of an abstract datatype.

To sanity-check the definition, and to prepare for proving completion, a generalization of the `CONSISTENT_COMMITMENT` lemma is verified:

$$
\begin{array}{l}
\text{SOUNDNESS}: \ \textsc{theorem} \\
\quad (\forall\ (o:\ \text{predicate}): \\
\quad\quad (\exists\ (S:\ \text{state}):\ \text{provable?}(o, \text{state\_database}(S))) \Rightarrow \text{model}(o))
\end{array}
$$

The above theorem says that, once an obligation is shown to be provable in a database, it means that the relation it denotes is true in the intended model. The theorem is established by induction, of which the `CONSISTENT_COMMITMENT` lemma serves as a base case.

All the definitions ready, we are now in the position to establish the correctness conditions.

A.4 Correctness Proofs

We skip the discussion of the safety condition, which can be checked by straightforward case analysis. Monotonicity is captured by the following theorem:

$$
\begin{array}{l}
\text{MONOTONICITY}: \ \textsc{theorem} \\
\quad (\forall\ (\text{DB1}:\ \text{database}, \text{DB2}:\ \text{database}): \\
\quad\quad (\text{DB1} \subseteq \text{DB2}) \Rightarrow \\
\quad\quad\quad (\forall\ (o:\ \text{predicate}):\ \text{provable?}(o, \text{DB1}) \Rightarrow \text{provable?}(o, \text{DB2})))
\end{array}
$$

The theorem says that, if an obligation is provable in a database, it stays provable even if we add more clauses into the database. This theorem can be proven by induction on the relation `provable?`. Strictly speaking, the above proof is not necessary because the use of Horn clauses trivially guarantees monotonicity (see section 3.3). Yet, the ability to establish monotonicity without relying on the syntactic form of the logic suggests that a more general form of logic may be used as long as the MONOTONICITY theorem can be proven.

The completion condition can be represented in the following theorem:

$$
\begin{array}{l}
\text{COMPLETION}: \ \textsc{theorem} \\
\quad (\forall\ (p:\ \text{primitive}, o:\ \text{predicate}): \\
\quad\quad \text{potential\_obligation?}(p)(o) \Rightarrow \\
\quad\quad\quad (\exists\ (S:\ \text{state}):\ \text{provable?}(o, \text{state\_database}(S))) \Rightarrow \\
\quad\quad\quad\quad \text{provable?}(o, \text{state\_database}(\text{STATE\_before}(p))))
\end{array}
$$

To understand the above formulation, recall that Completion requires that, if an obligation check fails, then subsequently asserted commitments cannot serve to establish it. Conversely, if an obligation can ever be established, it must be satisfied at the time of checking. Formally, if an obligation o can be attached to a primitive p, and if o is provable in some database, then it must be provable immediately prior to the execution of p.

Completion is by far the most challenging proof we have attempted. In order to establish the above theorem, the completeness of the query evaluation procedure is first established by induction and by applying the `MONOTONICITY` theorem:

$$
\begin{array}{l}
\text{COMPLETENESS}: \ \textsc{lemma} \\
\quad (\forall\ (p:\ \text{primitive}, o:\ \text{predicate}): \\
\quad\quad \text{potential\_obligation?}(p)(o) \Rightarrow \\
\quad\quad\quad \text{model}(o) \Rightarrow \text{provable?}(o, \text{state\_database}(\text{STATE\_before}(p))))
\end{array}
$$

The COMPLETENESS theorem says that, if an obligation o can be attached to a linking primitive p, and if the relation denoted by o is true in the intended model, then o must become provable before p is executed. It is easy to see that the `SOUNDNESS` theorem and the `COMPLETENESS` theorem together imply `COMPLETION`.

The idea behind the above proof is to show that the dynamically evolving commitment database is always consistent with a static typing model. Since the relations defined in the static model are invariant, and since the commitments always mirror these static relations, subsequently asserted commitments never contradict each other. Establishing consistency between sets of clauses by finding a shared model is a standard technique in formal logic.

REFERENCES

Bracha, G. A critique of *Security and Dynamic Loading in Java: A Formalisation*. Available at http://java.sun.com/people/gbracha/critique-jmt.html.

Cardelli, L. 1997. Program fragments, linking, and modularization. In *Proceedings of the 24th Annual ACM Symposium on Principles of Programming Languages (POPL'97)* (Paris, France, January 1997). Also available at http://research.microsoft.com/Users/luca/Papers/Linking.ps.

Carzaniga, A., Picco, G. P., and Vigna, G. 1997. Designing Distributed Applications with Mobile Code Paradigms. In *Proceedings of the 19th International Conference on Software Engineering (ICSE'97)* (Boston, MA, May 1997).

Colby, C., Lee, P., Necula, G. C., Blau, F., Cline, K., and Plesko, M. 2000. A certifying compiler for Java. In *Proceedings of the 2000 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'00)* (Vancouver, BC, Canada, June 2000). Also available as http://www-nt.cs.berkeley.edu/home/necula/public_html/pldi00b.ps.gz.

Dean, D. 1997. The security of static typing with dynamic linking. In *Proceedings of the Fourth ACM Conference on Computer and Communication Security* (Zurich, Switzerland, April 1997). Also available at http://www.CS.Princeton.EDU/sip/pub/ccs4.html.

Dean, D., Felten, E. W., and Wallach, D. S. 1996. Java security: From HotJava to Netscape and beyond. In *Proceedings of the 1996 Symposium on Security and Privacy* (Oakland, California, May 1996).

Devanbu, P. T., Fong, P. W. L., and Stubblebine, S. G. 1998. Techniques for trusted software engineering. In *Proceedings of the 20th International Conference on Software Engineering (ICSE'98)* (Kyoto, Japan, April 1998). Also available at http://seclab.cs.ucdavis.edu/~devanbu/files/icse98.pdf.

Drossopoulou, S., Eisenbach, S., and Wragg, D. 1998a. A fragment calculus — towards a model of separate compilation, linking and binary compatibility. Technical Report 98/13, Department of Computing, Imperial College, 180 Queen's Gate, LONDON SW7 2BZ, England. Also available at http://outoften.doc.ic.ac.uk/projects/slurp/papers.html#esop.

Drossopoulou, S., Eisenbach, S., and Wragg, D. 1998b. What can Java binary compatibility mean? Technical Report 99/1, Department of Computing, Imperial College, 180 Queen's Gate, LONDON SW7 2BZ, England. Also available at http://outoften.doc.ic.ac.uk/projects/slurp/papers.html#whatcan.

Drossopoulou, S., Wragg, D., and Eisenbach, S. 1998. What is Java binary compatibility? In *OOPSLA'98* (1998). Also available at http://outoften.doc.ic.ac.uk/projects/slurp/papers.html#bcoopsla.

Glew, N. and Morrisett, G. 1999. Type-safe linking and modular assembly language. In *Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'99)* (San Antonio, Texas, January 1999). Also available at http://www.cs.cornell.edu/talc/papers/mtal.pdf.

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

JavaSoft. Java security: Chronology of security-related bugs. Available at http://java.sun.com/sfaq/chronology.html.

JavaSoft. 1997. *Inner Classes Specification*. Available at http://java.sun.com/products/jdk/1.1/docs/guide/innerclasses/.

Jensen, T., Métayer, D. L., and Thorn, T. 1998. Security and dynamic loading in Java: A formalisation. In *Proceedings of the 1998 IEEE International Conference on Computer Languages* (Chicago, Illinois, May 1998), pp. 4–15. Also available at ftp://ftp.irisa.fr/local/lande/tjdlmtt-iccl98.ps.gz.

Kam, J. B. and Ullman, J. D. 1977. Monotone data flow analysis frameworks. *Acta Informatica 7*, 305–317.

Kimera Team. Security flaws in Java implementations. Available at http://kimera.cs.washington.edu/flaws/.

Letovsky, S. and Soloway, E. 1986. Delocalized Plans and Program Comprehension. *IEEE Software*, 41–49.

Liang, S. and Bracha, G. 1998. Dynamic class loading in the Java virtual machine. In *Proceedings of the 1998 ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA'98)* (Vancouver, British Columbia, October 1998), pp. 36–44. Also available at http://java.sun.com/people/gbracha/classloader.ps.

Lindholm, T. and Yellin, F. 1999. *The Java Virtual Machine Specification* (2nd ed.). Addison Wesley. Also available at http://java.sun.com/docs/books/vmspec.

Lloyd, J. W. 1987. *Foundations of Logic Programming* (2nd, extended ed.). Symbolic computation: Artificial intelligence. Springer-Verlag.

McCullough, D. 1987. Specification for Multi-Level Security and a Hook-Up Property. In *Proceedings for the IEEE Symposium on Security and Privacy* (1987), pp. 161–166.

McCullough, D. 1988. Noninterference and the Composability of Security Properties. In *Proceedings of the IEEE Symposium on Security and Privacy* (1988), pp. 177–186.

McGraw, G. and Felten, E. W. 1997. *Java Security: Hostile Applets, Holes, and Antidotes*. John Wiley & Sons, Inc.

Morrisett, G., Walker, D., Crary, K., and Glew, N. 1999. From system F to typed assembly language. *ACM Transactions on Programming Languages and Systems 21*, 3 (May), 85–97. Also available at http://www.cs.cornell.edu/talc/papers/tal-toplas.pdf.

Necula, G. C. 1997. Proof-carrying code. In *Proceedings of the 24th ACM Symposium on Principles of Programming Languages (POPL'97)* (Paris, France, January 1997). Also available at http://www-nt.cs.berkeley.edu/home/necula/public_html/popl97.ps.gz.

Princeton Secure Internet Programming Group. History of the group. Available at http://www.cs.princeton.edu/sip/history.

RUGABER, S., STIREWALT, K., AND WILLS, L. M. 1996. Understanding interleaved code. *Automated Software Engineering 3*, 1–2 (June), 47–76.

SARASWAT, V. 1997. Java is not type-safe. Available at `http://www.research.att.com/~vj/bug.html`.

SRI COMPUTER SCIENCE LABORATORY. The PVS specification and verification system. Available at `http://pvs.csl.sri.com/`.

WEIDE, B. W. AND HOLLINGSWORTH, J. E. 1992. Scalability of reuse technology to large systems requires local certifiability. In *Proceedings of the 5th Annual Workshop on Software Reuse* (1992).