

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

Philip W. L. Fong  
School of Computing Science  
Simon Fraser University, B.C., Canada  
`pwfong@cs.sfu.ca`

Robert D. Cameron  
School of Computing Science  
Simon Fraser University, B.C., Canada  
`cameron@cs.sfu.ca`

April 1, 1999

SFU CMPT TR 1999-02  
`ftp://fas.sfu.ca/pub/cs/techreports/1999/`

## Abstract

Although mobile code systems typically employ run-time code verifiers to secure 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.

## 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 [2]. 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 [10] 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 [15] in order to ensure that it cannot affect the host machine in an undesirable way. As some authors have pointed out, Java security is type safety [18]. As long as the bytecode obeys the typing rules of Java, a mobile code unit is guaranteed to behave well.

Relying on a run-time verifier to secure 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 [22, 11, 12]. 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 may become manifest in two problematic architectural features of the run-time system.

1. **Interleaved logic.** Typical Java implementations interleave bytecode verification and loading. Java programs are composed of classes, each being loaded into the JVM separately. In the middle of verifying a class *C*, a new class *D* may need to be loaded in order to provide enough information for the verification of *C* 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 typical JVM implementations.
2. **Delocalized implementation.** Typical bytecode verifiers for Java have a four-pass architecture. Pass one is the verification logic performed by the loader, as we have briefly mentioned. Passes two and three involve internal verification performed by the verifier proper at link time. 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 [21, 14]. This so-called “scattershot security” [18] 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. **Completeness:** 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 *completeness*. We argue that a well-designed mobile code architecture should achieve the goals of completeness 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 process of formulating and discharging verification obligations in this way.

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

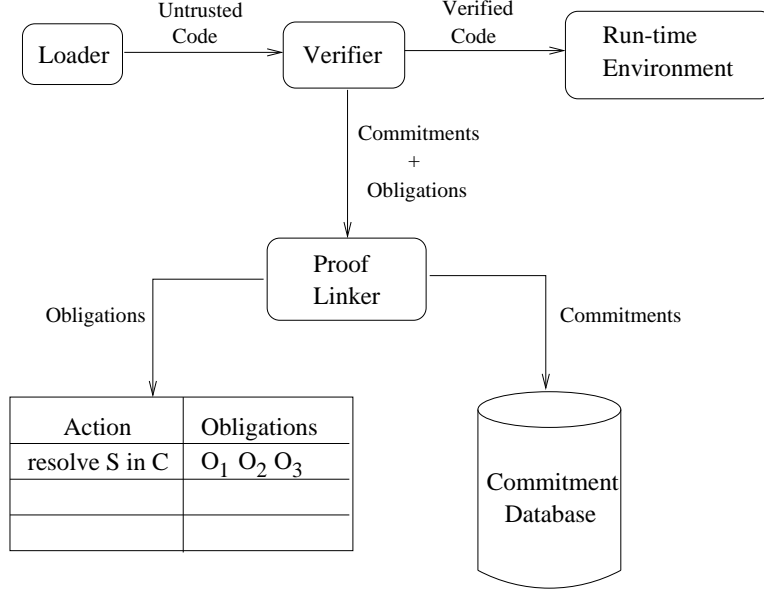


Figure 1: Modular Verification

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 be 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

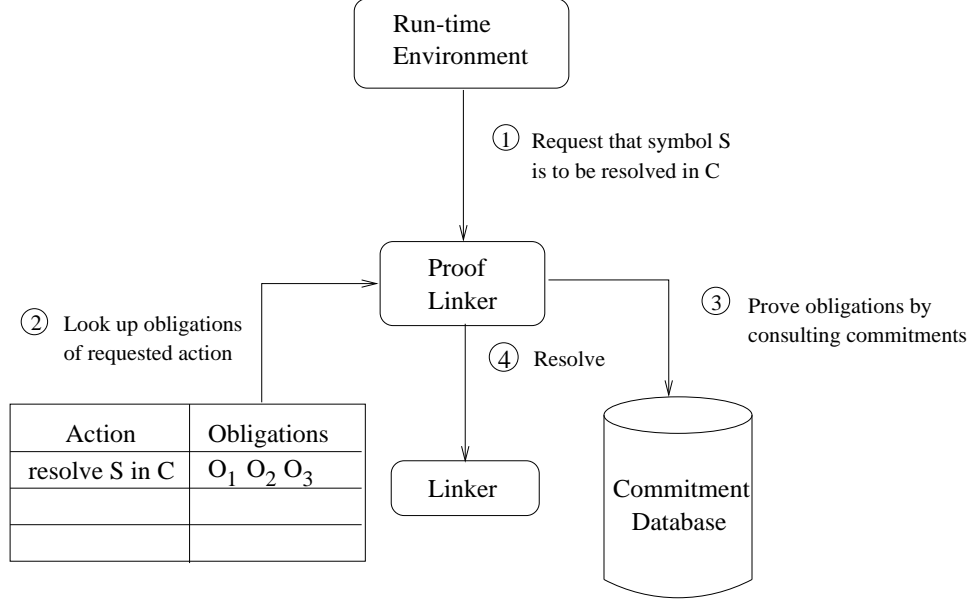


Figure 2: Proof Linking

recursively verifying (or even loading) the other code unit, the 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<sup>1</sup> `?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 `C`”<sup>2</sup>. 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. Each proof obligation is said to be *attached* to its associated action. The proof linker stores all obligations in a global *obligation table*, which provides a 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 gener-

<sup>1</sup>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 (“!”).

<sup>2</sup>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.

ates a commitment `!extends(ArithmeticException, Exception)` to indicate that `Exception` is the immediate superclass of `ArithmeticException`. The generated commitments are 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 a *proof linker*. The proof linker looks up 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 queries of the following form can be supplied to capture the transitive closure of 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 the deductive database model as a means of representing obligations and commitments, we do not propose that an actual system be implemented this way. Such an implementation would likely be unacceptably slow, as loading and linking in a mobile code system occurs frequently. Given queries and commitments of fixed signatures, and given a fixed initial theory, commitment assertion and obligation verification may be optimized for efficiency. In this case, the committed facts of a code unit can be encoded inside the code unit itself. The initial theory and the query mechanism may be programmed procedurally. For example, to make it efficient to check if one class is a subclass of another, a class can maintain a pointer to its immediate superclass, so that subclassing can easily be checked by pointer chasing.

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 fragment are checked before that fragment is executed.
2. **Monotonicity:** Once an obligation is checked, no subsequently asserted commitment will contradict it.
3. **Completeness:** All commitments that may be needed to satisfy an obligation are generated before the obligation is checked. That is, safety of a program will not be prematurely ruled out.

Our goal is to establish a set of properties to which an implementation of proof linking must conform in order to ensure that these correctness conditions hold.

#### 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, 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 proof linking as an algorithm over a set of *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 action  $p$ . These events occur asynchronously as the run-time system perform various linking actions. We assume that events are then queued up in some synchronized event queue, waiting to be examined by the proof linker. 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 before it.

Given a set  $P$  of linking primitives, a linking strategy  $\sigma = (P, \leq)$  is a 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 are fired by the run-time system. More precisely, an execution trace  $\tau$  is  $\sigma$ -conforming if the following hold: (1) all linking events in  $\tau$  initiate or terminate primitives from  $P$ , and (2) if “**begin**  $q$ ”  $\in \tau$ , then, for all  $p \in P$  such that  $p \leq_\sigma q$ , “**end**  $p$ ” occurs in  $\tau$  before  $q$ . To say that a run-time system implements a linking strategy  $\sigma$  is to say that the run-time system guarantees that all possible execution traces are  $\sigma$ -conforming. Notice that this definition of linking strategy allows primitives to be executed concurrently as long as the strategy do not explicitly order them.

A linking strategy  $\pi = (P, \leq_\pi)$  is a *substrategy* of another linking strategy  $\sigma = (P, \leq_\sigma)$  iff  $x \leq_\sigma y$  implies  $x \leq_\pi y$  for every  $x, y \in P$ . In English, a substrategy is a refinement of its superstrategy, the latter imposing fewer ordering constraints on the primitives.

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:**

**load**  $X$  < **verify**  $X$  < **resolve**  $S$  in  $X$

2. **Import-Checked Property:**

**verify**  $Y$  < **resolve**  $S$  in  $X$  < **use**  $S$  in  $X$

We shall only consider admissible strategies hereafter.

### 3.2 The Proof Linking Algorithm

The successful completion of every linking primitive generates two sets. The first is a set of facts (i.e., ground atoms) called *commitments*. Commitments describe the information collected as a result of executing a primitive. The second is a set of *guards*. A guard is an ordered pair of a linking primitive and a conjunctive set of ground queries. The queries are said to be the *obligations* of the associated linking primitives. The obligations state the preconditions that must be met before the associated primitive can safely be executed. Notice that obligations can be attached to primitives 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 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 (`InitialTheory`), 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, its associated obligations are retrieved from the obligation table (line 7). The verification of these queries is then attempted against the logic program in the commitment database (line 8). If the obligations are not satisfied, an exception is raised (line 11) to indicate that the linking primitive is unsafe. Alternatively, when the **end** event of a primitive is removed from the event queue, the commitments and



---

**algorithm** ProofLinker(InitialTheory):

```
01: DB  $\leftarrow$  InitialTheory; Obligations[]  $\leftarrow$   $\emptyset$ ;
02: Ready  $\leftarrow$   $\emptyset$ ; Checked  $\leftarrow$   $\emptyset$ ;
03: while ( $\neg$  terminated()) do
04:    $e \leftarrow$  get-next-event();
05:   switch  $e$  of
06:     case “begin  $p$ ”:
07:       for all  $o \in$  Obligations[ $p$ ] do
08:         if (DB  $\vdash$   $o$ ) then
09:           Checked  $\leftarrow$  Checked  $\cup$  {  $o$  };
10:         else
11:           raise exception in the thread that generated event  $e$ ;
12:         end if
13:       end for
14:       Ready  $\leftarrow$  Ready  $\cup$  {  $p$  };
15:     case “end  $p$ ”:
16:       DB  $\leftarrow$  DB  $\cup$  get-commitments( $p$ );
17:       for all  $\langle o, q \rangle \in$  get-guards( $p$ ) do
18:         Obligations[ $q$ ]  $\leftarrow$  Obligations[ $q$ ]  $\cup$  {  $o$  };
19:       end for
20:     end switch
21: end while
```

---

Figure 3: The Proof-Linker Model Algorithm

guards for the primitive are collected. The commitments are added to the commitment database (line 16). The guards associate new obligations to linking primitives. These new associations are incorporated into the obligation table (lines 17–19). 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 two auxiliary variables into the listing in Figure 3. “**Checked**” (lines 2 and 9) denotes the set of obligations that have already been checked at line 8. “**Ready**” (lines 2 and 14) denotes the set of primitives that are ready for execution.

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

1. **Safety:** All obligations are checked before a primitive is executed. For any linking primitives  $x$  and  $y$ , if  $x$  may introduce the guard  $\langle o, y \rangle$ , then we require that  $x <_{\sigma} y$ . That is, the following invariant must hold at all times:

$$\forall p \in \text{Ready} . \forall o \in \text{Obligations}[p] . o \in \text{Checked}$$

2. **Monotonicity:** Obligations may not be contradicted by subsequently asserted commitments. In our system, we confine our database to definite clause programs and definite queries<sup>3</sup>. This avoids the problems that could arise if negation by failure were used. In summary, the following invariant must hold at all times:

$$\forall o \in \text{Checked} . \Gamma \vdash o$$

3. **Completeness:** A commitment  $c$  is said to support an obligation  $o$  if  $c$  is required to make  $o$  provable. If a linking primitive  $p$  may assert a commitment that supports  $o$ , and if  $o$  is attached to linking primitive  $q$ , then we require that  $p <_{\sigma} q$ . Thus, if an obligation  $o$  of primitive  $q$  is eventually provable, then it must be provable when “**begin**  $q$ ” is fired.

In summary, the correctness of proof linking depends on (1) the linking strategy  $\sigma$ , (2) the kind of logic we are using, and, (3) the commitments and obligations returned by each linking primitive. Notice that the framework does 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 superstrategies to substrategies, 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

---

<sup>3</sup>A definite clause, or a Horn clause, is a rule of the form  $A \leftarrow B_1, B_2, \dots, B_n$ , where  $n \geq 0$ , and  $A, B_1, B_2, \dots, B_n$  are all atoms (i.e. positive literals). A definite clause program (or a Horn clause program) is a finite set of definite clauses. A definite query (or definite goal) is a conjunction of atoms. For definition of these and other logic programming terms used in this paper, consult standard text book like [4].

discussed in previous sections. In particular, we develop an admissible strategy for Java proof linking and prove the correctness conditions for an implementation of this strategy.

## 4.1 The Java Linking Model

In Java, a class is a code unit. A Java identifier may refer to a class or a member of a class. A member (a field or a method) of a class is uniquely identified by both the member’s name and its descriptor (type signature). 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. Class symbols and member symbols are resolved separately. We denote the linking primitive that resolves, in class  $X$ , the member  $M$  of class  $Y$  with descriptor  $S$  as “**resolve**  $Y::M(S)$  **in**  $X$ ”, and we reserve the usual 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 symbolic references as ready for resolution. These primitives are introduced to impose a desirable ordering among other primitives; they do not correspond to any actual linking activities. In particular, complex obligations are attached to them, and supports for such obligations are then forced to be asserted before the auxiliary primitives are fired<sup>4</sup>.

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**  $X$

and also

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

3. **Subtype Dependency Property:** If  $Y$  is a superclass or a superinterface of  $X$  then

**verify**  $Y$  < **endorse**  $X$

4. **Referential Dependency Property:** Sometimes, the knowledge of a class  $Y$  is needed before we can correctly endorse a method  $X::M(S)$ . In such case, we say that  $Y$  is *relevant* to the endorsing of  $X::M(S)$ , and we require that

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

---

<sup>4</sup>The actual Java loading model further involves two additional primitives “**prepare**  $X$ ” and “**initialize**  $X$ ”. In principle, one can always extend the current framework to include them if such refinement starts to interact with verification; but since they do not, we ignore them to facilitate our analysis.

In English, we need to verify all relevant classes before we endorse a method symbol for resolution. To implement such ordering, relevant classes are identified during the execution of “**verify**  $X$ ”. The latter then asserts an auxiliary commitment “**!relevant**( $Y, X::M(S)$ )” to inform the run-time system of the required ordering of linking events.

## 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 capture Java verification passes 1 to 3. Such arrangement effectively eliminates any need for the verifier to invoke the loader, and thus completely removes all verification logic from the latter module. In principle, we could have formulated commitments and obligations related to the checking of resolution errors — the fourth pass of verification [15, chapter 5]. The effect of which is further elimination of delocalization and interleaving. Since such an exercise conceptually parallels our endeavor here, and has less of an impact on enabling other verification protocols (see section 5.2 and 5.3), we omit it to facilitate presentation.

## 4.3 Correctness of Java Proof Linking — A Proof Sketch

The above arrangement satisfies the sufficient conditions for correct proof linking:

1. **Safety:** Only “**verify**  $X$ ” generates obligations. According to figure 5, obligations are only attached to “**endorse**  $X$ ”, “**endorse**  $X::M(S)$ ”, “**resolve**  $Y$  in  $X$ ”, and “**resolve**  $Y::M(S)$  in  $X$ ”. In any case, the Natural Progression Property and the Import-Checked Property guarantee that the contributors of obligations are always fired before the primitives to which the obligations are attached.
2. **Monotonicity:** The initial theory and the obligations are all definite.
3. **Completeness:** Consider the obligation `?subclassible( $Y$ )` attached to “**endorse**  $X$ ” by “**verify**  $X$ ”. Supports of the obligation are asserted by all “**verify**  $Z$ ”, where  $Z$  is either  $Y$  or one of its superclass. 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  $X$ , and therefore the Subtype Dependency Property guarantees that  $Y$  and all its superclasses 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 is asserted by all “**verify**  $W$ ”, where  $W$  is  $Y$ ,  $Z$ , or one of their superclasses or superinterface. According to figure 5, both  $Y$  and  $Z$  are relevant to  $X::M(S)$  if the obligation is to be asserted. It then follows from the Referential Dependency Property that the superclasses and superinterfaces of  $Y$  and  $Z$  are already verified when the obligation is tested. Thus, all the supports are already present, and the obligation can be consistently established.

---

```

!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.
!extends(X, Y)
    Y is a direct superclass of X.
!implements(X, Y)
    Y is a direct superinterface of X.
!member(X, M, S)
    M with descriptor S is a member of X.
!public_member(X, M, S)
    The member M with descriptor S is public in X.
!protected_member(X, M, S)
    The member M with descriptor S is protected in X.
!private_member(X, M, S)
    The member M with descriptor S is private in X.
!default_member(X, M, S)
    The member M with descriptor S has default access in X.
!relevant(Y, X, M, S)
    Inform run-time environment to verify Y and all its superclasses and superin-
    terfaces before endorsing X::M(S).

```

---

Figure 4: Commitments that may be asserted by **verify** *X*

---

<code>?subclassible(Y)</code>	<b>Target:</b> endorse $X$ <b>Intention:</b> Direct superclass $Y$ of $X$ can be subclassed.
<code>?interface(Y)</code>	<b>Target:</b> endorse $X$ <b>Intention:</b> Direct superinterface $Y$ of $X$ should be an interface class.
<code>?class(Y)</code>	<b>Target:</b> resolve $Y$ in $X$ <b>Intention:</b> $Y$ should be a non-interface class.
<code>?interface(Y)</code>	<b>Target:</b> resolve $Y$ in $X$ <b>Intention:</b> $Y$ should be an interface class.
<code>?throwable(Y)</code>	<b>Target:</b> endorse $X::M(S)$ <b>Intention:</b> Relevant to $X::M(S)$ , class $Y$ is throwable.
<code>?subclass(Y, Z)</code>	<b>Target:</b> endorse $X::M(S)$ <b>Intention:</b> Both relevant to $X::M(S)$ , $Y$ is a subclass of $Z$ .
<code>?assignment_compatible(Y, Z)</code>	<b>Target:</b> endorse $X::M(S)$ <b>Intention:</b> Both relevant to $X::M(S)$ , $Y$ is assignment compatible with $Z$ .
<code>?member(Y, M, S)</code>	<b>Target:</b> resolve $Y::M(S)$ in $X$ <b>Intention:</b> $M(S)$ is a member of class $Y$ .
<code>?accessible_instance_member(Y, M, S, X, Z)</code>	<b>Target:</b> resolve $Y::M(S)$ in $X$ <b>Intention:</b> Asserted when a method $N(T)$ of $X$ is verified. It requires that, being relevant to $X::N(T)$ , $Z$ can be used to reference the member $Y::M(S)$ .
<code>?accessible_special_member(Y, M, S, X, Z)</code>	<b>Target:</b> resolve $Y::M(S)$ in $X$ <b>Intention:</b> Asserted when a method $N(T)$ of $X$ is verified. It requires that, being relevant to $X::N(T)$ , $Z$ can be used to reference the special member $Y::M(S)$ .

---

Figure 5: Obligations that may be asserted by **verify**  $X$

---

```

;; subclassible(C) : Is C allowed to have subclasses?

subclassible('java/lang/Object').
subclassible(C) :- class(C), non_final(C), extends(C, D), subclassible(D).

;; subclass(C, D) : Is C a subclass of D?

subclass(C, C).
subclass(C, E) :- extends(C, D), subclass(D, E).

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

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

;; superinterface(C, D) : Is C a superinterface of D?

superinterface(C, C).
superinterface(E, C) :- implements(C, D), superinterface(E, D).
superinterface(E, C) :- extends(C, D), superinterface(E, D).

;; accessible_instance_member(C, M, S, D, R) :
;;   Can one applies getfield/putfield/invokevirtual in a class D to
;;   a reference type R in order to access the member C::M(S)

accessible_instance_member(C, M, S, _, _) :- public_member(C, M, S).
accessible_instance_member(C, M, S, D, R) :- protected_member(C, M, S),
                                             subclass(R, D).
accessible_instance_member(C, M, S, _, _) :- private_member(C, M, S).
accessible_instance_member(C, M, S, _, _) :- default_member(C, M, S).

;; accessible_special_member(C, M, S, D, R) :
;;   Can one applies invokespecial in a class D to a reference type R
;;   in order to access the member C::M(S)

accessible_special_member(_, '<init>', _, _, _).
accessible_special_member(C, _, _, C, _).
accessible_special_member(C, M, D, _, _) :- private_member(C, M, D).
accessible_special_member(C, M, D, D, R) :-
    subclass(D, C), accessible_instance_member(C, M, S, D, R).

;; assignment_compatible(S, T) : Can S be assigned to T?

assignment_compatible(S, T) :- subclass(S, T).
assignment_compatible(S, T) :- superinterface(T, S).

```

---

Figure 6: The Initial Theory

Using similar argument, one can establish the completeness of the incremental proof procedure for all obligations.

We have formalized and checked the above reasoning using the PVS specification and verification tool [3]. For details, please consult the appendix.

## 4.4 Implementing Commitment and Obligation Generation

To debug our formulation, we have implemented a stand-alone Java bytecode verifier that generates the commitments and obligations in figure 4 and 5. Two implementation details are worth noting.

**Meet computation.** Java type checking involves dataflow analysis. In the Sun implementation of the dataflow analyzer, merging of two classes  $Z_1$  and  $Z_2$  yields their most specific common superclass [15]. Computation of this superclass involves the recursive loading of all superclasses of  $Z_1$  and  $Z_2$ . Our implementation avoids recursive loading by representing the result of merging algebraically 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  $Z_2$ . All the obligations will then be formulated in terms of these *meet expressions*<sup>5</sup>. 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 nevertheless preserves type safety.

**Arrays.** Java arrays are classes, and there are special type rules for handling them. However, all such rules are ultimately formulated in terms of type rules of ordinary classes. For example, an array of  $A$  is assignment compatible to an array of  $B$  only if  $A$  is assignment compatible to  $B$ , and an array of  $A$  is always compatible to `java.lang.Object` and `java.lang.Cloneable`. 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 cases when the **relevant** fact is committed. Specifically, if the **relevant** fact is not committed, less ordering, and thus more laziness, results. To optimize the linking process, a modular verifier may choose not to assert relevance commitments that may be unnecessary in a particular case. For example, if the modular verifier attempt to attach `?subclass( $X$ ,  $Y$ )` to “**endorse**  $Z$ ”, normally, both  $X$  and  $Y$  will need to be declared relevant. But in the case when  $X = Y$ , one knows that `?subclass( $X$ ,  $Y$ )` is trivially true. So, the obligation and its corresponding relevance commitment need not be generated in this case. Similar optimizations

---

<sup>5</sup>Such algebraic representation does not affect the termination of the dataflow analysis since only finitely many class symbols may appear in a method, and thus the underlying lattice is bounded [13].



can be adopted for obligations that involve other special case combinations of rules in the initial theory.

An implementation such as the above is an infrastructure for the work in [5]. An online demonstration of modular verification and commitment/obligation generation can be accessed at <http://www.cs.sfu.ca/~pwfong/personal/Research/Verifier>.

## 5 Rethinking Verification

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

### 5.1 Replaceable Verifier Module

Since the verifier is completely decoupled from the loader and the linker, it is then possible for the mobile code verification technology to evolve independently of the mobile code hosting technology. In the context of Java applet verification, one may conceive that the bytecode verifier is manufactured as a replaceable component that can be “plugged” into any browser equipped with a proof linker. Third party vendors can 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. We believe this business model may yield higher quality and more secure mobile code hosting environments.

### 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 and obligations, 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 [5] 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 exemplifies this protocol. Verification is performed dynamically whenever a class-file 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 [20]**: 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 [5]**: 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 [23] and the composability of security features was studied as early as the 1980's [16, 17]. 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.

Safe, dynamic linking of mobile code units has recently been a topic of considerable interest. Dean considered the problem of type-safe execution of a dynamically-linked mobile code unit, assuming that the unit was properly type-checked at compile-time. He showed that mobile code systems must enforce the property of consistent extension of typing environments in order to ensure safety. In essence, consistent extension means that type judgments determined at compile time remain valid in any extended environment that may be created at run time. In our work, consistency of type extension is achieved by the monotonicity condition.

Cardelli [1] 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 a more-or-less a direct encoding of the typing environment in logical form.

Building on the work of Cardelli, Glew and Morriset [9] propose the typed object file as an extension to Typed Assembly Language (TAL) [19]. TAL object programs may thereby be annotated with type information, to be type checked at load 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 [8, 6, 7]. Binary compatibility prescribes conditions under which modified classes can be safely linked with other classes importing the modified classes. 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 also makes it possible consider several alternative architectures for mobile-code verification. These include the use of third-party verifier “plug-ins”, remote verification service providers, and heterogenous systems relying on the interoperability of verification protocols.

## Acknowledgement

The research was funded in part by a scholarship and an operating grant from the Natural Sciences and Engineering Research Council of Canada. We thank the anonymous referees of FSE-6 for their comments on an earlier version of this paper.

## References

- [1] Luca Cardelli. 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>.
- [2] Antonio Carzaniga, Gian Pietro Picco, and Giovanni Vigna. Designing Distributed Applications with Mobile Code Paradigms. In *Proceedings of the 19th International Conference on Software Engineering (ICSE'97)*, Boston, MA, May 1997.
- [3] Computer Science Laboratory, SRI. The PVS specification and verification system. Available at <http://pvs.csl.sri.com/>.
- [4] Subrata Kumar Das. *Deductive Databases and Logic Programming*. Addison-Wesley, 1992.
- [5] Premkumar T. Devanbu, Philip W. L. Fong, and Stuart G. Stubblebine. 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/icse98.ps>.
- [6] Sophia Drossopoulou, Susan Eisenbach, and David Wragg. 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, 1998. Also available at <http://outoften.doc.ic.ac.uk/projects/slurp/papers.html#esop>.
- [7] Sophia Drossopoulou, Susan Eisenbach, and David Wragg. What can Java binary compatibility mean? Technical Report 99/1, Department of Computing, Imperial College, 180 Queen's Gate, LONDON SW7 2BZ, England, 1998. Also available at <http://outoften.doc.ic.ac.uk/projects/slurp/papers.html#whatcan>.
- [8] Sophia Drossopoulou, David Wragg, and Susan Eisenbach. What is Java binary compatibility? In *OOPSLA'98*, 1998. Also available at <http://outoften.doc.ic.ac.uk/projects/slurp/papers.html#bcoopsla>.

- [9] Neal Glew and Greg Morrisett. 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://simon.cs.cornell.edu/home/jgm/papers/mtal.pdf>.
- [10] J. Gosling, B. Joy, and G. Steele. *The Java Language Specification*. Addison-Wesley, 1996.
- [11] The Princeton Secure Internet Programming Group. History of the group. Available at <http://www.cs.princeton.edu/sip/History.html>.
- [12] JavaSoft. Java security: Chronology of security-related bugs. Available at <http://www.javasoft.com/sfaq/chronology.html>.
- [13] John B. Kam and Jeffrey D. Ullman. Monotone data flow analysis frameworks. *Acta Informatica*, 7:305–317, 1977.
- [14] Stanley Letovsky and Elliot Soloway. Delocalized Plans and Program Comprehension. *IEEE Software*, pages 41–49, May 1986.
- [15] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification*. Addison Wesley, 1997.
- [16] Daryl McCullough. Specification for Multi-Level Security and a Hook-Up Property. In *Proceedings for the IEEE Symposium on Security and Privacy*, pages 161–166, 1987.
- [17] Daryl McCullough. Noninterference and the Composability of Security Properties. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 177–186, 1988.
- [18] Gary McGraw and Edward W. Felten. *Java Security: Hostile Applets, Holes, and Antidotes*. John Wiley & Sons, Inc, 1997.
- [19] Greg Morrisett, David Walker, Karl Crary, and Neal Glew. From system F to typed assembly language. In *Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'98)*, pages 85–97, San Diego, CA., January 1998. Also available at <http://simon.cs.cornell.edu/home/jgm/papers/tal.ps>.
- [20] George C. Necula. 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.cs.cmu.edu/~necula/pop197.ps.gz>.
- [21] Spencer Rugaber, Kurt Stirewalt, and Linda M. Wills. Understanding interleaved code. *Automated Software Engineering*, 3(1–2):47–76, June 1996.
- [22] The Kimera Team. Security flaws in Java implementations. Available at <http://kimera.cs.washington.edu/flaws/>.
- [23] B. W. Weide and J. E. Hollingsworth. Scalability of Reuse Technology to Large Systems Requires Local Certifiability. In *Proceedings of the 5th Annual Workshop on Software Reuse*, 1992.

# A Formal Verification of Java Proof Linking

The proof of safety, monotonicity, and completeness 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<sup>6</sup> [3]. In this appendix, we report our experience of such an exercise, in order to (1) demonstrate that the verification of safety, monotonicity, and completeness can be performed rigorously with the help of a theorem prover, (2) illustrate the specification and proof techniques that are found to be helpful in such endeavor, and (3) highlight the improved understanding of proof linking we gained as a result.

## A.1 Model

Before one can specify and prove theorems about the correctness of proof linking, one has to define a *model* for the first-order theory that is used in proof linking. Specifically, one has to define the meaning for predicate symbols like “**extends**”, “**subclass**”, etc. Consequently, properties of the class hierarchy resulted from modular verification (before the introduction of proof linking) must be specified. Such specification has to capture not only properties 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 super class, that interfaces have `java/lang/Object` as their only direct super class, and that all other classes have a unique direct super class. 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 are subclasses of each other) and subclassing from an interface class. Such anomalies must be made possible in our specification of the 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 sets, `(interface?)` and `(proper_class?)`.
- Postulate that there is a function `parent : [(proper_class?) -> class]` that maps a non-interface class to its unique, direct super class. Notice that circularity and subclassing from interface class is thus allowed.
- In terms of the `parent` function, a predicate `direct_super_class : [class -> class -> bool]` is defined to capture the facts that `java/lang/Object` has no super class and that interfaces extends `java/lang/Object`.

Other notions like subclassability and relevance are completely specified according to what the modular verifier enforces and allows. As such, this part of the specification documents the behavior characteristics that the modular verifier must follow in order for the correctness results to follow.

---

<sup>6</sup>Developed at SRI, PVS employs higher order logic as a specification language. It also offers an interactive theorem prover to assists the development of proofs for theorems.

## 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. **verify**(*C*) is a constructor for the primitive “**verify** *C*”). 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 a model of class hierarchy such as the one we have just defined.

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 if it actually defines a partial ordering over the set of primitives. To illustrate this necessity, consider an alternative formulation of the Subtype Dependency Property, in which we require

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

for all class *X* and its *direct* super class 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 super classes and super interfaces of *X* to be verified before *X* is endorsed. However, such formulation also introduces inconsistency — the resulting strategy is not well defined. 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 order. It is through such articulation that we have come to adopt our current formulation of Subtype Dependency Property instead of the alternative we mentioned.

To prove that **before** is a strict order, 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<sup>7</sup> to each primitive and, secondly, show 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**(*C*, *D*)) maps to the value of **direct\_super\_class**(*C*)(*D*).

We represent figure 5 by a relation **may\_attach?**(*q*)(*P*)(*p*) : **bool** that evaluates to true when primitive *q* might attach predicate *P* to primitive *p*. In addition, we represent figure 4 by a relation **commit?**(*p*)(*P*) : **bool** that evaluates to true when primitive *p* asserts predicate *P* as a commitment. Unlike **may\_attach?**,

---

<sup>7</sup>PVS automatically defines an ordinal number for members of an abstract datatype.

the relation `commit?` is defined in terms of the class hierarchy model. For example, `commit?(verify(C))(PRED_extends(C, D))` holds iff `direct_super_class?(C)(D)` is true in the class hierarchy. This formally captures the condition under which the modular verifier generates a specific commitment. One can then sanity-check the definition by proving the following challenge using case analysis:

**CONSISTENT\_COMMITMENT : CHALLENGE**

$$(\forall (p : \text{primitive}) : \forall (P : \text{predicate}) : \text{commit?}(p)(P) \Rightarrow \text{model}(P))$$

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 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 if 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_subclassible(C))(DB)` is true iff  $C = \text{java\_lang\_object}$  or there is a class  $D$  that satisfies the following conjunction:

$$\begin{aligned} & \text{provable?}(\text{PRED\_class}(C))(DB) \wedge \\ & \text{provable?}(\text{PRED\_non\_final}(C))(DB) \wedge \\ & \text{provable?}(\text{PRED\_extends}(C, D))(DB) \wedge \\ & \text{provable?}(\text{PRED\_subclassible}(D))(DB) \end{aligned}$$

To sanity-check the definition, and to prepare for proving completeness, a generalization of the **CONSISTENT\_COMMITMENT** lemma is verified:

**MODEL : THEOREM**

$$\begin{aligned} & (\forall (P : \text{predicate}) : \\ & (\exists (S : \text{state}) : \text{provable?}(P)(\text{state\_database}(S))) \Rightarrow \text{model}(P)) \end{aligned}$$

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:

**MONOTONICITY : THEOREM**

$$\begin{aligned} & (\forall (\text{DB1} : \text{database}) : \\ & \quad \forall (\text{DB2} : \text{database}) : \\ & \quad (\text{DB1} \subseteq \text{DB2}) \Rightarrow \\ & \quad (\forall (P : \text{predicate}) : \text{provable?}(P)(\text{DB1}) \Rightarrow \text{provable?}(P)(\text{DB2}))) \end{aligned}$$



It can be proven by induction on the relation `provable?`.

The completeness condition can be represented in the following theorem:

**STRONG\_COMPLETENESS : THEOREM**

$$\begin{aligned}
& (\forall (p : \text{primitive}) : \\
& \quad \forall (q : \text{primitive}) : \\
& \quad \quad \forall (P : \text{predicate}) : \\
& \quad \quad \text{may\_attach?}(q)(P)(p) \Rightarrow \\
& \quad \quad (\exists (S : \text{state}) : \text{provable?}(P)(\text{state\_database}(S))) \Rightarrow \\
& \quad \quad \text{provable?}(P)(\text{state\_database}(\text{STATE\_before}(p))))
\end{aligned}$$

Completeness is by far the most challenging proof we have attempted. In order to establish the above theorem, the following weaker form of completeness is first established by induction and by applying the **MONOTONICITY** theorem:

**WEAK\_COMPLETENESS : LEMMA**

$$\begin{aligned}
& (\forall (p : \text{primitive}) : \\
& \quad \forall (q : \text{primitive}) : \\
& \quad \quad \forall (P : \text{predicate}) : \\
& \quad \quad \text{may\_attach?}(q)(P)(p) \Rightarrow \text{model}(P) \Rightarrow \text{provable?}(P)(\text{state\_database}(\text{STATE\_before}(p))))
\end{aligned}$$

It is then easy to see that the **MODEL** theorem and the **WEAK\_COMPLETENESS** theorem together imply **STRONG\_COMPLETENESS**.

This proof illustrates how incremental proof linking works. If an obligation is provable in some state, then the property it describes will hold in the class hierarchy. In turn, when a property holds in the underlying class hierarchy, the obligation that describes it will be provable before the proof linker attempts to check it. As a result, every potentially provable obligation will be provable when it is discharged by the proof linker.