Proof Linking

A Modular Verification Architecture for Mobile Code Systems

Philip W. L. Fong

pwlfong@cs.uregina.ca

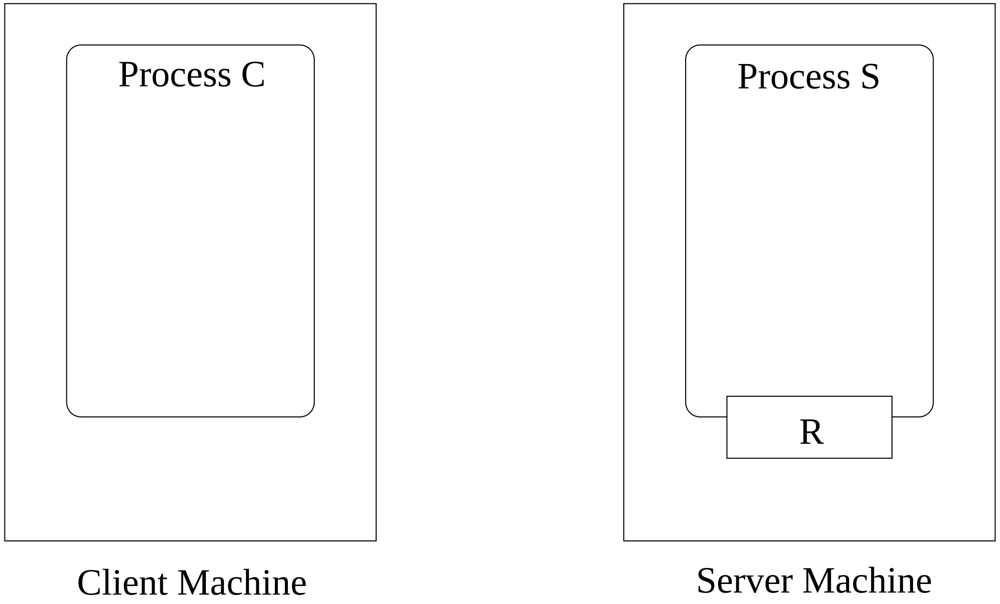
Department of Computer Science

University of Regina

Regina, Saskatchewan, Canada

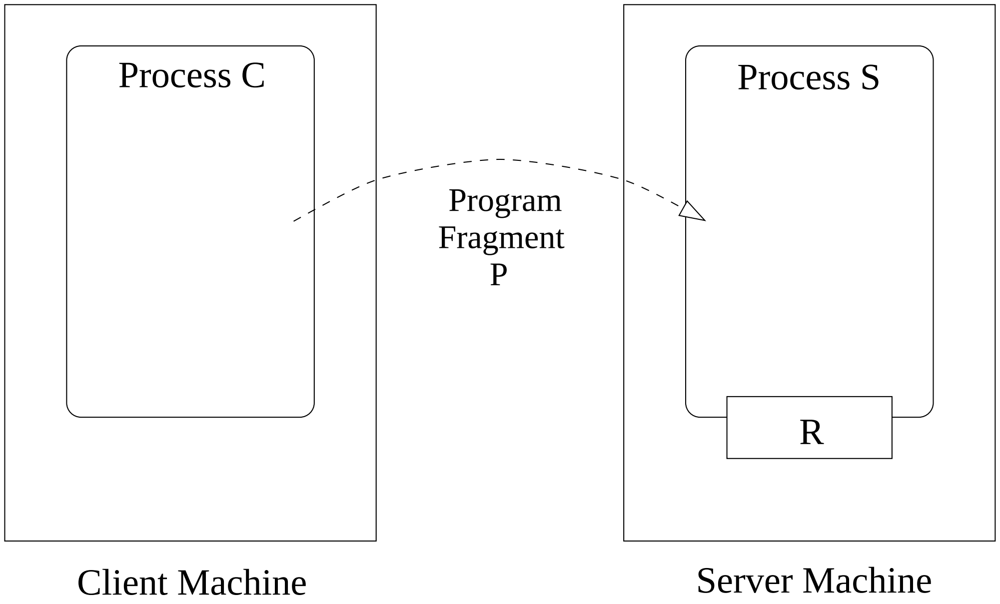
Mobile Code Systems

Code Mobility



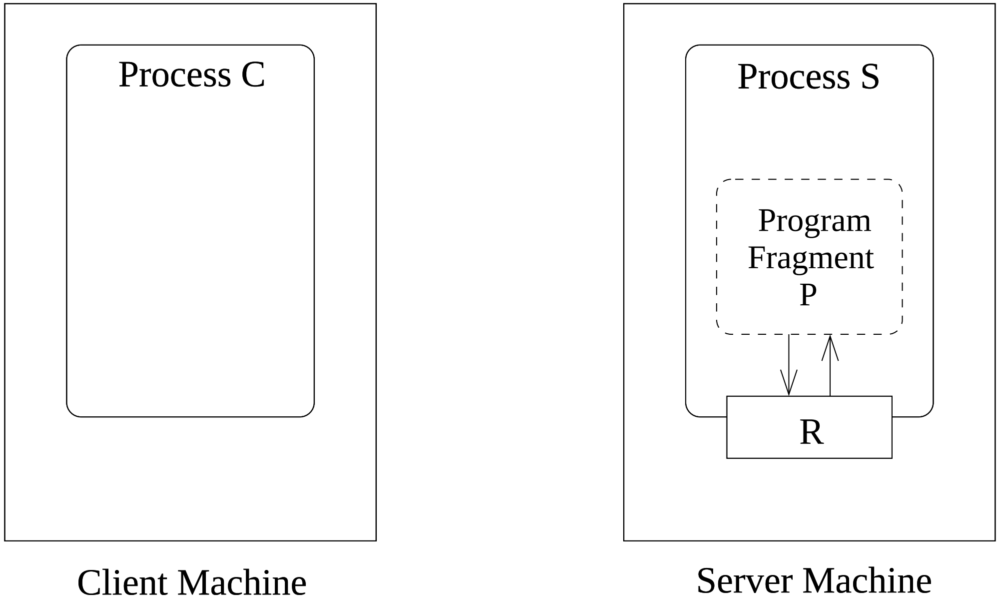
Mobile Code Systems

Code Mobility



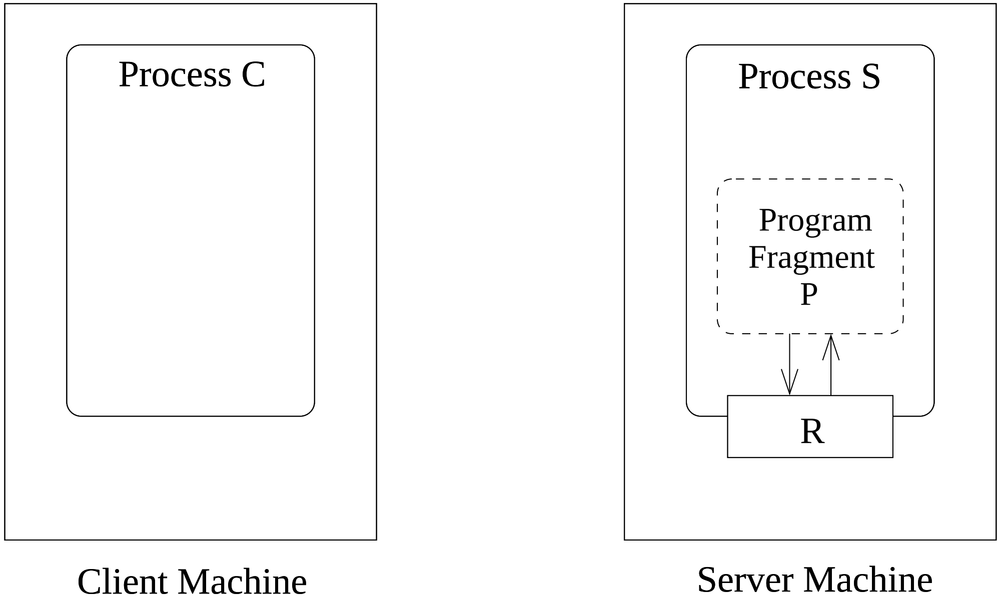
Mobile Code Systems

Code Mobility



Mobile Code Systems

Code Mobility



Examples

Postscript files
Active Disks
Active Packets
Java Applets

Java Virtual Machine

- JVM as an archetypical mobile code platform
- Java type safety

$$\text{Loader} \xrightarrow{\text{Classfile}} \text{Verifier} \xrightarrow{\text{Classfile}} \text{Runtime Env.}$$

- No type confusion $\Rightarrow$ Security manager protected

The JVM Verification Architecture

- Program safety is a whole-program notion:

  Intrachecking

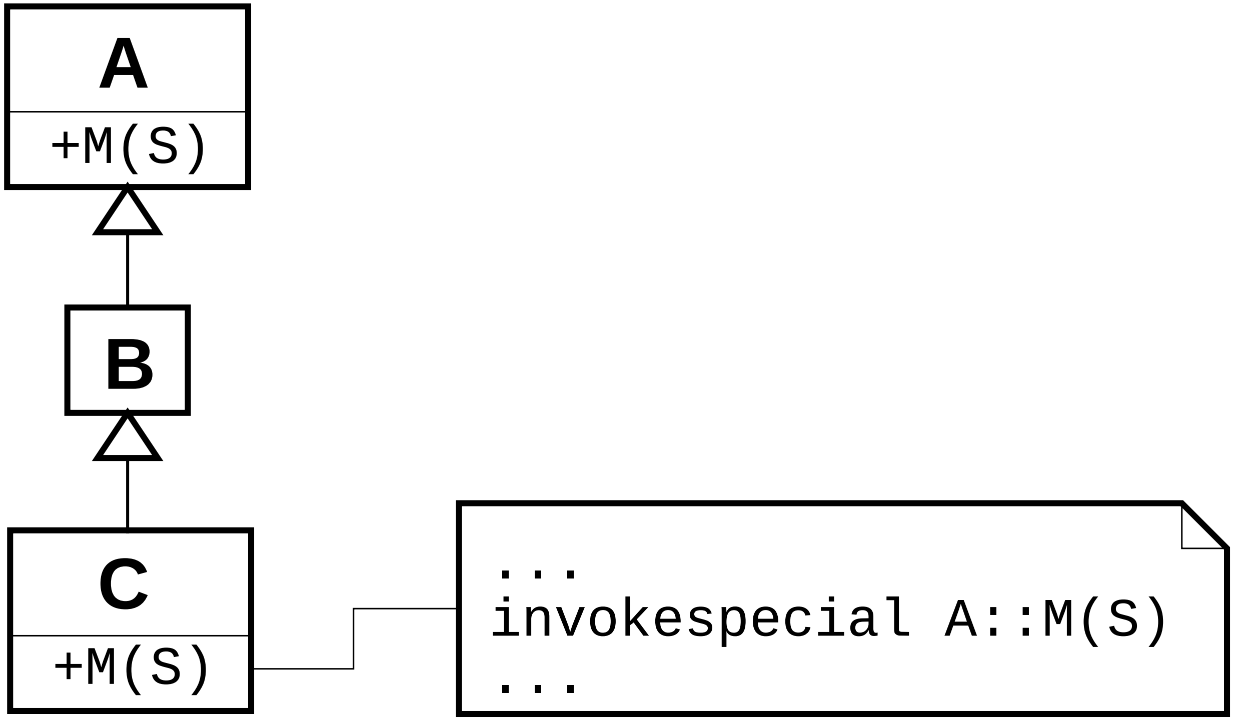
  Inferring the static properties of a classfile.

  Interchecking

  Checking that the inferred properties are compatible with the run-time environment.

- Interchecking and intrachecking are not cleanly separated in the JVM bytecode verifier.

Running Example



Need to show that C is a subclass of A.

Architectural Problems

Crux:

Lack of modularity: Tight coupling among loader, verifier and linker

Want:

1. Stand-alone verification modules
2. Distributed verification
3. Augmented type systems

The Proof Linking Architecture

1. **Modular verification:**

   Avoid the interchecking of external dependencies while intrachecking a code unit.

2. **Verification interface:**

   Record external dependencies in terms of proof obligations and commitments.

3. **Proof linking:**

   Incrementally discharge proof obligations at link time.

Verification Interface

Record external dependencies by a well-defined **verification interface**:

Verification Interface

Record external dependencies by a well-defined **verification interface**:

1. **Proof obligations:**

   Example:

   `subclass(`C`, `A`)`

Verification Interface

Record external dependencies by a well-defined **verification interface**:

1. **Proof obligations:**

   Example:

   `subclass(`C`, `A`)`

2. **Commitments:**

   Example:

   `extends(`C`, `B`)`

Verification Interface

Record external dependencies by a well-defined **verification interface**:

1. **Proof obligations:**

   Example:

   `subclass(`C`, `A`)`   **// resolve** A::$M(S)$ **in** C

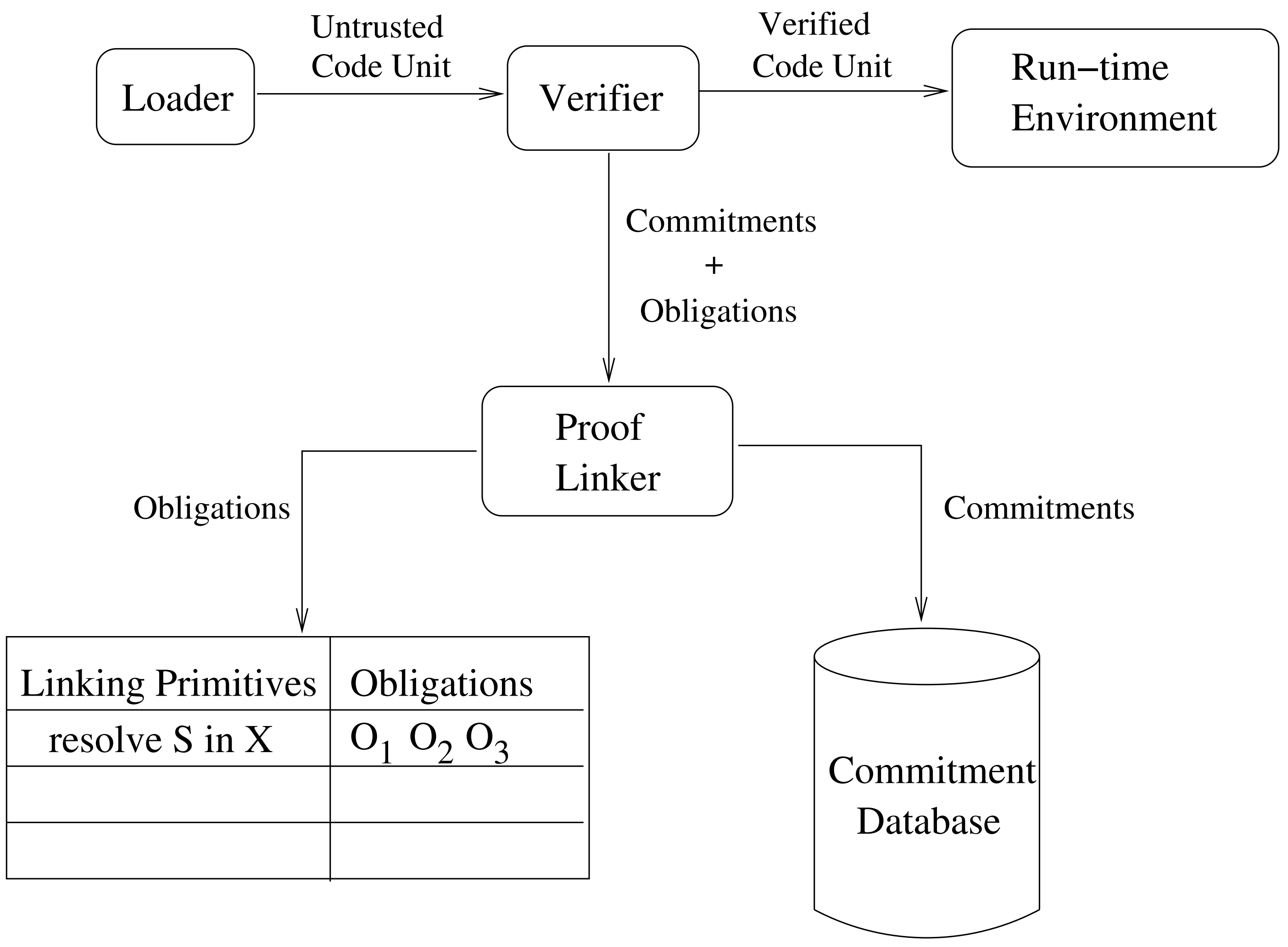
2. **Commitments:**

   Example:

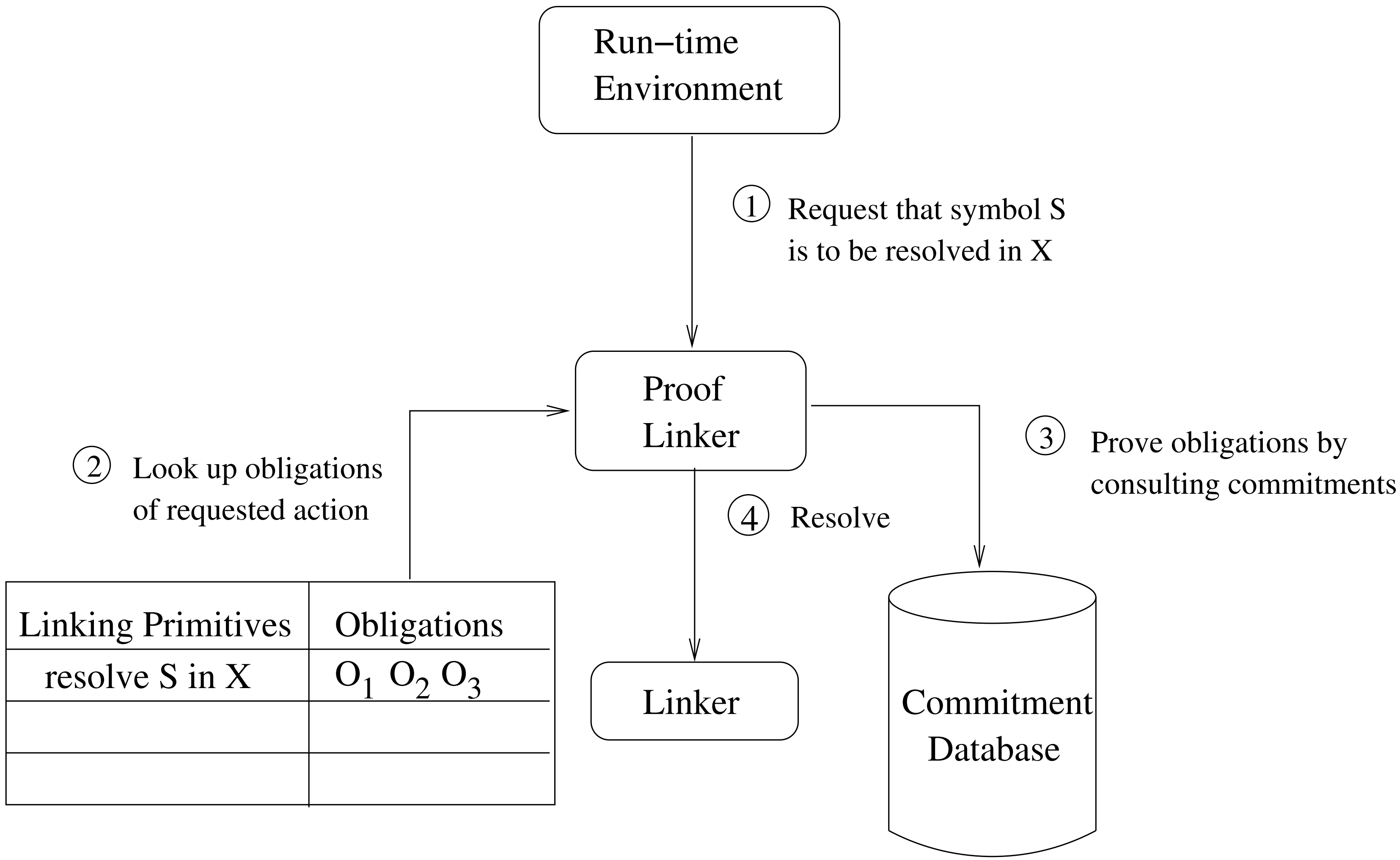
   `extends(`C`, `B`)`

3. **Obligation discharging schedule**

Modular Verification



Incremental Proof Linking



Initial Theory

- Arbitrary logic programs expressing type rules:

  Example:

  $\texttt{subclass(}X\texttt{,}\,X\texttt{)}$.

  $\texttt{subclass(}X\texttt{,}\,Y\texttt{)}$:-

  $\texttt{extends(}X\texttt{,}\,Z\texttt{)}$, $\texttt{subclass(}Z\texttt{,}\,Y\texttt{)}$.

- When coupled with commitments, Initial Theory allows proof linker to discharge proof obligations:

  Example:

  The following commitments

  $\texttt{extends(}C\texttt{,}\,B\texttt{)}$.

  $\texttt{extends(}B\texttt{,}\,A\texttt{)}$.

  allow us to discharge

  $\texttt{subclass(}C\texttt{,}\,A\texttt{)}$

Outline

1. *Modeling Adequacy and Soundness*

   The Proof Linking architecture can be instantiated to adequately model the semantic complexity of a production mobile code system, and to do so in a provably sound manner.

2. *Implementation Feasibility*

   The Proof Linking architecture can be feasibly realized to provide support for stand-alone verification modules, distributed verification and augmented type systems.

Modeling Adequacy and Soundness

Correctness Conditions

1. **Safety:**

   All obligations relevant to the safe execution of a linking primitive are generated and checked before that primitive is executed.

2. **Monotonicity:**

   Checked obligations may not be contradicted by subsequently asserted commitments.

3. **Completion:**

   All commitments that may be used for satisfying an obligation are generated before the obligation is checked.

Formal Modeling of Proof Linking

1. **Linking primitives**

load X	Acquire classfile X
verify X	Verify class X
endorse X	Endorse class X for resolution
endorse X::$M(S)$	Endorse member X::$M(S)$ for resolution
resolve Y **in** X	Resolve class symbol Y in class X
resolve Y::$M(S)$ **in** X	Resolve member symbol Y::$M(S)$ in class X

2. **Proof obligations and commitments**
3. **Initial theory**
4. **Linking strategy**

   Schedule of linking events in the form of a partial ordering on the linking primitives.

Linking Strategy

1. **Natural Progression Property**

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

2. **Import-Checked Property**

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

$$\textbf{endorse } Y < \textbf{endorse } Y{::}M(S) < \textbf{resolve } Y{::}M(S) \textbf{ in } X$$

3. **Subtype Dependency Property**

$$\textbf{verify } Y < \textbf{endorse } X \qquad \text{if } Y \text{ is a supertype of } X$$

4. **Referential Dependency Property**

$$\textbf{endorse } Y < \textbf{endorse } X{::}M(S) \qquad \text{if } Y \text{ is referenced in } X{::}M(S)$$

Establishing Correctness

Safety

Example:

verify $C \xrightarrow{\texttt{subclass}(C,A)}$ **resolve** $A{::}M(S)$ **in** C

Monotonicity

Use definite clause logic (aka Horn clauses).

Completion

Example:

1. $\texttt{subclass}(C, A)$ // **resolve** $A{::}M(S)$ **in** C
   1.1. $\texttt{extends}(C, B)$ // **verify** C
   1.2. $\texttt{subclass}(B, A)$
      1.2.1. $\texttt{extends}(B, A)$ // **verify** B
      1.2.2. $\texttt{subclass}(A, A)$

Correctness Results

- Established Safety, Monotonicity and Completion for a simplified model of Java dynamic linking *[FSE'98]*
- Formal verification by PVS *[TOSEM 9(4)]*
- Extension to account for multiple classloaders *[JVM'01]*

Implementation Feasibility

Implementation Efforts

- Aegis VM (`aegisvm.sourceforge.net`)
  - Reference implementation of Proof Linking
  - Open source JVM on GNU/Linux (x86)
- Three components
  - Generic proof linking framework
  - Stand-alone bytecode verifier
  - Pluggable Verification Modules (PVMs)
- Application
  - JAC – Java Access Control
- *UR CS TR 2003-11 (submitted for review)*

Generic Proof Linking Framework

- User-defined verification domains
  - Obligation discharging as native function dispatching
  - Pluggable Obligation Library
  - API for interrogating the internal state of the VM
- Standard representation of verification interface
  - Expressive obligation encoding scheme
- Correctness considerations
  - Safety and Completion guaranteed
- High fidelity to the Sun linking strategy

Pluggable Verification Modules

- An *extensible protection mechanism*
- Link-time bytecode verification is turned into a pluggable service that can be readily replaced, reconfigured and augmented.
- Application-specific verification services can be safely introduced into the dynamic linking process of the Aegis VM.
- Supports link-time enforcement of augmented type systems.

JAC – Java Access Control

Write-protecting the transitive state of an object:

```
public class List {
  public int data;
  public List next;
  public List(int data, List next) {
    this.data = data; this.next = next;
  }
}

readonly List x = new List(1, new List(2, null));
x.data = 5;      // Error: Writing to immediate state
x.next.data = 6; // Error: Writing to transitive state
```

JAC (Cont.)

```
public class Alice {
    public static void main(String[] args) throws Throwable
        List L = new List(1, new List(2, new List(3, null)));
        Class C = Class.forName(args[0]);
        Bob b = (Bob) C.newInstance();
        System.out.println(b.sum(L));
    }
}

public interface Bob {
    int sum(readonly List L);
}
```

JAC (Cont.)

```
public class Charlie implements Bob {
    public int sum(         List L) {
        int acc = 0;
        while (L != null) {
            if (L.next == null)  // corrupt last node
                L.data = 0;
            acc += L.data;
            L = L.next;
        }
        return acc;
    }
}
```

JAC (Cont.)

```
public class Charlie implements Bob {
    public int sum(readonly List L) {
        int acc = 0;
        while (L != null) {
            if (L.next == null)  // corrupt last node
                L.data = 0;
            acc += L.data;
            L = L.next;
        }
        return acc;
    }
}
```

Future Works

- Architectural constraints as security policies for software extensions
- Control flow constraints as proof obligations
- Aspect-oriented approaches to extensible protection mechanisms

Summary of Contributions

1. **The Proof Linking architecture**
   (a) Verification interface as *proof obligations* and *commitments*
   (b) The notion of *obligation discharging schedule*
2. **Modeling adequacy and soundness**
   (a) Correctness conditions: *Safety*, *Monotonicity*, *Completion*
   (b) Employing the notion of *linking strategy* to articulate correctness
   (c) *Formal verification* of Proof Linking for Java bytecode typechecking
3. **Implementation feasibility**
   (a) Reference implementation of Proof Linking in the *Aegis VM*
   (b) *Stand-alone bytecode verifier*
   (c) *Pluggable Verification Modules* as an extensible protection mechanism