Runtime support for type-safe dynamic Java classes

Scott Malabarba, Raju Pandey, Jeff Gragg, Earl Barr, and J. Fritz Barnes

Parallel and Distributed Computing Laboratory
Computer Science Department
University of California, Davis, CA 95616
http://pdclab.cs.ucdavis.edu/
{malabarb, pandey, gragg, barr, barnes}@cs.ucdavis.edu

Abstract. Modern software must evolve in response to changing conditions. In the most widely used programming environments, code is static and cannot change at runtime. This poses problems for applications that have limited down-time. More support is needed for dynamic evolution. In this paper we present an approach for supporting dynamic evolution of Java programs. In this approach, Java programs can evolve by changing their components, namely classes, during their execution. Changes in a class lead to changes in its instances, thereby allowing evolution of both code and state. The approach promotes compatibility with existing Java applications, and maintains the security and type safety controls imposed by Java's dynamic linking mechanism. Experimental analyses of our implementation indicate that the implementation imposes a moderate performance penalty relative to the unmodified virtual machine.

1 Introduction

Software systems must change over time. Changing business practices, the relentless advance of technology, and the demands of end users drive this evolution. The functionality required of applications inevitably changes in response to these factors. Consequently, in order to remain viable, applications must evolve to meet new requirements. Software component evolution is a major focus of effort in software engineering [27, 38].

The vast majority of commercial software is written in a few imperative languages, such as C++ or Java [3]. For these languages, software evolution is generally a slow, static process. Most of us are familiar with the process of waiting for the latest version of our favorite program to come out, stopping work to install the new version over the old one, then cleaning up the resultant mess of incompatible document formats and lost settings. The fact that a running program cannot be changed drives this cycle. Since any update requires stopping a program and overwriting all or part of it, incremental updates are often impractical, and major updates problematic. For a large class of critical applications, such as business transaction systems, telephone switching systems

and emergency response systems, the interruption poses an unacceptable loss of availability.

What is needed, then, is more support for applications that evolve *during execution.* In addition to supporting true evolution of software systems, dynamic evolution provides several other benefits:

- *Software distribution and management*: Dynamic evolution has applications in software distribution and management. Consider a distributed system in which changes in all active applications are either pulled or pushed from software servers to the active applications. While several applications, for instance NetscapeNavigator, MicrosoftInternet Explorer and RealAudioRealPlayer, currently support such application-specific updates, most use static updates for modifying applications.
- *Runtime optimization*: Often, specific properties of systems are best determined at runtime. For example, many applications can be highly optimized *if* some information about the input is known during development. However, these same optimizations result in specialized code restricted to a smaller input domain. If code can be modified at runtime, a program can accept a wider range of data, yet load and use methods optimized for the current data set.
- *Dynamic policy specification*: Dynamic evolution can be very useful in any application whose behavior is driven by a set of policies, for instance security policies. For example, dynamic security policies can be implemented using total mediation, without modifying code at runtime. This method requires a security check at every access of every resource [36]; due to the high performance cost, it is not widely used. Systems that employ total mediation implement dynamic policies by using general, static code to interpret dynamic data structures – a computationally expensive process. Dynamic evolution allows designers to move logic from interpreted data structures into directly executed code. This provides the efficiency of code-driven security enforcement [35] without sacrificing flexibility.

In this paper, we present an approach for dynamic evolution of Java programs. While Java [3] provides several mechanisms, such as inheritance, interfaces and dynamic linking, for program extensibility, it does not support true dynamic evolution, in which both the code and state of a program can evolve gracefully. In our approach, Java programs can evolve by changing their components, namely classes, during execution.

Java classes can be considered to have a life cycle with three discrete states: unloaded, or *static*, *loaded*, and *active.* A static class exists only in storage; it has not been loaded into the Java virtual machine. A loaded class has been loaded and possibly linked. Finally, an active class has live instances and/or methods running. We are concerned with changing active classes; a *dynamic class* can change while active.

We wished to preserve the syntax and semantics of the target language. Doing so ensures compatibility with existing code, and provides greater ease of use

as developers do not need to learn new language constructs. This constraint requires that we preserve the type safety characteristics of a program throughout its execution. Type safety encourages the development of safer, more disciplined code. In a dynamic system, type safety can restrict wild, unsound changes, alleviating the dangers inherent in changing code. Further, many of Java's security mechanisms, for instance separation of user and system name spaces and protection of private data, depend on the type-safe properties of Java programs. Therefore, we impose the restriction that all changes in a program preserve the type safety properties of the program. Section 2.1 presents our formal model, and defines valid class change. Using the formal model, we show that a valid class change preserves type safety. Further, in order to provide a convenient, backward-compatible interface, and to support changes in any Java class, we extended the Java class loader [29]. The new dynamic class loader allows a program to define a class multiple times. The dynamic class loader implements changes in a class, and any resulting changes in its instances, in an executing program. We describe the dynamic class loader in detail in Section 2.2.

Support for dynamic evolution, however, raises additional security issues, as malicious applets may use the dynamic class mechanism to modify the classes that enforce specific security policies of a host. Therefore, the dynamic class loader implements a security model that ensures that Java programs can dynamically modify only those resources to which they are authorized. We enforce this policy using name space separation and resource access control. We discuss the security model in Section 2.3.

We have implemented support for dynamic classes by modifying Sun's Java virtual machine (JDK 1.2). Dynamic classes can be implemented in several ways: by changing the language, through library-based support, or by modifying the virtual machine. As stated above, we did not wish to change the language. Library-based support proved to be too awkward and inefficient for our requirements. Thus, we chose to directly modify the virtual machine. Section 3 describes our implementation in detail.

We performed several experiments to measure the performance characteristics of our implementation. The experiments show that dynamic classes add about 6-10% of overhead to Sun's JVM. Further, the cost of updating classes is moderate. Section 4 presents these results, as well as further analysis with regard to alternative methods and related work.

2 Dynamic Classes

In this section, we formally describe the concept of dynamic classes. We begin by presenting a formal model of classes, objects, and inheritance in Java. We consider the potential effects of introducing dynamic classes into a running application. We then use our model to define type-safe dynamic classes. We discuss how best to support dynamic classes while addressing type safety and security issues, and describe our design. In general, we have made conservative

design choices, emphasizing compatibility with existing Java code, minimizing performance penalties, and maintaining type safety and overall system security.

2.1 Formal Model

We begin by formalizing the notion of classes, interfaces, inheritance, composition, and dependency among classes in Java. In doing so, we build upon the formal Java type model developed in [9], which includes type widening [8, 39].

Types, classes and objects: A type T denotes a set of objects. T is bound to a definition that describes the contents of the objects and operations that act on them. Specifically, this definition consists of T's *interface* and its *implementation*.

Definition 1. *(Type interface ($\mathfrak{i}(T)$)).* The interface of a type T, $\mathfrak{i}(\mathrm{T})$, is a set of public data fields and methods. $\square$

Definition 2. *(Type implementation ($\mathfrak{b}(T)$)).* The implementation, or body, of a type T, $\mathfrak{b}(\mathrm{T})$, is a set of private data fields and a set of method bodies. $\square$

Definition 3. *(Interface).* The Java interface construct describes, but does not implement, a type. An interface contains no data fields. $\square$

Definition 4. *(Class).* A class describes and implements a type. Thus, a class C is defined by the tuple $<\mathfrak{i}(\mathrm{C}), \mathfrak{b}(\mathrm{C})>$, where $\mathfrak{b}(\mathrm{C})$ contains implementations for all methods declared in $\mathfrak{i}(\mathrm{C})$. Java supports abstract classes, which provide only a partial implementation. $\square$

Definition 5. *(Implements ($\geq_I$)).* The relation $C \geq_I I$ is true if C implements the interface I: $\mathfrak{i}(\mathrm{I}) \subseteq \mathfrak{i}(\mathrm{C})$. $\square$

Definition 6. *(Program).* A program is a set of classes. $\square$

A Java class C_1 *depends* on another class C_2 if $\mathfrak{b}(C_1)$ contains references to C_2. The references may include method invocations, field accesses and inheritance. Any change to C_2 may mean that C_1 must change as well [6].

Definition 7. *(Dependency ($\propto$)).* The relation $C_1 \propto C_2$ is true if C_1 depends on C_2. Transitivity applies, denoted by $\overset{*}{\propto}$. The dependency relationship applies to specific methods or fields as well. For instance, $C_1.M \propto C_2.N$ is true if $C_1.M$ invokes $C_2.N$. $\square$

Definition 8. *(Composition ($\oplus$, $\ominus$)).* $C_1 \oplus C_2$ denotes the union of C_1 and C_2, where C_1 and C_2 are two sets of methods and data. $C_1 \ominus C_2$ denotes their difference; the methods and fields that are defined in C_1, but not in C_2. These operators provide an abstraction for the Java inheritance mechanism. Thus, the Java composition semantics of scoping and overloading are implicit in the definitions of $\oplus$ and $\ominus$. $\square$

We do not define $\oplus$ and $\ominus$ precisely because our focus in this paper is more on examining the effects of dynamic classes.

Definition 9. *(Inheritance($\sqsubseteq$)).* The relation $C \sqsubseteq C_S$ is true if C directly extends C_S. Inheritance affects the composition of a class. $\mathfrak{b}$(C) contains the implementations of all of C's superclasses, and $\mathfrak{i}$(I) contains their interfaces. Stated formally:

$$\forall\, T : C \overset{*}{\sqsubseteq} T : \mathfrak{b}(T) \subseteq \mathfrak{b}(C)$$
$$\forall\, T : C \overset{*}{\sqsubseteq} T : \mathfrak{i}(T) \subseteq \mathfrak{i}(C)$$

Transitivity applies, denoted by $\overset{*}{\sqsubseteq}$. Java does not permit recursive inheritance. Thus, $C_1 \overset{*}{\sqsubseteq} C_2 \implies \neg\,(C_2 \overset{*}{\sqsubseteq} C_1)$. Finally, $C_1 \sqsubseteq C_2 \implies C_1 \propto C_2$. Inheritance can apply to both classes and interfaces. Let $\sqsubseteq_C$ specify class extension, and $\sqsubseteq_I$ specify interface extension. $\sqsubseteq$ can refer to either case. □

Definition 10. *(Defines: class ($\geq_C$)).* The relation $C_{def} \geq_C C$ is true if C_{def} is the class definition bound to the name C; C_{def} defines C. A $\geq_C$ relationship is not necessarily permanent, but it is singular; $C_{def} \geq_C C \implies \forall\, C_i$: $C_i \neq C_{def}$, $\neg\,(C_i \geq_C C)$. This restriction preserves Java name semantics — a name should only be bound to one value. □

Definition 11. *(Instantiation ($\leq$)).* The relation $O \leq T$ is true if the object O is an instance of the class or interface T. $O \leq C \wedge C \overset{*}{\sqsubseteq} D \implies O \leq D$. Likewise, $O \leq C \wedge C \geq_I I \implies O \leq I$. □

Definition 12. *(Defines: object ($\geq_O$)).* The relation $C_{def} \geq_O O$ is true iff $C_{def} \geq C \wedge O \leq C$. As with $\geq_C$, $\geq_O$ is not necessarily permanent, and is singular. □

Note that $\leq$ is the transverse of $\geq_O$.

Type safety issues: Changing a class C can have a serious impact on type safety. The interface and/or implementation may be affected. Methods can be added, deleted, or modified, and data fields may be added or deleted. Furthermore, the type itself can change. Adding or removing superclasses or interfaces effectively changes the set of types that an instance of C can be cast or assigned to, with potential effects on any variables bound to such an object. Type violations caused by dynamic changes in class definitions fall into two categories: *static* type violations and *dynamic* type violations. The design and implementation of Java contain mechanisms to prevent either from occuring in a static program. Our system must also prevent them from occuring in a dynamic program, due to class changes.

Here we define static and dynamic type violations, and describe how both the standard JVM and our model prevent these violations and ensure type safety. In doing so, we use the notion of the *type set* of a class C, which is the set of all classes and interfaces to which an instance of C can be cast. The type set contains C itself, all classes from which it inherits, and all interfaces that C or one of its superclasses implements.

Definition 13. *(Type set ($\tau(C)$)).* Let I_C be the set of all interfaces i such that $C \geq_I i$. Let C_S be C's superclass; $C \sqsubseteq C_S$. Then, $\tau(\mathrm{C}) \equiv \{C\} \cup I_C \cup \tau(C_S)$. From the definition of instantiation, $O \leq C \implies \forall\, T\colon T \in \tau(\mathrm{C})\colon O \leq T$. □

A static type violation is an invalid field or method reference. For example, if a method in class C_1 references the field $C_2.X$, and C_2 does not contain a field called X, the reference to X is invalid. This type of violation can be detected statically by examining the source program. The Java compiler and dynamic linker detect static type violations in source code. This mechanism cannot prevent static type violations caused by dynamic class changes. For instance, a class D may invoke a method, say `C.foo()`, of a class C. Now, any dynamic changes in `C` that removes `foo` will cause invalid references to `foo`.

A dynamic type violation occurs when some event results in a reference being bound to an object of an incompatible type. For example, let O be an instance of C. C does *not* implement the interface I. If O is bound to a variable i of type I, a dynamic type violation results. This type violation cannot be detected statically, since it depends on O. The JVM performs dynamic type checking during operations such as assignment and type casting. If an operation might result in a dynamic type violation, the JVM throws an exception. This type of checking does not always catch dynamic type violations caused by class change, since an assignment might have occurred prior to the class change. For instance, assume that C implements interface I. Let O be an instance of C. Some other object has a reference to O, via i, of type I. If C changes such that it no longer implements I, i's reference to O becomes invalid, since O's type has changed. In this example, i has already been assigned a value. If O's type changes, then any subsequent access to i might cause an error. The only way to prevent such an error would be to type check every object reference instruction, which the JVM currently does not do.

We now show how dynamic type violations can occur when classes are changed dynamically. Assume that class C implements the interface I. Thus, $\tau(\mathrm{C}) \equiv \{C, I, Object\}$. We first assign an object of type C to a variable of type I, a legal action. We then modify C such that it no longer implements I; $\tau(\mathrm{C}) \equiv \{C, Object\}$. The reference `i.foo()` causes an error, because the object bound to `i` is no longer of type `I`.

We can now define type safety formally:

Definition 14. *(Type safety).* A class C is type-safe if it contains neither static type violations nor dynamic type violations that cannot be detected by the JVM's runtime type checking. A program P is type-safe if all of its component classes are type-safe. □

There are two approaches to ensuring type safety during class changes. We could place no constraints on how classes can change, and type check every object reference and method invocation instruction. Or, reduce the necessity for extra runtime type checking by placing constraints on class changes. Various definitions of a valid class change are possible, depending on the approach used.

We have defined a valid class change as one that cannot cause type violations, either static or dynamic.

We chose this approach for two reasons. First, we wished to preserve the type semantics of the Java language. A valid Java program, P, does not contain these type violations. Second, efficiency – type checking all method and object references requires significant CPU time. Our model requires only static checking before a class is modified. No extra runtime type checking is necessary.

Formally, we define the semantics of class change to prevent static and dynamic type violations as follows:

Notation: Let $\underline{C}$ denote the definition bound to class C before a change.
Notation: Let $\overline{C}$ denote the definition bound to class C after a change.
Notation: Let ΔC denote the changes made between $\underline{C}$ and $\overline{C}$; $\Delta C \equiv (\underline{C} \ominus \overline{C}) \oplus (\overline{C} \ominus \underline{C})$.

Definition 15. *(Dynamic class change ($\mapsto$)).* The operation $\underline{C} \mapsto \overline{C}$ describes a change to C's definition, and is valid if and only if the following two conditions hold true:

1. No class defined in P, where P is the enclosing program, depends on fields or methods being removed from C.
   $\forall\, C_D \in P : \neg\, (C_D \stackrel{*}{\propto} (\underline{C} \ominus \overline{C}))$
2. An element of C's type set cannot be removed if other classes depend on it.
   $\forall T : T \in \tau(\underline{C} \ominus \overline{C}) : \neg(\exists\, C_D : C_D \neq C \wedge C_D \in P : C_D \propto T)$.

Under these conditions, $\mathfrak{b}$(C) may be changed in any way. Methods and data may be added to $\mathfrak{i}$(C), and removed if doing so does not cause type violations. C's superclass may be changed, and abstract interfaces added or removed as long as types with dependents are not removed from C's type set. Further, $\underline{C} \mapsto \overline{C}$ has the following effects on C subclasses and instances:

1. The change in C's definition is reflected in all subclasses.
   $\forall\, C_D$: $C_D \sqsubseteq C$, $\underline{C_D} \mapsto \overline{C_D}$.
   By the definition of inheritance, $\Delta C_D \equiv \Delta C$.
2. All instances of C change to match the new defintion.
   $\forall\, O$: $O \leq C$, $\underline{O} \mapsto \overline{O}$, where $\underline{C} \geq_O \underline{O}$ and $\overline{C} \geq_O \overline{O}$. See Section 3.2 for more information about this requirement.

□

Note that $C_D \stackrel{*}{\propto} C$ does *not* mean that C_D must change if C does. If C_D depends on C via method invocation, field access, or aggregation (C_D contains an instance of C), then no change to C_D's definition is implied. We discuss this, as well as other details such as method table updates, further in Section 3.3.

The two conditions for $\mapsto$ preserve type safety. The first condition prevents static type violations, and the second prevents dynamic type violations. No other constraints are needed. After any number of changes, a program is still type-safe. Formally, we state this as a theorem:

Theorem 1. *Given* $\underline{P} \overset{*}{\mapsto} \overline{P}$*, if* $\underline{P}$ *is type-safe, then* $\overline{P}$ *is type-safe.*

We prove Theorem 1 using induction on the number of class changes enacted.

Base step: if no change has been made to P, then P is type-safe. True by the definition of a valid Java program.

Inductive step: If $\underline{P}$ is type-safe, then $\overline{P}$ is type-safe. We prove this using contradiction: we have some $\underline{C} \mapsto \overline{C} \implies \underline{P} \mapsto \overline{P}$, where $\underline{P}$ is type-safe and $\overline{P}$ is not. Therefore, $\exists$ some class $X \in P$: $\underline{C} \mapsto \overline{C} \implies \underline{X} \mapsto \overline{X} \wedge \overline{X}$ is not type-safe. There are two cases:

Case 1: $\overline{X}$ contains a static type violation: $\exists\, Y$: $\underline{C} \mapsto \overline{C} \implies \underline{Y} \mapsto \overline{Y} \wedge X \propto (\underline{Y} \ominus \overline{Y})$. Recall Condition 1, which requires that $\forall X \in P : \neg(\exists\, Y, X \propto (\underline{Y} \ominus \overline{Y}))$. This condition contradicts the above.

Case 2: $\overline{X}$ contains a dynamic type violation: $\exists\, C_D : C_D \neq C : C_D \propto T$. $\tau(\underline{X}) \nsubseteq \tau(\overline{X})$. However, $\underline{X} \mapsto \overline{X} \iff \neg(\exists\, C_D : C_D \neq C : C_D \propto T)$ by Condition 2 of $\mapsto$, and we have a contradiction.

Therefore, if $\underline{P}$ is type-safe, then $\overline{P}$ is type-safe. □

2.2 Support for Dynamic Classes

Dynamic classes can be implemented in several ways: (i) by changing the Java language to support mutable classes, as done in [8], (ii) using library-based support, as done with C++ in [19], or (iii) by modifying the virtual machine. We did not wish to modify the syntax or semantics of the Java language. The library-based solution is inefficient and contains intractable implementation problems. In this section, we describe our design, which uses a modifed virtual machine to provide runtime system support for dynamic classes, and extends the class loader to provide an interface.

Java class loader: The interface by which users manipulate dynamic classes is an extended Java class loader. Thus, we begin our discussion with some pertinent background on the Java class loading mechanism.

```
public abstract class ClassLoader {
  public Class loadClass(String name);
  protected Class findClass(String name);
  protected Class defineClass(String name, byte[] b, int off, int len);
  protected void resolveClass(Class c);
        ...
}
```

Fig. 1. Java VM class loader

The JVM resolves references to a class during runtime using a mechanism called the *class loader* [28]. A class loader is responsible for locating the definition

of a class, which takes the form of a class file, and loading it into the JVM. A class in Java is, thus, defined by both its name *and* the class loader that loaded it. The JVM defines two kinds of class loaders: the system class loader and user-defined class loaders. The system class loader is the default class loader used for locating and loading system classes and user-defined classes. Users can override the behavior of the default class loader by defining their own class loaders. To build a specialized class loader, the user must extend the abstract base class `ClassLoader`. Figure 1 depicts part of the interface of `ClassLoader`, as well as the methods that can be overriden in user-defined subclasses.

The dynamic class loader: The programming interface for dynamic classes is the *dynamic class loader*. This class, `DynamicClassLoader`, extends the JVM class loader. In addition, it supports replacement of a class definition, and update of objects and dependent classes. Any class loaded by an instance of `DynamicClassLoader` is automatically a dynamic class.

We chose this approach for several reasons. Since the class loader loads, stores, and examines class definitions, it is a logical choice for a module that modifies class definitions. The design extends Java's dynamic linking mechanism, instead of replacing it. Thus, it supports existing code, with little or no modification. Users can choose to use dynamic classes when and where they see fit. Most importantly, our design preserves the security mechanisms inherent to the class loader system, which include namespace separation and bytecode verification.

```
public class DynamicClassLoader extends ClassLoader {
  public Class reloadClass(String newc);
  public final int replaceClass(String oldc, Class newc);
    ...
  // several overloaded versions of replaceClass are defined
  // for convenience
}
```

Fig. 2. `DynamicClassLoader` interface

The dynamic class loader loads classes from disk in the same manner as the system class loader. It complies fully with the specified semantics of a Java class loader, as described above. However, the dynamic class loader provides additional methods (`reloadClass` and `replaceClass`) that can reload an active class and replace it with a new version. Using runtime system support, these methods implement the semantics of class change ($\mapsto$) as stated in Definition 15. Method `reloadClass` is similar to `loadClass` in that it reads a designated class file from the disk, creates a class object, and returns it. However, `loadClass` does not load classes that are already defined in the system, whereas `reloadClass` succeeds whether the target class was previously defined or not. Given $\overline{C}$, `replaceClass` defines $\overline{C}$ to be the new definition of C, and initiates instance update. These rely

on several native methods that interface with the VM's internal data structures. We provide relevant implementation details in Section 3. Figure 2 summarizes the interface to `DynamicClassLoader`.

Users can extend the dynamic class loader by redefining `reloadClass` or `findClass`. Method `replaceClass` is a `final` method and cannot be overridden. This ensures consistent class redefinition and security, as `replaceClass` performs verification of $\overline{C}$, and enforces namespace constraints.

2.3 Security

In Java 1.2, the JVM prevents classes from performing forbidden actions by using bytecode verification, supporting namespace partitioning, and enforcing user-defined access control policies. The bytecode verifier examines each class before loading it into the JVM, checking for type violations and other illegal operations. Figure 3 depicts a typical namespace configuration in a system that hosts mobile, untrusted applets, such as a web browser. Applets are each run in their own namespace, defined by separate class loaders. Resource classes provided by the host are placed in another namespace Access between namespaces is only permitted `down` the tree; applets are effectively isolated from one another. Furthermore, the user can specify access control policies for more fine-grained protection. In Sun's JDK 1.2 security model, the class `AccessController` acts as a security monitor [2, 14, 15]. All protected resources must call `AccessController.check()`, which checks the access against the permissions specified in the security policy. Although the security policy, and thus permissions, can change, the set of protected resources is static.

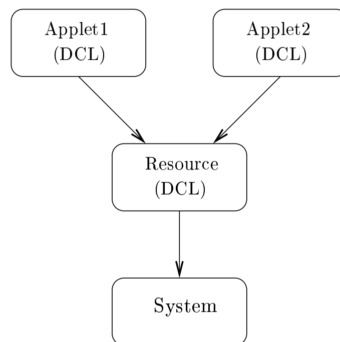

Fig. 3. Typical namespace configuration. `DCL` indicates a dynamic class loader.

Dynamic classes pose new security hazards. Malicious code could potentially bypass many existing security mechanisms, by modifying either itself or the protected classes it targets. Specifically, a malicious class could modify itself in order to perform forbidden actions, or modify sensitive classes to either perform

or allow forbidden actions. Consider, for instance, Figure 4. A host provides a resource, `myResource`, to which access is restricted via an access control policy. A malicious applet, `evilApplet`, contains code that replaces the protected resource with a new version that does not invoke the access controller. `evilApplet` can then gain access to which it is not entitled.

```
public class myResource { // original version
  public static void foo() { // protected resource method
    // perform security check
    accessController.check(new myResourcePermission());
    // access sensitive resource
    ...
  }
}
public class myResource { // weakened version
  public static void foo() { // method now unprotected
    // skip security check...
    // access sensitive resource
    ...
  }
}
public class com.evilDomain.evilApplet extends Applet {
  // malicious mobile applet
  public void start() {
    // get handle to dynamic class loader
    DynamicClassLoader dcl = getClass().getClassLoader();
    // get a URL class loader, linked to originating, evil host
    URLClassLoader ucl = new URLClassLoader(``evil.domain.com'');
    // use it to load a weakened version of the resource class
    Class wr = ucl.loadClass(``myResource'');
    // now use dcl to replace protected resource with weakened version
    dcl.replaceClass(``myResource'', wr);
    // invoke resource, which should be denied but won't be
    myResource.foo();
  }
}
System security policy: only allow local namespace access to resource
grant codebase ``localhost'' {
  permission myResourcePermission;
}
```

Fig. 4. Using dynamic classes to bypass access control.

We want to ensure that dynamic classes do not introduce any new security risks. Therefore, Java's security mechanisms must extend to dynamic classes. This requires several measures.

```
grant codebase ''localhost'' {
    permission ucd.pdclab.dynclass.modifyClassPermission;
}
```

Fig. 5. Grant class modification privileges *only* to classes in the local codebase.

The dynamic class loader subjects all modified classes to bytecode verification before loading them into the JVM, so a malicious class cannot instrument itself to include illegal bytecode operations. The dynamic class loader honors the separation between namespaces by replacing only those classes defined within its own namespace. Returning to Figure 3, let `DCL` denote a dynamic class loader. Thus, applets and resources are dynamic classes. An applet running in namespace `Applet1` cannot use its own class loader to replace a class defined in `Applet2`. The scenario depicted in Figure 4 cannot happen.

These steps do not, however, prevent a malicious applet in `Applet1` from invoking the *resource* namespace dynamic class loader and modifying resource classes. Thus, dynamic class loaders should be protected by an access control policy. Figure 5 contains a simple example of such a policy: only classes from the local codebase, or namespace, can invoke the dynamic class loader. Applets are excluded. `DynamicClassLoader` contains appropriate calls to the access controller, as described in earlier. Under this policy, applets in Figure 3 cannot modify system or resource classes, nor can they modify themselves. A similar policy could provide full protection for `myResource` in Figure 4.

In practice, it is possible to violate Java's type model and compromise security [37]. This is due to problems in the semantics of dynamic linking and the implementation of the virtual machine. The issue does not bear directly upon dynamic classes, and we do not address it.

Another compelling question is that of the security behavior of the program itself. Ideally, we could like to ensure that the security behavior of $\overline{C}$ is no weaker than that of $\underline{C}$; that is, no potential security holes are introduced into the code. However, this problem is unsolvable in the most general case. Conceivably, heuristics could be used, together with assumptions about or constraints on program behavior to solve the problem for specific cases. Such heuristics are, however, beyond the scope of this paper. It remains the responsibility of the programmer to maintain security behavior across changes.

3 Implementation

`DynamicClassLoader` requires virtual machine support for reloading a class definition, finding and updating dependent classes, and finding and updating instances of modified classes. We have modified the Solaris version of Sun's JVM (JDK 1.2). Much of our discussion here pertains specifically to that VM. Our implementation includes a shared library containing functions that support class replacement and instance update. We have also made minor changes in some data

structures and functions internal to the JVM to support the library functions. In the remainder of this paper, we refer to this modified, dynamic classes-enabled virtual machine as DVM.

Adding support for dynamic classes requires understanding and manipulating the JVM's internal data structures and functions in several areas. The JVM uses several optimizations to increase performance, and we take this into account in our design. In this section, we first provide the necessary background on the JVM. We then discuss our implementation.

3.1 Background: Java Virtual Machine

Here we describe the general architecture of the JVM. We focus only on those aspects of the architecture that are relevant to to support for dynamic classes. Specifically, we describe JVM's runtime memory organization, the structure and function of class definition objects, and the optimizations within the bytecode interpreter.

Execution of Java programs: Java programs are composed of classes, each of which is stored in a separate class file. A class file contains the types and definitions of fields and methods defined in the class. All references to classes, fields, or methods are symbolic and contain enough information to allow the JVM to link classes in a type safe manner.

Java heap organization: All Java objects are allocated within a data structure known as the Java heap. In many JVM implementations, including JDK 1.2, the heap is divided into a handle pool and an object pool. Java objects are always addressed indirectly through their handles. The use of handles facilitates garbage collection. When an object is moved, only the pointer in its corresponding handle needs to be updated; the handles never move.

This model is very useful when handling object update for dynamic classes, as described in Section 3.2. The DVM can allocate new space for an object when updating it, without changing the handle used to reference the object.

Class objects: A class object, an instance of `Class`, is created for each loaded class. This object contains the entire class definition, including field types, method signatures and bytecode, and inheritance information. Class objects are special in that they are allocated on the Java heap, but some fields contain pointers into the interpreter's C++ heap. Thus, the code segment for an executing program is distributed among several Java class objects. All names – or classes, methods, fields, etc. – used by the class are stored in the constant pool. In the bytecode, indices into this constant pool are used as symbolic references. Our implementation uses the semantic information contained in class objects to assess dependency relationships among classes and methods.

JVM optimizations: The JVM performs several optimizations that can obfuscate internal data structures and cause problems during class changes. These optimizations include the use of method tables, inlining, quick instructions, and direct referencing. Below, we describe the problems that the optimizations raise during dynamic class implementation and how we resolve them.

Each class data structure contains method and field tables used by virtual method calls and other instructions. These tables contain the names of all methods or fields defined within a class C and its superclasses; each entry has a pointer to the method body or field visible in C's scope. When changing C, the DVM rebuilds the method tables in C and all of its subclasses.

When a class is first loaded, its constant pool contains symbolic references, and its bytecode contains indices into the constant pool for all method and data access instructions. The first time the JVM encounters any such instruction, it checks if the constant pool entry has been resolved, and resolves the entry if needed. Then, the JVM changes the instruction to a special *quick* instruction that does not perform the check. Any subsequent execution of that instruction is relatively fast. Certain quick instructions contain offsets into objects or method tables that may change when a class is modified. To make class updating cleaner, the DVM only uses quick instructions that do not contain any offsets or direct references. This avoids the need to update bytecode, but incurs a slight performance penalty.

JDK 1.2 includes a Just-in-Time (JIT) compiler [22]. JIT compilers provide a significant speed boost to a Java VM by generating native machine code from Java bytecode on the fly. This optimization has an impact on dynamic classes – if a method is modified, previously generated machine code becomes invalid. Therefore, if the JIT compiler is enabled, the DVM must ensure that any modified methods are recompiled. We have not yet implemented this step. At present, the JIT compiler is disabled within the DVM.

The JVM also performs inlining, where some method invocation instructions are replaced by the actual bytecode of the method called. This technique also affects dynamic classes, as inlined code may be invalidated by a class change. We have, therefore, disabled method inlining for all classes loaded by a dynamic class loader. System classes and non-dynamic classes are inlined as usual. We plan to re-enable inlining for dynamic classes by forcing a recompile of any methods that contain inlined code for methods that have changed.

3.2 Updating Instances

There are several alternatives for handling existing instances when a class changes: none, some, or all of them can change to match the new definition. We discuss the options, and justify our decision to enforce global update. Then, we address the implementation details involved in finding, locking, and updating the objects.

Instance update models: Possible models for instance update include a version barrier, passive partitioning, global update, and active partitioning. We describe and compare these models here. Our definitions of version barrier, passive partitioning, and global update, as well as Figure 6, are based on [19].

First, the DVM could use a barrier on object versions. With this solution, $\underline{C} \mapsto \overline{C}$ cannot occur until all objects defined by $\underline{C}$ have expired, as shown in Figure 6(a). Note that, in this case, t_{update} is delayed until all old objects have expired. This solution lacks the flexibility we desired. Effectively, active classes cannot change.

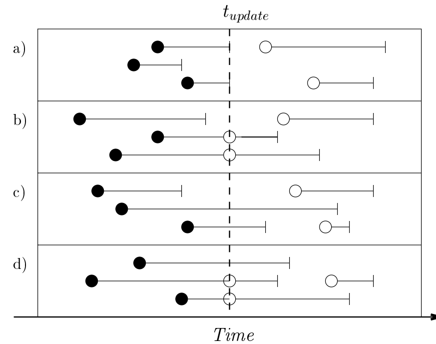

Fig. 6. Object update models: (a) version barrier, (b) global update, (c) passive partitioning, and (d) active partitioning. Black objects denote old version, and white are new version.

Another possibility is passive partitioning, where objects created before $\underline{C} \mapsto \overline{C}$ are unchanged, and any created afterwards reflect the new type. Figure 6(c) depicts this model. In this case, as with the previous, multiple definitions of a class can be active simultaneously. This breaks the Java name-binding semantics, and introduces ambiguity that we wished to avoid.

Active partitioning allows the user to actively select which objects to update and which to leave at the previous version, thus partitioning the objects into type spaces. Such a model effectively implements fully dynamic typing, as one could redefine classes at the granularity of individual objects, giving each object its own dynamic type descriptor. Figure 6(d) illustrates this model. As discussed in Section 2, we chose not to make such a drastic change in Java's type system.

Therefore, our model uses the fourth method: global update of all objects defined by C, shown in Figure 6(b). We defined $\mapsto$ (see Definition 15) such that the DVM must locate and update all instances of C and its subclasses to reflect the new definition, $\overline{C}$.

Implementing incremental global update: Once the DVM has determined that a modified class's instances must be updated, the problem remains of actually locating and processing them. This problem is similar to that of garbage collection. In both cases, there are three major steps: find relevant objects on the heap, lock them, and process them.

Garbage collection algorithms generally fall into one of three categories: basic, incremental, and generational [43]. Basic algorithms use techniques such as reference counting or mark and sweep to identify and process objects in a single transaction. This transaction is atomic in that all other threads must block while garbage is collected, and has the undesirable effect of temporarily halting program execution. Incremental algorithms interleave garbage collection with program execution, alleviating the pause effect. Generational algorithms exploit temporal locality in memory usage to optimize garbage collection. We required a more efficient method than a basic algorithm, and generational algorithms rely

on assumptions about program behavior that may not apply to instance update. Therefore, we chose an incremental mark-and-sweep approach to updating. In this method, there are two phases: the *mark* phase, during which objects are identified, and the *sweep* phase, in which they are actually updated. The mark phase is atomic, and the sweep phase proceeds incrementally.

In the mark phase, the DVM finds the objects by scanning the handle pool, looking for instances of C. When it finds one, the DVM sets a bit in the object header to indicate that the object needs to be updated. Finally, the DVM modifies the class object pointer present in the corresponding handle structure to point to the new definition, $\overline{C}$. The mark phase eliminates the need to check all object references, thus increasing overall efficiency – the DVM only traps references when updates are pending.

In the sweep phase, the DVM incrementally updates marked objects. To maintain the heap in a consistent state, the DVM traps all accesses of marked objects. If any objects need to be updated, the DVM checks the update bit of the target object when interpreting an object reference instruction. It may seem that this sweep technique implements version partitioning in that old and new versions may actually be present on the heap at the same time. However, the implementation guarantees that any old object will be updated *before* it is referenced. The state of an inactive object does not matter. An advantage of this technique is that the DVM does not update objects destined for garbage collection. The disadvantage is a slight performance cost.

The DVM takes several steps to update an object O. Since other threads may be active, it first locks O to prevent race conditions. Then, processing may continue. The DVM allocates a new object $\overline{O}$, where $\overline{C} \geq_O \overline{O}$, and copies $\underline{O}$'s data to $\overline{O}$. It initializes any *new* fields within $\overline{O}$ to zero (`null`), then switches the handles of $\underline{O}$ and $\overline{O}$. Any references to $\underline{O}$ now point to $\overline{O}$, and O is reclaimed by the garbage collector. Finally, the DVM unlocks O, and the method that triggered the update may continue.

3.3 Updating Dependent Classes

When redefining C, the DVM changes the state of all dependent classes to enable C's new definition. This process requires several steps. First, the DVM identifies all dependents and categorizes them by their relation to C – subclass, method usage, etc. It re-resolves all dependent classes, and updates additional information within subclasses.

The DVM identifies dependent classes by scanning the constant pools of all loaded classes for C. Then, it updates each one according to its relation to C. When a class is first loaded, its constant pool contains symbolic references to other class objects and their methods and fields. When the JVM resolves the class, it replaces these references with actual pointers to the referenced object. When the DVM redefines C, any pointers to $\underline{C}$ become invalid, and the DVM replaces all resolved references with the original symbolic references. It then resolves the class and replaces the references with pointers into $\overline{C}$. To support

restoration of symbolic references, we added an additional field to the class object structure that contains the original constant pool.

C's subclasses require additional processing. If any data or methods are added to C, the DVM updates the method and field tables of all subclasses. It rebuilds these tables when it re-resolves a class. Classes that contain instances of C as data require no further action. Since Java objects contain references in their component fields and not entire objects, classes are not affected by a change in the class definition of one of their components.

3.4 Pitfalls in Dynamic Classes

The introduction of arbitrary new code into a running Java application has many potentially negative consequences. Type safety may be affected, race conditions may result, etc. In this section, we analyze several problems that we encountered during the implementation and the present the solutions.

Type safety: Recall from Section 2 that the operation $\underline{C} \mapsto \overline{C}$ is allowed if it does not cause static or dynamic type violations. Therefore, before making any change, the DVM verifies these conditions. This maintains program correctness and type-dependent security mechanisms.

The DVM checks for static type violations by examining $\underline{C}$, $\overline{C}$, and any dependents. Assume that $\underline{C}$ has a field X, and the switch to $\overline{C}$ removes X. There are two possible cases for invalid references – within C, and in other classes. Since we enforce the constraint that $\overline{C}$ must be a valid class definition, the first case is impossible. Before enacting the change, the DVM resolves $\overline{C}$ and runs through the bytecode verifier. Therefore, whether $\overline{C}$ is a compiled class or was generated on the fly, it cannot contain any references to X. However, we may have another class C_D that is dependent on C and references X. In Section 3.3, we described how to identify dependent classes quickly by scanning their constant pools. We extend this technique to locate references to deleted fields or methods such as X – the name $C.X$ must be present in C_D's constant pool. If any such references are found, the DVM invokes a user-supplied handler, passing it a list of classes that depend on $\underline{C}$. This handler may then throw an exception, update dependent classes if possible, etc. This step ensures that all classes defined, and thus the program itself, are valid after the change.

Likewise, the DVM checks the second condition by comparing $\underline{C}$ and $\overline{C}$, and recursively examining C's superclasses. If $\underline{C} \stackrel{*}{\sqsubseteq} C_S$ and $\overline{C}$ does not, the DVM searches for any classes that depend on C_S. If any are found, the DVM throws an exception. Similar steps are taken if $\underline{C} \stackrel{*}{\geq}_I T$ and $\overline{C}$ does not. It is a straightforward matter to extract this information from the class object data structures.

Race conditions in multithreaded applications: Multithreaded applications raise the issue of race conditions on the definitions and instances of dynamic classes. During redefinition, the data in $\underline{C}$ and $\overline{C}$ are in an inconsistent, transitory state. If an active thread references C during this time, runtime system errors

will likely result. The DVM prevents this event by blocking all threads prior to performing the replacement.

Ideally, the DVM could identify all threads that depend on C, and block only those threads. Unfortunately, this requires a lengthy recursive search of all loaded classes for every frame on every thread stack. This operation is actually much more expensive than the class replacement, which is fairly brief. Thus, the DVM blocks all threads except for the thread performing the change, and allows the threads to continue after the change is complete.

Native methods: The JVM allows users to run native methods, and the potential for race conditions during a class change exists here as well. Since native methods do not use a Java stack, and their code cannot be easily examined for dependencies, it is very difficult to determine if it is safe to make a change while a native method is active. Further, native code does not consist of discrete bytecode instruction sections. It is difficult to determine when it is safe to block a native method without causing race conditions as described above.

One solution is to simply disallow class changes while native methods are active. Unfortunately, many native methods are involved in I/O and include polling loops; they are perpetually active. Therefore, the DVM does not block native threads, nor does it wait for them to finish or reach any particular state before continuing with a class change. There is a danger that a native thread could access some internal data structure while the DVM is modifying a class. However, since the JVM cannot control the execution of native methods, there is always the danger that one will corrupt the runtime state in some manner. We assume that all native methods are trusted to behave properly.

Race conditions during object update are easier to handle. Native methods should "pin" Java objects before accessing them, a form of locking. Before changing a class, the DVM scans the heap and ensures that no instances of that class are pinned.

Changing active methods: An interesting problem involves changing a method *that is currently running*. Given a method $C.M$, we must first determine if $C.M$ has changed. Whenever the dynamic class loader loads a class, it calculates and stores a hash value for each method. The DVM can then determine if M has changed by comparing the old and new hash values.

This cannot be done by a simple string compare of $\underline{C}$'s and $\overline{C}$'s versions of M, since the constant pools indices used as arguments in the bytecode may change, even if the method code does not. Any deeper examination of the bytecode becomes costly. So, whenever the dynamic class loader loads a class, it calculates and stores a hash value for each method. This hash value includes all bytecode instructions, and the full names of all classes, methods, and fields referenced, rather than the symbolic references. Then, the DVM can determine if M has changed by comparing the old and new hash values. There is a slight possibility of collision, where two different methods give the same hash value, thus causing a false negative. Therefore, in the event of a match, the DVM also checks other information such as bytecode length and stack size.

Once it has determined that M has changed, the DVM must include $C.M$ in its search of the active thread stacks. Given that M is at an arbitrary point in execution, that Java bytecode contains no semantic information about control flow, and that no particular relationship between $\underline{M}$ and $\overline{M}$ is required, it is impossible, in the general case, to determine where and how to continue execution in $\overline{M}$. For instance, if $\underline{M}$ and $\overline{M}$ solve the same problem using different algorithms, there may not be a point in $\overline{M}$ corresponding to the current location in $\underline{M}$. Or, $\overline{M}$ may use local data that is not present in $\underline{M}$, and that must be initialized. This problem is similar to that posed by security behavior across class changes, as discussed in Section 2.3. Again, heuristics might be used to solve specific cases, but such heuristics are beyond the scope of this paper. Therefore, active methods cannot be changed. If the user attempts to change an active method, the DVM throws an exception, aborting the offending thread. The user may handle this exception in another manner, by continuing the thread but aborting the replacement, terminating and re-invoking the method, etc.

4 Discussion

In this section, we first analyze the performance of the DVM, as compared to the standard JVM. We then compare our design and implementation with other work related to dynamic evolution.

4.1 Performance Analysis

We are concerned with two performance factors: baseline performance of the modified VM, and the cost of replacing a class and updating its instances. We have performed a series of experiments to determine precisely where penalties are incurred and their degree, and to suggest optimizations and improvements. These results pertain to an unoptimized DVM; work on optimization is proceeding apace.

Overhead of adding dynamic classes to JVM: It is straightforward to test the baseline performance of the DVM, simply by running a series of benchmark programs on both the DVM and unmodified JVM. We ran the SpecJVM '98 benchmark suite [41], with a problem size of 100, on a 266 MHz Intel Pentium II running SunOS 5.6. Figure 7 summarizes the results. The performance penalty varied between applications from around five percent to nearly ten, with the average around six percent.

We ran another experiment to determine the penalty caused by each of our modifications. For this experiment, we used a simpler set of benchmark programs [16], run with different versions of the DVM. Each successive DVM version activates an additional instrumentation of the unmodified JVM. Instrumentations include the elimination of quick instructions, checking if an update is needed in object reference instructions (see Section 3.2), and the class replace lock check for object reference and method invocation instructions. Figure 8 summarizes the performance cost distribution. The costly modifications are the

SpecJVM Programs	*JVM*	*DVM*	*JVM/DVM*	*DVM w/ repl.*	*no repl./repl.*
jess	1420.888	1562.581	90.9%	1738.559	90%
db	2675.772	2932.931	91.2%	3257.733	90%
javac	1692.285	1840.9	91.9%	2181.056	84%
mpegaudio	6383.705	6743.099	94.7%	6853.353	98%
mtrt	1709.399	1883.119	90.8%	2163.25	87%
jack	2083.441	2306.306	90.3%	2559.645	90%
Total	15966.552	17269.977	92.5%	18753.596	92%

Fig. 7. SpecJVM benchmark results. All time in seconds.

elimination of quick instructions and the class replace lock; each incurs an approximately 5% penalty. The penalty caused by the object update check is very small. Current efforts focus on reducing these penalties by implementing a more efficient locking mechanism, and possibly re-enabling quick instructions for non-dynamic classes.

VM version	*time*	*JVM/DVM*	*penalty*
JVM	54.6	–	–
DVM	55.8	97.8%	2.2%
No quick instructions	58.8	92.9%	4.9%
Update object check	58.9	92.7%	0.2%
Class replace lock check	62.4	87.5%	5.2%

Fig. 8. Performance cost distribution.

These data inform the wide range in performance cost reported in Figure 7. Applications that have a higher proportion of object reference and method invocation bytecode instructions, as compared to other instruction types, suffer more from both the loss of quick instructions and the class replace lock check.

Cost of modifying classes: The acquisition of meaningful data about the cost of replacing a class and updating instances is more complex. Many variables are involved, including the behavior of the application (object allocation and usage, etc.) and the state of the runtime system (number of classes loaded, thread state, etc.). Thus, different applications will generate widely varying data. We have experimentally modeled this cost by running the Spec benchmarks, as above, alongside a thread that periodically replaced a randomly selected user class. We did not modify any Spec classes; any such class is replaced with itself, causing no instance update. We included a "dummy" class that, when changed, has a different implementation. Our extra thread allocates and periodically accesses many instances of this class. The number of objects used in this set of experiments was 10000, and the interval between class changes was 5 seconds – we consider this to be a fairly heavy replace/update load. We show the results in Figure 7. The overall performance penalty ranged from ten to sixteen percent, with average at eight percent.

4.2 Related Work

We survey related work in dynamic evolution in the context of programming models. We loosely classify techniques according to the semantics of changing code and the programming interface.

Dynamic classes: Under dynamic classes, the definition of a type may be changed at runtime. However, the defining type of an individual object may not, as is the case with dynamic typing. Any change is applied directly to the type definition rather than its instances. Therefore, objects in memory must somehow be partitioned between different versions of the class.

C++-style templates, at first glance, seem to provide some dynamic capability – a template class or function can change based on what template parameter is provided. However, this is static. Effectively, templates generate new classes during compilation, but cannot generate or modify classes at runtime. The Java interface construct suffers from similar limitations, as discussed under dynamic linking. The Java interface construct is not sufficient either; one may load and use a new implementation class for an existing interface, but any existing instances of the original implementation are not affected.

Hjalmtysson and Gray [19] implement dynamic classes in C++. The system uses a wrapper, or proxy class, method that essentially implements Java style interfaces in C++, and further extends the mechanism to allow linking of a new implementation class at runtime, and the presence of multiple active versions. This does not require runtime system support or language extensions, and could be applied to Java as well — we chose not to do so for performance reasons.

Shadows [12] is a system for projecting objects between type spaces, and has been implemented in C++. Shadows also uses a form of proxy class, called a *shadow map*. This map is used to map nodes from the original data structure or type into an extended structure, or *shadow*. Shadows uses runtime type checking to maintain type safety. As with dynamic C++ classes, Shadows does not require compiler or runtime support, but can only be used with specifically coded programs and incurs overhead that might be prohibitive in a Java environment.

Delegation [30] provides a mechanism by which Kniesel [26] implements dynamic classes. Delegation permits *object*- rather than class-based inheritance. A class can contains *delegates*, which are objects invoked to perform certain functions. By changing the delegates bound to a function, one can easily change that function's implementation.

Dynamic linking: Dynamic linking [20, 24, 11] allows names to be bound when the program begins execution. Once done, this binding cannot be changed without restarting the program. The common point among all traditional stages of binding is that any type or method name can only be bound *once* across all phases. Further, dynamic linking contains no notion of state or correctness. Even if it were possible to re-link a dynamic library, there is no semantic framework dictating how and when it may be done.

Load-time transformation: Several projects exist that support modification or generation of classes at load-time (before or during class loading). This tech-

nique can be used to optimize or reconfigure applications by generating and loading specialized classes. However, the method is subject to the limitations of dynamic linking. New classes can be generated and loaded, but classes and objects previously present in the JVM are not affected. Classes can change in the static or loaded state, but not while active. Linguistic reflection [25], Binary Component Adaptation [23] and JOIE [7] implement load-time transformation.

Dynamic architectural frameworks: Architectural frameworks such as COM [5], CORBA [1] and C2 [42] provide a mechanism by which a program can be described in terms of high-level components such as modules and connectors. In general these frameworks are static – once defined, a program is static and its design cannot be changed at runtime. Dynamic frameworks allow the user to change the high-level architectural specification of a program at runtime.

Archstudio [34] provides graphical and command-line tools used to modify a C2-Java program specification at runtime. An attempt to change the specification invokes an Architecture Evolution Manager, which checks the request for validity, and modifies the program's implementation accordingly.

The Argus language [31], which provides a client/server model for distributed computing, supports dynamic update of servers, or guardians [4]. Similarly, Conic [32] provides a module-based environment using message passing. Modules communicate via ports, and may be dynamically updated by switching all links from the present version of a module to a new one. However, the ports between modules are static, thus connections cannot be created or broken dynamically.

Dynamic typing: CLOS [40] and Smalltalk [13] support dynamic typing, in which the type descriptor of an object may be changed freely at runtime. Method code may be modified, data fields and methods may be added or removed, etc. For example, the Information Bus [33] distributed systems architecture uses a CLOS-derived language to implement dynamic classes. Fabry [10] implements a dynamic type system using capabilities. Widening [39] provides a mechanism for *constrained* dynamic type changes, in which objects may be temporarily "widened" to a subtype of their defining class. [8] implement a mechanism similar to widening, for imperative languages, and present a formal type system with proof of soundness.

Dynamic typing, in its unconstrained form, supports the greatest flexibility. However, static type checking of any kind becomes infeasible, so the runtime system must support complete runtime type checking, with all associated overhead.

Parallel versions: One approach to replacing one version of a program ($\underline{P}$) with a new version ($\overline{P}$) is to beginning running *both* versions in parallel, transferring $\underline{P}$'s state to $\overline{P}$ at an appropriate time. Both software and hardware-based solutions exist. Gupta and Jalote [17] use processes as update vectors, and SCP [38] uses redundant CPUs.

While efficient, redundant hardware is obviously expensive, and only practical in certain situations such as the telecommunications environment towards which SCP is targeted. Parallel processes are an efficient technique. However, transfer

of state, which may include open files, displays, and elements not affected directly by the change, can be awkward.

5 Conclusion

We have described the design and implementation of dynamic classes in Java, using runtime support. Our solution is novel in the combination of type safety preservation, nearly unrestricted changes, support for any Java class, and efficiency. These features balance efficiency, convenience, safety, and power of expression.

We have developed a dynamic security infrastructure using dynamic classes [18], as well a mechanism that enhances the dynamism of JDK 1.2's native security model. We are also working on a dynamic architectural framework based on Java Beans [21], and a code distribution mechanism. These applications, in conjunction with our performance analysis, show that dynamic Java classes are a useful language extension that supports an exciting class of software systems. Further optimization of the DVM is an ongoing process.

Currently, our primary focus for future work is the extension of the dynamic classes model to distributed systems. The introduction of distributed applications running across multiple hosts, with objects migrating between them, has many implications. For example, due to latency and packet dropping over the network, our current synchronization model does not scale well to multiple hosts. It is difficult to avoid race conditions while maintaining efficiency. One solution is to simply accept race conditions and work around them. This approach implicitly creates a multiple-version model of classes, which merits further examination.

6 Acknowledgements

We would like to express our appreciation toward Brant Hashii, David Peterson, and Michael Haungs for their support and assistance. We also thank the anonymous reviewers for their excellent comments and suggestions.

This work is supported by the Defense Advanced Research Project Agency (DARPA) and Rome Laboratory, Air Force Materiel Command, USAF, under agreement number F30602-97-1-0221. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright annotation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the Defense Advanced Research Project Agency (DARPA), Rome Laboratory, or the U.S. Government.

References

1. The Common Object Request Broker: Architecture and Specification, Revision 2.0. Object Management Group, July 1996. http://www.omg.org/corba/corbiiop.htm.

2. James P. Anderson. Computer security technology planning study. Technical Report ESD-TR-73-51, Vol. II, Electronic Systems Division, Air Force Systems Command, Hanscom AFB, Bedford, MA 01731, October 1972. [NTIS AD-758 206].
3. K. Arnold and J. Gosling. *The Java Programming Language*. Addison Wesley, 1996.
4. T. Bloom. *Dynamic Module Replacement in a Distributed Programming System*. PhD thesis, MIT, 1983.
5. K. Brockschmidt. *Inside OLE 2*. Microsoft Press, 1994.
6. Eduardo Casais. Managing class evolution in object-oriented systems. In *Object-Oriented Software Composition*. Prentice Hall, 1991.
7. Geoff A. Cohen, Jeffrey S. Chase, and David L. Kaminsky. Automatic program transformation with JOIE. In *Proceedings of the USENIX Annual Technical Symposium*, 1998.
8. Sophia Drossopoulou, Mariangiola Dezani-Ciancaglini, Ferruccio Damiani, and Paola Giannini. Objects dynamically changing class. August 1999.
9. Sophia Drossopoulou, Tanya Valkevych, and Susan Eisenbach. Java type soundness revisited. October 1999.
10. R. S. Fabry. How to design a system in which modules can be changed on the fly. In *2nd International Conference on Software Engineering*, 1976.
11. Michael Franz. Dynamic linking of software components. *IEEE Computer*, 18(9162):74–81, March 1997.
12. Jonathan J. Gibbons and Michael J. Day. Shadows: A type-safe framework for dynamically extensible objects. TR TR-94-31, Sun Microsystems, 2550 Garcia Avenue, Mountain View, CA 94043, 1994. Available from www.sunlabs.com.
13. Adele Goldberg and David Robson. *Smalltalk 80: the Language and its Implementation*. Addison Wesley, Menlo Park, CA, 1983.
14. L. Gong. Java security: Present and near furture. *IEEE Micro*, 17(3):14–19, May-June 1997.
15. L. Gong, M. Mueller, H. Prafullchandra, and R. Schemers. Going beyond the sandbox: An overview of the new security architecture in the Java Development Kit 1.2. In *Proceedings of the USENIX Symposium on Internet Technologies and Systems*, Monterey, California, December 1997.
16. William Griswold and Paul Phillips. Bill and Paul's Excellent UCSD Benchmarks for Java (version 1.1). UCSD Software Evolution Group. http://www-cse.ucsd.edu/users/wgg/JavaProf/javaprof.html.
17. Deepak Gupta and Pankaj Jalote. On-line software version change using state transfer between processes. *Software – Practice and Experience*, 23(9), September 1993.
18. B. Hashii, S. Malabarba, R. Pandey, and M. Bishop. Supporting reconfigurable security policies for mobile Java programs. In *Proceedings of WWW9*, May 2000. To appear. Currently available at http://pdclab.cs.ucdavis.edu.
19. Gisli Hjalmtysson and Robert Gray. Dynamic C++ classes: A lightweight mechanism to update code in a running program. In *Proceedings of the USENIX Annual Technical Conference*, New Orleans, Louisiana, June 1998. USENIX.
20. W. W. Ho and R. A. Olsson. An approach to genuine dynamic linking. *SOFTWARE–Practice and Experience*, 21(4):375–390, April 1991.
21. JavaSoft. Component-based software with JavaBeans and ActiveX. White paper.
22. JavaSoft. *The Java Native Code API*.

23. R. Keller and R. Hölzle. Binary component adaptation. In *ECOOP'98 Proceedings*, Lecture Notes in Computer Science. Springer Verlag, 1998. Also available at http://www.cs.ucsb.edu/oocsb/papers/TRCS97-20.html.
24. James Kempf and Peter B. Kessler. Cross-address space dynamic linking. TR TR-92-2, Sun Microsystems, 2550 Garcia Avenue, Mountain View, CA 94043, 1992. Available from www.sunlabs.com.
25. Graham Kirby, Ron Morrison, and David Stemple. Linguistic reflection in Java. *Software-Practice and Experience*, 28(10), 1998.
26. Gunter Kniesel. Type-safe delegation for run-time component adaptation. In *European Conference on Object-Oriented Programming*. Springer, 1999.
27. Robert Laddaga and James Veitch. Dynamic object technology. *Communications of the ACM*, 40(5):36–38, March 1997.
28. S. Liang and G. Brach. Dynamic class loading in the java virtual machine. In C. Chambers, editor, *Object-Oriented Programming Systems, Languages and Applications Conference,* in *Special Issue of SIGPLAN Notices*, number 10, Vancouver, October 1998. ACM.
29. S. Liang and G. Bracha. Dynamic class loading in the Java Virtual Machine. Draft. JavaSoft, Sun Microsystems, April 1998.
30. Henry Lieberman. Using prototypical objects to implement shared behavior in object oriented systems. In *OOPSLA*, 1986.
31. B. Liskov. Distributed programming in Argus. *Communications of the ACM*, March 1988.
32. J. Magee, J. Kramer, and M. Sloman. Constructing distrubuted systems in Conic. *IEEE Transactions on Software Engineering*, June 1989.
33. Brian Oki, Manfred Pfluegl, Alex Siegel, and Dale Skeen. The Information Bus – an architecture for extensible distributed systems. *ACM Operating Systems Review*, 27(5):58–68, December 1993.
34. Peyman Oreizy, Nenad Medvidovic, and Richard N. Taylor. Architecture-based runtime software evolution. In *Proceedings of the International Conference on Software Engineering*, 1998.
35. R. Pandey and B. Hashii. Providing fine-grained access control for Java programs. In *13th Conference on Object-Oriented Programming. ECOOP'99*, Lecture Notes in Computer Science. Springer-Verlag, June 1999.
36. J. H. Saltzer and M. D. Schroeder. The protection of information in computer systems. *Proceedings of the IEEE*, 63(9):1278–1308, September 1975.
37. Vijay Saraswat. Java is not type-safe. Technical report, AT&T Research, 1997. http://www.research.att.com/ vj/bug.html.
38. Mark Segal and Ophir Frieder. On-the-fly program modification: Systems for dynamic updating. *IEEE Software*, March 1993.
39. Manuel Serrano. Wide classes. In *European Conference on Object-Oriented Programming*. Springer, 1999.
40. Stephen Slade. *Object-Oriented Common Lisp*. Prentice Hall, Upper Saddle River, NJ 07458, 1998. Chapter 13.
41. Standard Performance Evaluation Corporation. *SPECjvm98 Documentation*, 1.01 edition, August 1998. http://www.spec.org/osg/jvm98/.
42. R. Taylor, N. Medvidovic, K. Anderson, E. Whitehead, J. Robbins, K. Nies, P. Oreizy, and D. Dubrow. A component- and message-based architectural style for GUI software. *IEE Transactions on Software Engineering*, June 1996.
43. Paul R. Wilson. Uniprocessor garbage collection techniques. In *Proceedings of the Memory Management International Workshop*. Springer-Verlag, 1992.