Binary Component Adaptation

Ralph Keller and Urs Hölzle
Department of Computer Science
University of California
Santa Barbara, CA 93106
http://www.cs.ucsb.edu/oocsb

Technical Report TRCS97-20
December 3, 1997

Abstract. Binary component adaptation (BCA) allows components to be adapted and evolved in binary form and on-the-fly (during program loading). BCA rewrites component binaries before (or while) they are loaded, requires no source code access and guarantees release-to-release compatibility. That is, an adaptation is guaranteed to be compatible with a new binary release of the component as long as the new release itself is compatible with clients compiled using the earlier release. We describe our implementation of BCA for Java and demonstrate its usefulness by showing how it can solve a number of important integration and evolution problems. Even though our current implementation was designed for easy integration with Sun's JDK 1.1 VM rather than for ultimate speed, measurements show that the load-time overhead introduced by BCA is small, in the range of one or two seconds. With its flexibility, relative simple implementation, and low overhead, binary component adaptation could significantly improve the reusability of Java components.

1. Introduction

Object-oriented programming promises to improve programmer productivity by fostering component reuse. Ideally, new programs are mostly composed of existing components which can be adapted to the specific purpose by using language mechanisms such as polymorphism and inheritance. Thus, relatively little time is spent creating entirely new code, and most of the programming effort lies in combining or specializing existing reusable components [Cox86]. In other words, the main activity associated with software development is not the origination of new programs but the integration, modification, and evolution of existing ones [Win79].

Unfortunately, this vision cannot always be realized in environments that cannot be centrally managed and coordinated. In particular, combining independently developed components can be difficult [Höl93], as can be the evolution of components over time. Even if small (trivial) changes could rectify a problem, source code changes may be impossible for a variety of reasons. Therefore, interoperability between software components is a major issue in software development [Weg96].

Binary component adaptation (BCA) [KH97] allows more flexible object couplings by shifting many small but important decisions (e.g., method names or explicit subtype relationships) from component production time to component integration time, thus enabling programmers to adapt even third-party binary components to their needs. BCA rewrites component binaries before (or while) they are loaded. This rewriting is possible if binaries contain enough symbolic information (as do Java class files, for example). Component adaptation takes place *after* the component has been delivered to the programmer, and the internal structure of a component is directly modified *in place* in order to make changes. Rather than creating new classes such as wrapper classes, the definition of the original class is modified. By directly rewriting binaries, BCA combines the flexibility of source-level changes without incurring its disadvantages. In particular, binary component adaptation

- requires no source code access, so that it can be used on third-party libraries;
- preserves release-to-release compatibility, so that compatibility problems only arise in situations where they would also arise with unmodified components;
- is very flexible, allowing a wide range of modifications (including method addition, renaming, and changes to the inheritance or subtyping hierarchy);
- can be deferred until load-time, so that the adaptations can be distributed and performed “just in time”; and
- introduces only a small load-time overhead.

We have implemented a working prototype of BCA for Java using Sun's JDK 1.1.3 virtual machine (VM). After reviewing some background, we describe our implementation in detail in section 3. Section 4 then presents example uses of BCA and analyzes its overhead.

1.1 Overview

The general structure of a binary component adaptation system is quite simple. Component adaptation takes place *after* the component has been delivered to the programmer, and the internal structure of a component is directly modified *in place* in order to make changes. Rather than creating new classes such as wrapper classes, the definition of the original class is modified.

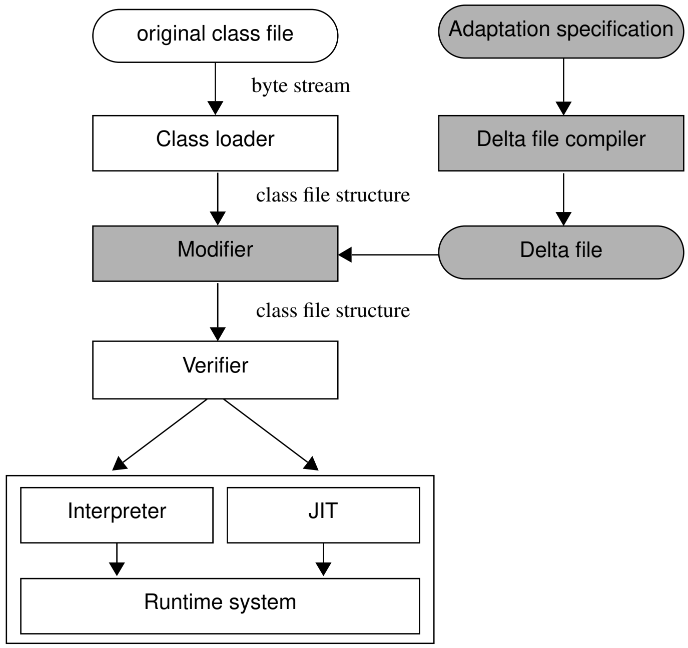

Figure 1. Overview of a binary component adaptation system

Figure 1 illustrates the general structure of BCA system integrated into a Java Virtual Machine (JVM) implementation. The class loader reads the original binary representation of a class (the *class file*); this file was previously compiled from source code by a standard Java compiler and appears in the standard format (i.e., the JVM class file format [LY96]). The class file is retrieved from either a file system or network as an unstructured stream of bytes. It contains enough high-level information about the underlying program to allow inspection and modification of its structure. The file includes code (*bytecodes*) and a symbol table, as well as other ancillary information required to support key features such as safe execution (verification of untrusted binaries), dynamic loading, linking, and reflection. Unlike other object file formats, type information is present for the complete set of object types that are included in a class file. These properties include the name and signature of methods, name and type of fields (class or instance variables), and their corresponding access rights. All references to classes, interfaces, methods, and fields are symbolic and are resolved at load time or during execution of the program. Most of this symbolic information is present because it is required for the safe execution of programs. For example, the JVM requires type information on all methods so that it can verify that all potential callers of a method indeed pass arguments of the correct type. Similarly, method names are required to enable dynamic linking.

The loader parses the byte stream of the class file and constructs an internal data structure to represent the class. In a standard implementation of the JVM (Java Virtual Machine), this internal representation would be passed on to the verifier. With BCA, the loader hands the data structure to the modifier which applies any necessary transformations to the class. The modifications are specified in a *delta file* that is read in by the modifier at start-up of the VM. (We call it *delta file* since the file contains a list of differences or *deltas* between the standard class file and the desired application-specific variant.) The user defines the changes in form of a *delta specification*, which is compiled to a binary format (delta file) in order to process it more efficiently at load time.

After modification, the changed class representation is passed on to the verifier which checks that the code can safely be executed, i.e., does not violate any JVM rules. Since adaptation happens prior to verification, the presence of binary component adaptation has no effect on security. After successful verification, the class representation is then handed over to the execution part of the JVM, e.g., an interpreter and/or compiler. BCA does not require any changes to either the verifier or the actual JVM implementation.

In order to ensure consistency, the adaptation system must have access to the complete program, i.e., all class files that could possibly be affected by modifications. If the complete set of classes can be determined statically, *off-line (static) binary component adaptation* could simply read the transitive closure of all classes and transform them all off-line, producing a new set of class files that forms the complete modified application. Off-line adaptation has the disadvantage that it physically duplicates class files and thus increases disk space usage since each application potentially needs its own copy of every class in every library. In return, off-line adaptation completely eliminates any runtime overhead since no modifications are needed at load time or during program execution. Furthermore, off-line adaptation also allows delivering a complete application to a customer without requiring any changes in the customer's setup (i.e., using the customer's existing, standard virtual machine).

Alternatively, *online (dynamic) binary component adaptation* performs all adaptations dynamically, i.e., as classes are loaded during execution, as shown above. Online adaptation may impose a certain runtime overhead during the loading phase, but code will execute at full speed afterwards. Also, online BCA requires delta files to be distributed with the application, and users must use a BCA-aware runtime system. However, this disadvantage can be compensated by a number of advantages. First, online adaptation requires no additional disk space beyond the space required for the delta file. Second, it does not require a priori knowledge of the set of classes loaded and thus can handle the full Java language including constructs for explicit dynamic class loading. For example, Java's Class.forName method allows a class be looked up by name, causing the class to be loaded dynamically if it is not already present. Since the argument to the call can be an arbitrary string expression, it is not possible in general to determine the transitive closure of classes for a program containing such calls, so that static component adaptation would not be sufficient.

This paper contains the following contributions:

- It presents binary component adaptation, a new mechanism for adapting and integrating software components.
- It describes a working prototype of binary component adaptation for Java integrated into Sun's JDK 1.1.3 virtual machine and shows that an efficient implementation is possible.
- It illustrates how the modifications work, and especially how byte code is added to a class in an efficient way.
- It provides a performance evaluation of our implementation and analyzes the reasons of the overhead.

In the remainder of this paper, we discuss our implementation of online BCA for the JDK 1.1.3 Java implementation. Before going into implementation details, section 2 reviews the motivation for BCA, i.e., what problems BCA is intended to solve. Section 3 then describes our particular implementation, including some of the trickier aspects an online BCA system must solve. Section 4 then evaluates the performance of our prototype implementation, i.e., the load-time overhead imposed by performing adaptations on-the-fly.

2. The Problem

Component reusers face two main problems, integration and evolution. In this section, we review these problems and show how binary component adaptation can solve them. The discussion is kept brief for space reasons and summarizes the more detailed description found in [KH97].

2.1 The Integration Problem

Assume an application using components A and B, each obtained from a different vendor or organization. The simplified class interfaces of these types are as follows (in reality both classes would define additional methods, of course):

```
class A {
    public void output(PrintStream os);
}

class B {
    public void print();
}
```

Note that both classes define support for printing, although the details (method names and signatures) differ slightly. Unfortunately, these minor differences suffice to make the two classes hard to integrate into the same program. For example, suppose a programmer wanted to store components derived from A and B in a list in the application, and later iterate over the list to print all objects:

```
Enumeration e;
while (e.hasMoreElements()) {
    Printable p = (Printable)e.nextElement();
    p.print();
}
```

However, this usage is impossible because of two superficial problems. First, in class A the print method is called output and expects a stream as a parameter. If the programmer had access to source code, it would be possible to add a print method to A; its implementation would simply call output(System.out). But source code access is unlikely since the component was bought from an independent vendor.

Even after solving the first problem, the two components still cannot be combined because of a second problem: A and B have no common superclass or supertype. This problem could be resolved by adding an implements Printable clause to both classes, indicating that they both support the interface Printable. Again, this change requires source code access and a recompilation.

Alternatively, the component user could try to solve integration problems using wrapper objects or other programming techniques. However, such attempts can introduce significant additional programming effort, runtime overheads, and difficult subtyping problems [Höl93] and thus we do not consider them a satisfactory general solution.

One could argue that integration would be seamless if components were "well-designed" right from the beginning. However, we believe that it is highly implausible that such perfection would be common in a open, large component market.[†] Components cannot be viewed in isolation: even if all components are internally consistent and well-designed, their combination may not be consistent. Programs are likely to combine may different components or component frameworks, making perfect harmony unlikely.

Sometimes, "imperfection" is present by design. For example, while it might be very desirable for some applications that the standard class String contain encrypt and decrypt methods, such functionality may have been considered too specialized by the designers of class String and thus was deliberately not included in order to keep the class interface simple. Other component programmers may well have different or even conflicting requirements and therefore would choose a different interface. Even if a component producer could anticipate all possible uses a priori (and that is a big "if"), the resulting interface would be too complicated, and most programmers would be overwhelmed by a multitude of methods, most of which they never need.

In summary, minor incompatibilities prevent pervasive reuse because a few details do not match: a method is misnamed, parameters may appear in a different order, a method requires a slightly different argument type, a component does not fit into a type hierarchy, etc. Therefore, components must be adapted in some manner before they can be successfully integrated into an application.

Binary component adaptation allows classes to be modified on a per-application basis; a *delta file* describes the differences between the standard classes and the application-specific variant. To integrate components A and B from, both classes require a common supertype Printable; in addition, class A needs a print method. These changes result in the following delta specification:

```
delta class A {
    add interface Printable;
    add method
        public void print() { output(System.out); }
}
delta class B {
    add interface Printable;
}
```

Both changes retain binary compatibility: adding a new method cannot affect existing components since they do not refer to this method, and adding an interface is similarly transparent to code that does not use this interface.

2.2 The Evolution Problem

Evolution represents the continuous cycle of activities involved in the development, use, and maintenance of software systems. Software systems evolve over time in response to numerous requirements, including bug fixes, user demands for greater func-

[†] A detailed argument can be found in [Höl93].

tionality, and especially to support changes in related software [Som92]. Because such change is inevitable, mechanisms must be developed for evaluating, controlling and making modifications.

In widely distributed and relatively loosely coordinated environments like the Internet, it is impractical or impossible to automatically recompile pre-existing binaries that depend on a component that is evolving. Thus, component evolution should preserve binary compatibility with pre-existing applications. A change is *binary compatible* with pre-existing client binaries if these binaries that previously executed without linkage errors continue to execute without linkage errors. Linkage errors are type errors detected at link time (e.g., missing method or mismatched method signatures) or at runtime (e.g., invoking an abstract method). Of course, the actual intention of binary compatibility is to ensure that existing binaries continue to run *correctly* with the new component version, but such strong semantic guarantees cannot be verified by a compiler.

Therefore, the evolution of a class library distributed over the Internet is only practical if changes do not abandon support for the already compiled applications. In other words, a library vendor must maintain release-to-release binary compatibility [F+95].

2.2.1 Interface Evolution

The requirement for binary compatibility places stringent restrictions even on the use of simple language constructs such as Java's interfaces. For example, assume an interface Enumeration as shown below which provides a uniform way to iterate through various collections:

```
interface Enumeration {
    boolean hasMoreElements();
    Object nextElement();
}
```

The method hasMoreElements checks for the end of the iteration and the method nextElement returns the next element of the collection.

After this interface was released, suppose we would like to extend it with a method that returns the last element:

```
Object lastElement();
```

However, adding a new member to an interface requires all classes conforming to the interface to provide an implementation for the new method. That is, existing classes that implement Enumeration must implement lastElement because a client may depend on the modified interface. Therefore, this change to the interface does not preserve binary compatibility.

Note that the new method can be expressed by the functionality already provided by the public interface, with an implementation like this:

```
Object lastElement() {
    Object o = null;
    while (hasMoreElements())
        o = nextElement();
    return o;
}
```

While it is easy to provide a default implementation of lastElement for all classes that need it, this does not help. The basic problem is that interface changes must be reflected *immediately* by all classes that implement a specific interface. But these classes are generally not known to the interface developer since they were developed independently by others, and thus they cannot be changed or recompiled.

Interfaces are widely used and generally considered good programming practice, and thus the interface evolution problem is severe. Since any change abandons the support for already compiled code, interfaces are essentially unevolvable. In other words, the interface designer gets "only one chance to do it right".

This problem is easy to solve with binary component modification as well. To add the lastElement method to the Enumeration interface, we just need to provide a default implementation in the delta file, to be added to any class supporting the old interface:

```
delta interface Enumeration
    add method
        public Object lastElement() {
            // code omitted, see above
        }
}
```

This adaptation specification extends interface Enumeration with the additional method declaration and implicitly performs an add method adaptation to every class that supports Enumeration (or any subinterface of Enumeration).

2.2.2 Evolution of Class/Interface Hierarchies

The requirement for binary compatibility also makes it difficult to change a class or interface hierarchy. For example, during the transition from Sun's JDK 1.1.beta3 to the final JDK 1.1, the class java.security.Key was transformed into an interface. This change did not preserve compatibility with pre-existing binaries that depend on Key even though the functionality defined by the interface was exactly the same as in the class (i.e., no methods were added or changed). Nevertheless, the change made it illegal to inherit from Key since it was no longer a class.[†]

In this particular case, the change was possible since it was a change to a beta release that was understood to be unfinished, i.e., programmers did not expect complete binary compatibility with subsequent versions. Had class Key already been distributed widely, the change would have been impossible.

2.2.3 Changed Specifications

Changing a semantic specification poses a major problem for component evolution. Consider a minor specification faux-pas in Java's Date class which provides a system-independent abstraction of dates and times. While days are numbered from 1 to 31, months are numbered from 0 to 11; i.e., 0 is January, and so on. In retrospect, this numbering scheme was probably a poor design decision. Unfortunately, correcting this flaw would introduce major incompatibility problems, since applications that were constructed for the old Date class would no longer work correctly. To prevent these problems, the specification will probably never be updated, and programmers will have to deal with the unintuitive definition indefinitely, a solution which is not entirely satisfactory.[‡]

This problem can also be solved with binary component adaptation, although the solution is somewhat more complicated. The basic idea is to rename the old definition to obsolete_month and then to add the new definition of month:

```
delta class Date {
    rename method month to obsolete_month;
    add method
        public int month() {
            return obsolete_month() + 1;
        }
}
```

A complication arises because the old and new month methods have identical signatures. Therefore, it is unclear whether a particular call site should be redirected to obsolete_month or whether it was compiled with the new definition. Section 3.7.1 describes how our system uses versioning information to solve this problem.

2.3 Discussion

The presence of a flexible and effective adaptation mechanism like binary component adaptation affects both component producers and consumers. Producers benefit from component adaptation because it makes their components more reusable and therefore more valuable. The increased number of situations in which a specific component can be used may also broaden the market for this component, further benefiting the producer. Furthermore, component adaptation may reduce the maintenance and technical support overhead of the producer since fewer customers will request minor changes. Also, the producer benefits from facilitated component evolution as discussed above. Finally, all these advantages are available without delivering source code to the customers, thus helping to protect the producer's intellectual property investment.[††]

[†] In Java, a class can only inherit from another class; classes cannot inherit from interfaces because interfaces do not contain an implementation.

[‡] As a slight improvement, one could add a new function that returns the "correct" month number. But the most intuitive name for this function is already taken, and the presence of multiple functions for the same functionality could further confuse programmers.

Component reusers benefit equally. With binary component adaptation, third-party components are almost as malleable as self-written code. In fact, BCA is arguably better than source code modification, even in situations where source code availability is not a problem. In particular, BCA handles evolution better, guaranteeing compatibility with future versions of the base component. In contrast, source-based versioning tools are much more fragile, since simple changes such as reformatting or rearranging source code may prevent the automatic incorporation of adaptations into a new version. In addition, online BCA handles open systems, including programs that dynamically load unknown classes that must be adapted.

3. Implementing Binary Component Adaptation

In this section we will discuss binary component adaptation and its implementation in detail. We will discuss a Java implementation only, but the concepts should transfer to other languages in a straightforward way.

3.1 Modifications in Java

The range of possible adaptations is limited only by two constraints: the amount of symbolic information available in binaries, and the desire to enforce binary compatibility. In the case of Java, class files contain enough symbolic information to allow virtually any change, so that the first constraint does not apply.

Table 1 lists a wide range of useful changes to a Java class or interface; all of them are supported by our current implementation. Note that by composition more complicated modifications can be constructed. For example, reimplementing an existing method can be accomplished by first renaming the method, then adding a method with the original name. In addition, a number of other modifications could be implemented whose usefulness is currently less clear. For example, visibility attributes (private, public, etc.) can be changed, although some changes may not preserve binary compatibility. One could also add new parameters to an existing method, providing default values to be passed from existing call sites.

	modification	parameter	description
Class	renameClass	<new class name>	rename a class
	renameClassRef	<new class name>	rename a reference to a class (due to renaming of the class)
	changeSuperClass	<new superclass name>	change the super class
Interface	addInterface	<interface name>	add an interface to the implements clause
	changeInterface	<interface name>	change the implements list
Method	addMethod	<method name, signature, byte codes, ...>	add a method to a class
	renameMethod	<method name, signature, new name>	rename a method structure
	renameMethodRef	<method name, signature, new name>	rename a symbolic reference to a method
Field	addField	<field name, initial value>	add a field to a class
	renameField	<field name, new name>	rename a field structure
	renameFieldRef	<field name, new name>	rename a symbolic reference to a field

Table 1. Modifications on a class structure

3.2 Changing the class file

An online binary component adaptation system rewrites components while they are loaded. The modifier operates on the internal representation of the class that the loader builds. The class loader parses the Java class file and stores the various components into an object hierarchy. Each component of the class file format [LY96], like fields, methods, attributes, and various other entries of the constant pool is represented by a C++ class.

Figure 2 shows the high-level representation of a class and its type information. In particular, the class representation contains the name of the class and superclass, the class access rights, a list of implemented interfaces, a method and field table, attributes and a constant pool. Each method can have various optional attributes. A Code attribute holds byte codes and auxiliary infor-

[††] BCA does not change the level of intellectual property protection provided by the base system. Since Java bytecodes can be decompiled relatively easily, their level of protection is low.

mation (e.g. exception handler table), and instructions refer to the constant pool for their operand (e.g., lcd_w #1 refers to the string “hello”). An Exceptions attribute is used to indicate which checked exceptions a method may throw. Fields are listed in a table as well, but unlike methods, they contain no code. Since a class points to the superclass, the complete type hierarchy can be reconstructed, if needed. Knowing the type hierarchy is useful for changes such as moving class members to the superclass, splitting a class, or merging two classes.

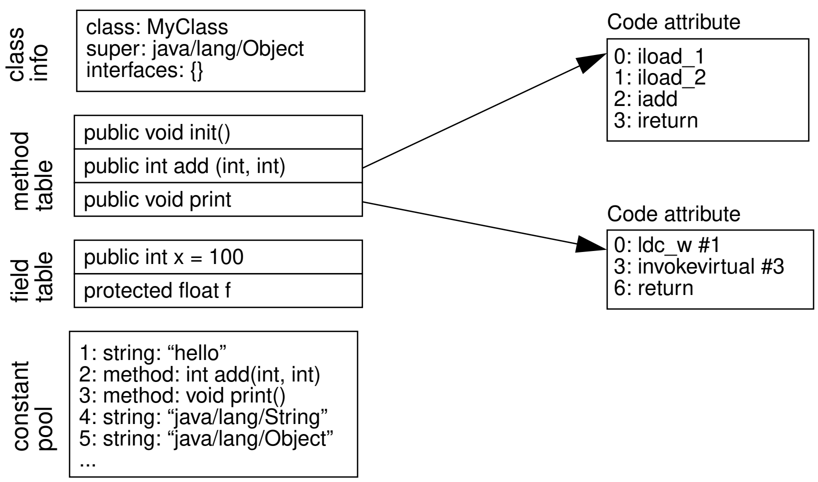

Figure 2. Intermediate class file representation

Modifying the class structure involves adding new parts (methods, fields, interfaces) and changing or deleting existing ones. Adding an instance or class variable (field) to the class is straightforward: the modifier extends the field table by a new field structure. This field structure contains constant pool references to the field name and descriptor, and optionally an initial value attribute. The modifier looks up the field name and descriptor in the symbol table (or adds them if not present) and sets the indices in the field structure.

A class, field, method, or interface is renamed by changing the reference to the constant pool containing that name. To preserve consistency, the modifier must update all references from other classes. Therefore, the constant pool of each loaded class is examined for a reference to the renamed object and rewritten to use the new symbolic name; the same operation is performed on any class loaded in the future.

3.3 Delta files

The delta file contains the changes the modifier must perform at load time. The format should be a compact representation of the adaptation specification so that the modifier can handle it efficiently during class loading. In general, an adaptation specification includes Java source code for added or reimplemented methods. The delta file compiler translates the source code fragments into JVM byte codes and stores them in the delta file.

The delta includes a list of differences, each represented by a pair <*precondition*, *modification*>. The *precondition* defines the property the class must fulfill in order to perform the corresponding *modification.* For example, to add a method to classes implementing a given interface, the precondition requires that the class indeed lists the interface in the implements clause. In our implementation possible preconditions include a *specific class name*, the *class implements a specific interface*, or simply *true* if every class needs modification (useful to update references due to renaming).

The component programmer can generate a delta file by specifying the adaptations using a specialized tool (i.e., graphical editor) or by creating a textual adaptation specification that is subsequently compiled into a delta file. For example, an adaptation specification to add a method to a class could look like this:

```
delta class MyClass {
    add method
        public void print () { System.out.println("hello"); }
}
```

3.4 Delta file compilation

Figure 3 depicts the process of compiling an adaptation specification into a delta file. The delta file compiler generates a constant pool independent format of the byte codes since the constant pool layout of the target class is unknown. Each instruction that refers to the constant pool is annotated with the explicitly resolved constant. A table stores the byte offset of the missing constant pool index together with the resolved constant.

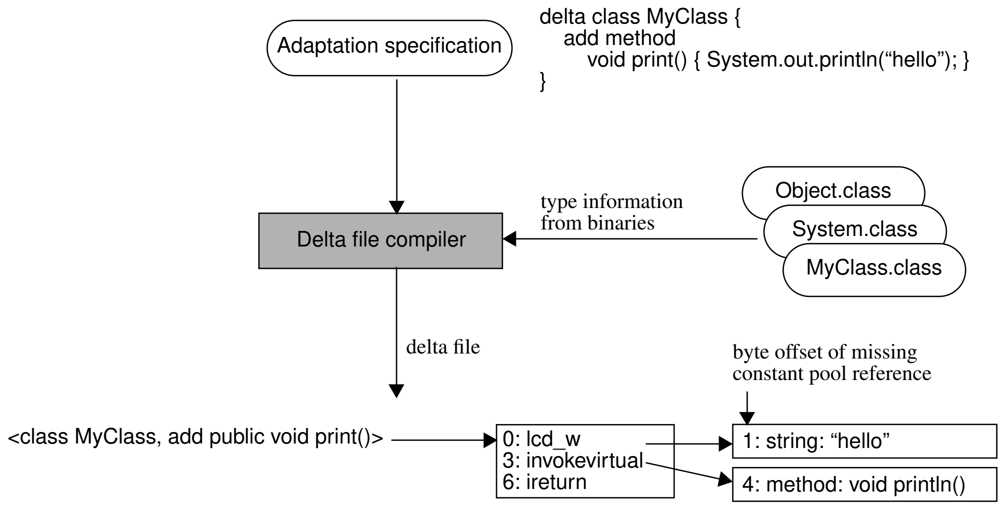

Figure 3. Delta file generation from an adaptation specification

For simplicity, our current implementation uses javac, the standard Java compiler, to generate byte codes for a new method. To add a method to a class, the delta file compiler parses the binary class file, looks up the class and super class name, interface list, access rights, and inspects the variable and method tables. It then generates a textual Java declaration for that class and extends it by the new method. (To add a method to an interface, we create a declaration for a concrete class that conforms to the interface.) The delta compiler then invokes javac to compile the file into a binary class file from which it extracts the new byte codes.

Since the byte codes are specific to the constant pool for the compiled class, the delta compiler must transform them into a constant pool independent format in order to plug them into any destination class file. For that reason, the delta file compiler resolves every constant pool reference and stores the location of the missing constant pool entry together with the constant in a table. During class loading, the modifier updates all missing constant pool indices and adds constants to the constant pool if required.

We must also ensure that the modifier can indeed patch every instruction operand referring to the constant pool during loading. The JVM supports both one- and two-byte constant pool addressing. Updating a one-byte constant pool reference may lead to a conflict if the constant pool already has 255 or more entries. Instruction widening (e.g., ldc is converted into ldc_w) eliminates this restriction but requires an additional byte for addressing of the constant pool. Inserting bytes into the code may invalidate jump targets, and therefore jump addresses may need corrections. Furthermore, instructions requiring 4-byte operand alignment (e.g., tableswitch and lookuptable) may need pad byte adjustments.

In order to prevent these problems, we require that any code added to a class use two-byte constant pool references. In fact, the only instruction using a one-byte constant pool reference is ldc; replacing it with ldc_w guarantees that constant pool references can always be patched during class loading. Because of this restriction, the modifier can avoid instruction widening, jump target translation, and realignment of instruction operands, greatly simplifying the addition of new code to a class.

The delta file compiler transforms the byte codes into a suitable representation in two passes:

1. The compiler determines the new instruction layout by widening constant pool references and aligning instruction operands (e.g., lookuptable and tableswitch) by adjusting pad bytes. The compiler also builds an offset table that stores the beginning of every instruction.

2. In the second pass the compiler generates instructions as determined by the previous pass and adjusts jump targets using the offset table. The compiler resolves all constant pool references and builds a table containing the constant and the location of the reference.

At runtime, the modifier simply walks through the table of constant pool references and updates each two-byte index with the actual index into the constant pool of the modified class.

3.5 Checking the validity of deltas

Component adaptations must preserve a component's internal type correctness as well as existing subtype relationships. The adaptation must result in a program that still satisfies all semantic rules defined by the underlying language (i.e., Java). These constraints can be expressed by a set of preconditions for each modification. For example, the implements clause can only be extended by a new interface if the class indeed implements all required methods. The delta compiler must therefore type-check an adaptation specification before it generates a delta file.

To ensure binary compatibility with future releases of the base component, and to avoid having to re-typecheck the adaptation at load-time, additional constraints are needed. For example, any code introduced by the delta must not reference any non-public members of the base component. The exact rules are very similar to Java's own binary compatibility rules [LY96][GJS96], with the exception that some name clashes are legal as described further below in section 3.7. (Java faces a similar problem because classes can be independently recompiled, and the Java Language Specification [GJS96] contains an extensive section defining binary compatible changes.)

3.6 Compiling against adapted classes

New source code that uses an adapted class must compile correctly and link against adapted classes. For example, assume that a client uses a class that is modified by a delta to include a new method. The compiler must know the changes introduced by the delta in order to call the new method. In other words, each class that the compiler loads must reflect the changes from the deltas. For that reason, the compiler passes every class it loads to the adaptation system before it parses it to retrieve type information.

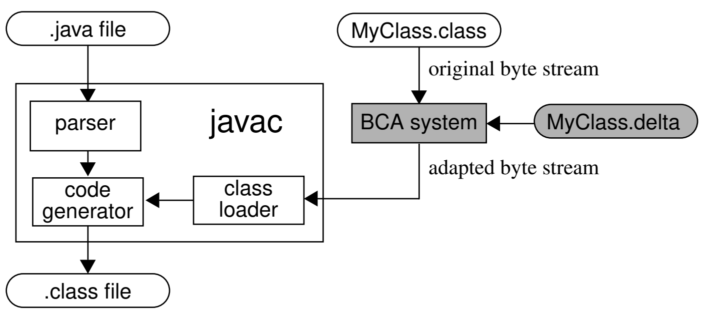

Figure 4. Integration of our BCA system into javac

We integrated BCA into the standard Java compiler javac by modifying its class loader (Figure 4). Whenever the class being compiled references another class, javac loads the corresponding class file. We modified javac to call the BCA system whenever such a class file is loaded. Integrating the adaptation system was simple and required only a few extra lines that invoke the modifier via a native method call. (We wrapped the modifier, which is written in C++, into a Java class containing a single native method.) The adaptation logic is the same as for the VM class loader, and thus both can use the same dynamic library.

3.7 Enabling Binary Compatibility

Component adaptations should preserve binary compatibility, unless the programmer is willing to forgo its advantages. Binary compatibility rules require adaptations to depend only on the public items (classes, variables, methods, etc.) of a component, so that the adaptation can be applied to any compatible future release of the component. Therefore, the compatibility rules ensure that adapted binaries are at least as compatible with future releases as ordinary clients would be.

3.7.1 Disambiguation

With component adaptation, classes are subject to changes via two independent paths, namely *evolution* and *adaptation*. This situation poses a new problem: Changes introduced by evolution may interfere with changes made by adaptation. Figure 5 illustrates this point: if a programmer adds a method print to component A through a delta Printable, then the delta could introduce a conflict when applied to a future release of the component A' that added a print method with the same signature. Since there are two identical method declarations, we have a name clash. Fortunately, this conflict is simple to detect when a class is loaded. Renaming can resolve the conflict as long as the BCA system can identify, for each call site, which of the two method it calls. The BCA system transparently renames one of the methods (including all affected calls) to resolve the conflict without any programmer intervention.

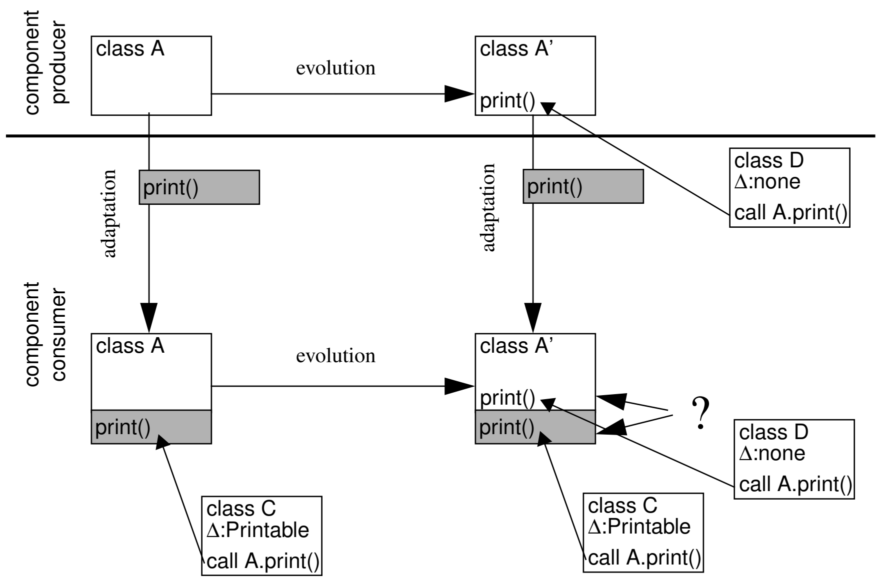

Figure 5. Disambiguation of a name clash

But how can the system tell whether a particular call site was compiled against the new component version (and thus refers to the new method introduced by the component producer) or was compiled against the adaptation created by the component consumer? To solve this problem, we require each client of an adapted binary to record this relationship.

3.7.2 Marking classes

This relationship is expressed by an attribute that is automatically added to every class compiled against an adapted component. Thus, when a name clash occurs, the system can inspect the Deltafile attribute of every class containing an ambiguous call. If the Deltafile attribute is absent, the class was compiled against an unadapted component, and any calls to print from this class must refer to the print method introduced by the producer.

In Figure 5 the compiler marked class C with the Printable delta, since it was compiled against an adapted class A. On the other hand, class D includes no marker since it uses A', the evolved version of class A. Even when the adaptation is applied to the evolved A' (as shown in the lower right part of Figure 5), the adaptation system can still handle the situation and resolves the print call in class C to the code introduced by the adaptation because C includes a marker. Likewise, the print reference in class D must refer to the unadapted code since no marker is present in D.

No class can contain calls to both methods, since such a class would have been compiled against both the adaptation and the new component release. If a programmer needs to write a class that uses both the adaptation and the new (conflicting) features of the new release, he or she must first resolve the conflict by renaming the method in the adaptation, i.e., by creating a new delta.

We changed javac to automatically annotate the output class file with a DeltaFile attribute containing the names of all deltas used to compile the class.

4. Experiments

Binary component adaptation introduces a certain overhead during the class loading phase, but code executes at full speed afterwards. Since class loading is part of the total runtime, the rewriting of classes must be efficient. In this section we show some example uses of BCA and analyze their execution performance.

We designed our implementation for easy integration with Sun's JDK 1.1 VM rather than for ultimate speed. In the current implementation we modify class files before they are passed on to the native JDK loader (Figure 6). This organization is simple to implement since the presence of modifications is invisible to the standard virtual machine (VM), and thus the VM needs only minor modifications in order to insert BCA into the loading process.

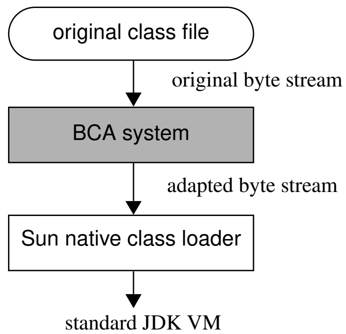

Figure 6. Integration of our BCA system into JDK VM

The main disadvantage of this approach is its higher overhead: the class file is parsed twice, once by the modifier and once by the native class loader. That is, the BCA system first parses the class file and builds its internal representation. Then it modifies this representation and writes a new class file into a memory buffer from which the native class loader loads the modified class. We will quantify the extent of this redundant parsing and unparsing further below.

4.1 Benchmarks

We chose a variety of Java applications written in pure Java, containing no native methods. Since BCA introduces a certain overhead during class loading but not while byte codes are executed, we expect BCA to perform better on applications that run for a longer period of time since the relative load time overhead is smaller. To measure this effect, both short- and long-running programs were tested. Table 2 shows the Java applications used for our benchmarks. All applications are realistic programs that typically consist of more than 100 classes.

application	input data	description	#loaded classes	Kbytes of byte codes
javac	javac source files	Sun Java compiler compiling its own sources	219	754
JLex 1.2	sample.lex	lexical analyzer generator	71	251
Pizza 1.1	Main.pizza	Pizza language extensions compiler	171	748
SunCC 0.7pre5	Corba-IDL specification	Java compiler-compiler	115	682
JCUP	parser.cup	parser generator	107	391
Appletviewer	curtime.class	loading and displaying applet, then quitting	225	675

Table 2. Description of benchmark programs

We also developed delta files intended to be typical BCA uses. BCA performance depends primary on two parameters, namely the kind of modifications made and the number of classes affected. To examine the behavior depending on the number of rewritten classes, we designed deltas that mutate no class at all, one class, a few, and many. To study the overhead introduced by the BCA system, all deltas rewrite the applications in a way that does not produce a semantically different program (e.g. no new methods are called). Mutating the control flow would lead to side effects and could therefore distort the results. Table 3 shows the deltas and their file size.

delta file	description	size (bytes)	affects classes
Empty	contains no modifications at all	6	no class
VectorEnumeration	make Vector conformant to the interface Enumeration by adding methods hasMoreElements and nextElement together with a new instance variable currentElement; include Enumeration in the implements clause	299	class Vector
AddPrintSerializable	add a print method to every class implementing Serializable	218	classes implementing interface Serializable
RenameStringLength	rename method length in class String to newlength and update all references in all existing classes	112	class String; all classes are checked for existing references

Table 3. Delta files

The Empty delta file performs no changes at all. This delta is useful to get an idea of the base overhead introduced by our prototype implementation due to class file conversion. An ideal BCA implementation would be as fast as Sun's VM for this delta. The VectorEnumeration delta complements class Vector with two missing methods and an instance variable so that it conforms to the interface Enumeration. Deltas can also rewrite classes that are not explicitly mentioned by name. For example, AddPrintSerializable changes all classes implementing Serializable by adding a print method. This may affect a large number of classes, depending on how many classes that conform to Serializable are loaded. The delta RenameStringLength changes the method name length in class String and updates all references in existing classes.

4.2 Measurements

Table 4 shows absolute execution times in seconds for Sun's JDK 1.1.3 VM and our modified VM. We ran each benchmark 5 times on an otherwise idle 40MHz SPARCstation-10 and reported the sum of user and system time of the fastest run.

application	Sun's JDK VM	BCA-enabled VM							
		Empty		VectorEnumeration		AddPrintSerializable		RenameStringLength	
	execution time	execution time	%	execution time	%	execution time	%	execution time	%
javac	56.6	56.5	-0.3%	57.3	1.1%	57.2	1.0%	57.9	2.3%
JLex	28.8	30.1	4.7%	31.0	8.1%	29.7	3.4%	30.3	5.4%
Pizza	4.2	6.9	66.7%	6.8	62.8%	6.9	66.2%	7.0	67.6%
SunCC	26.2	27.6	5.3%	28.0	6.8%	27.9	6.3%	28.0	6.6%
CUP	7.7	9.1	18.3%	9.3	20.1%	9.2	19.6%	9.1	18.2%
Appletviewer	8.0	10.4	29.1%	10.7	33.8%	10.7	33.2%	10.7	33.7%

Table 4. Benchmark execution times (in seconds)

Overall, the load-time overhead introduced by BCA is small in absolute terms, even though we designed our implementation for easy integration with Sun's JDK VM rather than for ultimate speed. The base loading overhead (Empty delta) is about 1-2 seconds for the above benchmarks. As explained in section 4.1, an implementation that integrated the modifier into the native loader (as illustrated in Figure 1) would not incur this overhead. The table also reveals some random variations which are probably caused by cache effects or network activity. For example, javac executes slightly faster than Sun's VM when using the Empty delta file. In other words, the base overhead of BCA is so small that it can be obscured by random variations in execution time, especially for long-running programs.

The overhead of the other deltas is not much higher than the base overhead incurred by Empty; the largest difference is 0.9 seconds (VectorEnumeration for JLex). To find out how much of the overhead is actually spent modifying the class data structures (rather than parsing and unparsing the class files), we profiled our BCA implementation with gprof. Table 5 shows the number of changes that the modifier performed on the class files and the percentage of the runtime overhead used to make modifications.

application	VectorEnumeration					addPrintSerializable					renameStringLength				
	#add method	#add field	#rename method	#rename method ref	% of overhead for modifications	#add method	#add field	#rename method	#rename method ref	% of overhead for modification	#add method	#add field	#rename method	#rename method ref	% of overhead for modifications
javac	2	1	-	-	0.2%	12	-	-	-	9.0%	-	-	1	219	12.8%
JLex	2	1	-	-	2.6%	9	-	-	-	15.0%	-	-	1	71	13.0%
Pizza	-	-	-	-	0%	10	-	-	-	9.4%	-	-	1	171	10.7%
SunCC	2	1	-	-	1.1%	11	-	-	-	7.1%	-	-	1	115	12.4%
CUP	2	1	-	-	1.0%	17	-	-	-	19.7%	-	-	1	107	17.2%
Appletviewer	2	1	-	-	0.3%	33	-	-	-	26.7%	-	-	1	225	12.8%

Table 5. Percentage of overhead used for modifications

Clearly, modifications contribute only a minor part to the overhead. In other words, the overhead of our current implementation is dominated by the (unnecessary) reading and writing of the class file. The actual modifications typically take only about 10% of the overall time, indicating that a factor of 10 speedup may be possible with an implementation that avoided the extra reads and writes. In the worst case, the modifications consume about one quarter of the overall overhead in the case of addPrintSerializable which adds a print method to each class implementing Serializable.

4.3 Discussion

Rewriting class files while they are loaded is efficient and imposes only a minor load time cost. Clearly, the performance our implementation could be improved significantly, but we have not yet optimized it because even the unoptimized version appears to be fast enough, and we did not want to lose the ease of integration into existing VMs (e.g., to port BCA to newer versions of the JDK). We measured the overhead on a relatively slow SPARCstation-10 to improve the accuracy of our data through longer program executions. On current machines such as a 200MHz Pentium Pro, the typical overhead of 1-2 seconds measured above drops to a few tenths of a second, i.e., the CPU overhead is negligible. For such machines the dominating cost of BCA will most likely be the extra file I/O to read the deltas files from disk or over a network. Since delta files are highly encoded and small compared to the base class files, this I/O overhead is likely to be very small as well. In fact, BCA can even reduce the I/O overhead, for example, when adding a method to a group of classes. Without BCA, such a change would increase the file size of each class, whereas with BCA the method is read only once. In summary, we believe there is strong evidence that BCA will impose only a very small execution penalty on a BCA-aware system.

4.4 Status and Future Work

We have developed a working prototype of dynamic BCA for Java that implements a wide range of the adaptations. Our system includes a BCA-aware Java virtual machine, a delta file compiler to generate deltas from an adaptation specification, and a modified version of javac in order to compile programs that use adapted components. We plan to make this implementation publicly available in order to facilitate experimentation with binary component adaptation.

Our current prototype does not yet include an easy-to-use tool for generating delta files. That is, the delta compiler is provided as an API and lacks a front end parser or graphical editor. However, the API itself is fully implemented and functional. In the near future, we plan to support additional modifications for refactorings (e.g., split a class, merge two classes, move a method upwards in the type hierarchy, etc.). The delta compiler also does not include a type checker yet but relies on javac to report errors in methods added by deltas.

In the future, we would like to generalize the BCA concept to allow more modifications directly on the byte codes. While our current implementation supports adding new byte codes and adjusting the constant pool, it does not mutate existing code within classes. Modifying the byte codes could be especially interesting for a wide range of new considerations. For example, a profiler might require some method entry and exit code to measure execution times and to build a call graph. This code would automatically be introduced when a class is loaded, making it unnecessary for the user to compile a profile version of every class. Or a graphical debugger could construct its own representation of user data structures or control flow in order to visualize them in a user-friendly way. BCA could also be used to implement generic types (e.g. templates) by instantiating parametric types during loading as proposed by Agesen et al. [AFM97].

5. Related Work

Much previous work has viewed the problems of component adaptation and evolution as a programming language problem, and several language features have been proposed that can address some (but not all) problems. *Descriptive classes* [San86] allowed the programmer to create supertypes after the fact, solving part of the example problem in section 2.1. The usefulness of creating new superclasses for existing classes was discussed in detail by Pedersen [Ped89] and implemented in Cecil [Cha93a]. *Predicate classes* [Cha93b] offer yet another way to extend and adapt objects, although they were conceived for a different purpose. In languages allowing multiple dispatch (e.g., CLOS and Cecil) new methods can be attached to existing classes.

Horn's *enhancive types* [Hor87] do not alter types (classes) directly but instead allow a base type to be coerced into another type (the enhanced type) that offers additional methods implemented in terms of the base type's public interface. Unlike BCA, enhancive types do not allow the addition of instance variables or the renaming of methods or classes. Remarkably, enhancive types respect compatibility, i.e., no existing code needs re-typechecking because of the enhancements, and thus they can be applied to future versions of the code without problems. Since existing code is left untouched, however, the range of permissible modifications is much smaller than with BCA.

Objective-C's *categories* are named collections of method definitions that are added to an existing class [ObjC96]. Rather than defining a subclass to extend an existing class, category methods are added directly to the class. As with subclassing, there is no need for source code for the extended class. Thus, categories allow class adaptations similar to binary component adaptation. However, changes are limited to adding and overriding methods; a category can't declare any new instance or class variables or rename a method. When a category method overrides an existing method, the new version cannot invoke the method it replaces. Finally, the Objective-C compiler cannot guarantee compatibility with future versions of the class.

In previous work [Höl93], Hölzle discussed the shortcomings of language-based solutions to component integration and observed that many problems could be solved if types could be modified in place. He also proposed to deliver executables in a higher-level format to allow such modifications, but since no language supported a standardized high-level executable file format at the time the idea seemed impractical.

Palsberg and Schwartzbach recognized similar problems with traditional reuse mechanisms and proposed a type substitution mechanism aimed at maximizing code reuse [PS90]. However, for adaptation purposes, their solution is both too general and too restricted: too general because in the presence of subtype polymorphism it may require the re-typechecking of a component's implementation, and too restricted because it does not allow adding new methods or renaming methods.

Nimble [PA91] is a tool for procedural languages that allows programmers to transform actual procedure parameters at run-time. A map defines the rules of parameter conversion and is used to generate an adaptor which is linked into the application, performing the parameter translation at run-time. Although Nimble does not require source code and can bridge simple interface mismatches, it is much more restricted than BCA since it only addresses parameter type conversions.

Bracha [Bra92] presents a framework for modularity in programming languages, viewing inheritance as a mechanism for module manipulation. Bracha proposes *mixins*, a more powerful form of inheritance since a mixin is not inextricably bound to a parent. A mixin is a function from classes to classes, parametrized by a parent which it is modifying. In our context, a delta can be regarded as a mixin which is then applied to a class. Unlike mixins, deltas modify the class in place rather than producing a new class. Even though Java does not support mixins at the language level, they could be simulated using binary component adaptation.

An alternative adaptation strategy to BCA, *dynamic interface adaptation* [MS97] dynamically loads adaptors (wrappers) to bridge interface mismatches between Smalltalk components. Adaptors (or wrappers) implement a mapping function between call-out and call-in interfaces of components. If the interfaces between components show mismatches, the run-time system looks up an appropriate adaptor and configures it on demand. Yellin and Strom [YS97] define adaptors that in addition to interface mismatches can also bridge sequencing constraints (protocols). Unlike BCA, adaptors do not modify classes in place and usually slow down calls to adapted components.

BCA could be used to support program refactoring. For example, Opdyke [Opd92] defines a set of restructuring operations (refactorings) to support the design, evolution and reuse of object-oriented application frameworks. Refactorings do not by themselves change the behavior of a program, but they restructure it in a way that makes the software easier to extend and reuse.

Similarly, Hürsch [Hür95] presents a framework for evolution that automatically maintains the overall consistency of an object-oriented system. Both systems could operate on binaries using BCA instead of transforming source code.

Forman et al [F+95] discuss release-to-release binary compatibility between class libraries and its importance for software distribution. IBM's System Object Model (SOM) guarantees that new methods and classes can be added to a SOM library without recompiling client applications. SOM also supports evolution of class libraries through a large number of compatibility preserving transformations, but all the transformations require source code availability. However, it would be conceivable to construct a BCA system for SOM binaries.

Adaptable binaries [G+95] also allow direct transformations on a binary, but unlike BCA the adaptation is performed at the instruction level, i.e., it ignores the programming language semantics. Its main applications are quite different from BCA and include software-fault isolation (software memory protection), machine-retargeting, and optimizations.

A number of higher-level binary distribution formats (e.g., ANDF [OSF91], Omniware [A+96], Slim Binaries [FK96], and BRISC [E+97]) could be extended to support BCA. In addition to a description of all classes and types, the object files need enough information to allow dependent code to be updated (e.g., because dispatch tables or object sizes changed).

Harrison and Ossher [HO93] as well as Smith and Ungar [SU96] argue for subject-oriented programming, i.e., an object model that allows multiple different perspectives of the same object. Ideally, different (subjective) views of the same object can coexist simultaneously in a single application. Binary component adaptation does not go quite that far, allowing only one application-specific version of a class. Subject programming also assumes full source-level access at integration time and does not necessarily preserve binary compatibility, and views subjectivity more as a programming language level mechanism than as a binary rewriting tool.

6. Conclusions

Component producers and consumers spend considerable effort on integrating and evolving components. Binary Component Adaptation (BCA) can reduce this effort by enabling the reuser to more effectively customize components to the needs of the particular application and by supporting predictable and non-predictable component evolution.

BCA differs from most other techniques in that it rewrites component binaries before (or while) they are loaded. Since component adaptation takes place *after* the component has been delivered to the programmer, BCA shifts many small but important decisions (e.g., method names or explicit subtype relationships) from component production time to component integration time, thus enabling programmers to adapt even third-party binary components to their needs. By directly rewriting binaries, BCA combines the flexibility of source-level changes without incurring its disadvantages:

- It allows adaptation of any component without requiring source code access.
- It provides release-to-release binary compatibility, guaranteeing that the modifications can successfully be applied to future releases of the base component.
- It can perform virtually all legal modifications (at least for Java), such as adding or renaming methods or fields, extending interfaces, and changing inheritance or subtyping hierarchies. Several of these changes (e.g., extending an existing interface) would be impossible or impractical without BCA since they would break binary compatibility.
- BCA handles open, distributed systems well because the programmer can specify adaptations for an open set of classes (e.g., all subclasses of a certain class) even though the exact number and identity of the classes in this set is not known until load time.
- Since binary adaptations do not require the re-typechecking of any code at adaptation time, BCA is efficient enough to be performed at load time.

To our knowledge, no other mechanism combines all of these advantages.

Binary component adaptation is suitable for any system that includes enough high-level information in its object files. Languages that support type-safe dynamic linking and structural reflection already include much of the necessary information in binaries. In particular, binary component adaptation fits very well with Java's JVM binary file format which supports virtually all legal modifications without impacting Java's security model.

We described an efficient implementation of BCA for Java using Sun's JDK 1.1 VM. The implementation requires only minimal changes to the virtual machine and the Java compiler. No changes to the JVM class file format are required, and neither does our implementation require any changes to the bytecode verifier or any other part of the JVM definition. Therefore, it can easily be integrated into other JVM implementations and is upwards compatible with existing VMs.

Even though our current implementation was designed for easy integration with an existing VM rather than for ultimate speed, our measurements show that the load-time overhead introduced by BCA is small, in the range of one or two seconds of execution time. A more efficient implementation that is more tightly integrated with the basic VM would substantially reduce this overhead further.

We believe that with its flexibility, relatively simple implementation, and low overhead, binary component adaptation could significantly improve the reusability of Java components by enabling the reuser to more effectively customize components to the needs of the particular application.

7. References

[AFM97] Ole Agesen, Stephen N. Freund, John C. Mitchell. Adding Type Parameterization to the Java Language. In OOPSLA' 97 Conference Proceedings, Volume 32, Issue 10, October 1997.

[A+96] Ali-Reza Adl-Tabatabai, Geoff Langdale, Steven Lucco, Robert Wahbe. Efficient and Language-Independent Mobile Programs. In *PLDI'96 Proceedings*, pp. 127-136, Philadelphia, Pennsylvania, May 1996.

[Bra92] Gilad Bracha. *The Programming Language Jigsaw: Mixins, Modularity and Multiple Inheritance,* Ph.D. dissertation, University of Utah Computer Science Department, March 1992.

[Cha93a] Craig Chambers. *The Cecil Language—Specification and Rationale.* Technical Report 93-03-05, Computer Science Department, University of Washington, Seattle 1993.

[Cha93b] Craig Chambers. Predicate Classes. In *ECOOP '93 Conference Proceedings*, Kaiserslautern, Germany, July 1993.

[Cox86] Brad Cox. *Object-Oriented Programming: An Evolutionary Approach.* Addison-Wesley, Reading, MA 1986.

[E+97] Jens Ernst, William Evans, Christopher W. Fraser, Steven Lucco, Todd A. Proebsting. Code Compression. In *PLDI'97 Proceedings*, pp. 358-365, Las Vegas, Nevada, June 15-18, 1997.

[FK96] M. Franz and T. Kistler. Slim Binaries. *Technical Report No. 96-24*, Department of Information and Computer Science, University of California, Irvine, June 1996.

[F+95] Ira R. Forman, Michael H. Conner, Scott H. Danforth, Larry K. Raper. Release-to-Release Binary Compatibility in SOM. In *Proceedings of OOPSLA '95, ACM SIGPLAN Notices,* Volume 30, Number 10, October 1995.

[GJS96] James Gosling, Bill Joy and Guy Steele. *The Java Language Specification*. Addison-Wesley, 1996.

[G+95] Susan L. Graham, Steven Lucco, Robert Wahbe. Adaptable Binary Programs. In *USENIX*, Winter 1995, pp. 315-325.

[HO93] William Harrison and Harold Ossher. Subject-Oriented Programming. *OOPSLA '93 Conference Proceedings*, Washington DC, October 1993.

[Hür95] Walter L. Hürsch. *Maintaining Consistency and Behavior of Object-Oriented Systems during Evolution*. Ph. D. Thesis, College of Computer Science of Northeastern University, August 1995.

[Hor87] Chris Horn. Conformance, Genericity, Inheritance and Enhancement. In *ECOOP '87 Conference Proceedings,* pp. 223-233, Paris, France, June 1987. Published as Springer Verlag LNCS 276, Berlin, Germany 1987.

[Höl93] Urs Hölzle. Integrating Independently-Developed Components in Object Oriented Languages. In *Proceedings of ECOOP'93,* Springer Verlag LNCS 512, 1993.

[KH97] Ralph Keller and Urs Hölzle. ICSE'98 submission. Also available as *Technical Report TRCS97-15*, Department of Computer Science, University of California, Santa Barbara, September 1997.

[LY96] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification*, Addison-Wesley, September 1996.

[MS97] Kai-Uwe Mätzel and Peter Schnorf. Dynamic Component Adaptation. *Ubilab Technical Report 97.6.1*, Union Bank of Switzerland, Zürich, Switzerland, June 1997.

[ObjC96] Apple Computers. *Object-Oriented Programming and the Objective-C Language*. http://devworld.apple.com/dev/SWTechPubs/Documents/OPENSTEP/ObjectiveC/objctoc.htm.

[Opd92] William F. Opdyke. *Refactoring Object-Oriented Frameworks.* Ph.D. Thesis, University of Illinois at Urbana-Champaign, 1992.

[OSF91] Open Systems Foundation. *OSF Architecture-Neutral Distribution Format Rational.* Open Systems Foundation, June 1991.

[PA91] Purtilo J. and Atlee J. Module Reuse by Interface Adaptation. In *Software Practice and Experience*, Vol. 21, No. 6, 1991.

[Ped89] Claus H. Pedersen. Extending ordinary inheritance schemes to include generalization. In *OOPSLA ‘89 Conference Proceedings*, pp. 407-417, New Orleans, LA. Published as *SIGPLAN Notices 24(10)*, October 1989.

[PS90] Jens Palsberg and Michael Schwartzbach. Type substitution for object-oriented programming. In *ECOOP/OOPSLA ‘90 Conference Proceedings*, pp. 151-160, Ottawa, Canada, October 1990.

[San86] David Sandberg. An Alternative to Subclassing. In *OOPSLA ‘86 Conference Proceedings*, pp. 424-428, Portland, OR, October 1986. Published as *SIGPLAN Notices 21(11)*, November 1986.

[Som92] Ian Sommerville. *Software Engineering*. 4th ed., Addison-Wesley, 1992.

[SU96] Randall B. Smith and David Ungar. A Simple and Unifying Approach to Subjective Objects. *Theory and Practice of Object Systems 2(3)*:161-178, 1996.

[Weg96] Peter Wegner. Interoperability. In *ACM Computing Surveys*, Vol. 28, No. 1, 1996.

[Win79] Terry Winograd. Beyond programming languages, In *Communications of the ACM*, 22:7, pages 391-401, July, 1979.

[W+93] Robert Wahbe, Steven Lucco, Thomas E. Anderson, Susan L. Graham. Efficient Software-Based Fault Isolation. *SOSP 1993*, pp. 203-216.

[YS97] Daniel M. Yellin and Robert E. Strom. Protocol Specifications and Component Adaptors. IBM T.J. Watson Research Center. In *ACM Transactions on Programming Languages and Systems*, Vol. 19, No. 2, March 1997, pages 292-333.