Implementing Binary Component Adaptation for Java

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

Technical Report TRCS98-21
22 August, 1998

Abstract. This paper gives a detailed description of our implementation of binary component adaptation (BCA) [KH98] for Java. We describe the adaptation specification, its translation into the delta file, and how class files are modified during class loading. We also explain how we integrated BCA into the JDK1.1.5 and how we modified javac to compile against adapted classes.

1. Introduction

Binary component adaptation (BCA) [KH98] is a mechanism that modifies existing components (such as Java class files) to the specific needs of a programmer. Binary component adaptation allows components to be adapted and evolved in binary form and on the fly (during program loading). That is, a programmer can add methods and fields to classes and interfaces, rename and reimplement methods, and alter the type hierarchy even without access to source code

BCA rewrites component binaries while they are loaded. This rewriting is possible if binaries contain enough symbolic information (as Java class files do). 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*. 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 (e.g., 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.

In the remainder of this paper, we discuss our implementation of BCA for the JDK 1.1.5. Section 2 describes the adaptation specification. Section 3 covers its translation into the delta file and section 4 describes the delta file format. Section 5 discusses the core of the BCA system, namely the class loader and modifier. Section 6 describes the integration of the BCA into Sun’s JDK 1.1.5 VM.

1.1 Overview

Figure 1 illustrates the general structure of BCA system integrated into the Java Virtual Machine (JVM). The class loader reads the original binary representation of a class (*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 a byte stream. It contains enough high-level information about the underlying program to allow inspection and modification of its structure. The file includes code (*bytecodes*), a symbol table (*constant pool*), and other ancillary information required to support key features such as safe execution (verification of untrusted binaries), dynamic loading and linking, and reflection. Unlike other object file formats, type information is preserved for the complete set of object types that are present 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 for all methods to verify that all potential callers of a method indeed pass arguments of the correct type. Similarly, method names are required to enable dynamic linking.

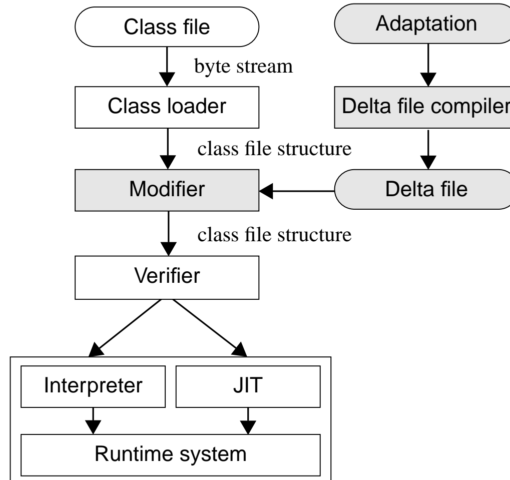


Figure 1. Overview of a Binary Component Adaptation system

The class 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, this internal representation would be passed 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* which is read in by the modifier at VM start-up. (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 an *adaptation specification*, which is compiled to a binary format (delta file) to process it more efficiently at load time.

After modification, the changed class representation is passed to the verifier which checks that the code does not violate any constraints imposed by the JVM and therefore can safely be executed. 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.

2. Adaptation Specification

An adaptation specification describes changes to certain classes that are applied during class loading. In general, an adaptation specification includes Java source code fragments for added or reimplemented methods. The complete grammar for the adaptation specification is described in Appendix A. An adaptation has the following structure:

ImportDeclsopt `delta` *SimpleName* `adapts` *SpecifierDecl UsesDeclopt* { *ModificationDeclsopt* }

2.1 ImportDecls

ImportDecls contains a list of import statements. An import declaration allows a type declared in another package to be referred to by a simple name. Note that currently the import statements affect *only* the body of modifications but not on the type declarations used in modification declarations. That is, each *NameDecl* must be fully qualified.

2.2 SpecifierDecl

The class files specifier defines the set of classes for which the adaptation is applied:

Class Files Specifier	Description
`class` *NameDecl*	class file that represents specific class
`interface` *NameDecl*	class file that represents specific interface
`implements` *NameDecl*	all class files (classes and interfaces) that implement the interface

2.3 UsesDecl

The *UsesDecl* defines a set of deltas that the adaptation depends on. It is used when an adaptation requires classes that are itself adapted by one or more deltas. For an example see the `BinaryClass` adaptation (which depends on the `Main` delta) in section 7.1.

2.4 ModificationDecls

Currently, an adaptation can include the modifications as listed in Table 2.

	Productions
fields	`add field` *AccessFlagDeclsopt TypeDecl Identifier*
	`rename field` *Identifier* `to` *Identifier*
	`rename reference field` *NameDecl Identifier* `to` *Identifier*
methods	`add method` *AccessFlagDeclsopt TypeDecl Identifier* `(` *FormalParameterListDeclopt* `)` *ThrowsDeclopt MethodBodyDeclopt*
	`rename method` *TypeDecl Identifier* `(` *FormalParameterListDecl* `)` `to` *Identifier*
	`rename reference method` *NameDecl TypeDecl Identifier* `(` *FormalParameterListDecl* `)` `to` *Identifier*
interfaces	`add implements` *NameDecl*

3. Delta file Compilation

The delta file compiler translates a textual *adaptation specification* into a compact binary *delta file*. 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. This translation is necessary so that the modifier can perform the changes efficiently during class loading. The delta file compiler consists of a lexical analyzer, a parser, an environment builder, and a code generator. It is written in Java.

As depicted in figure 2 the delta file compilation includes various stages. To illustrate the process we use the following adaptation specification which add a field and a method as an example:

```
delta StringPrintable adapts class java.lang.String {
  add field static float f;
  add method public void print() {
    System.out.println(“hello”);
  };
}
```

3.1 Lexing and Parsing

The first step in creating the delta file consists of parsing the adaptation specification and generating a corresponding parse tree. We used JLex [Ber97] to build a lexical analyzer for the adaptation specification. The lexer breaks up the input stream of characters into tokens and then passes each token to the parser. To build a parser for our adaptation language we used CUP [Hud97] which is an LALR parser generator.

Lexing and then parsing the example adaptation produces the following structure:

```
delta: StringPrintable
imports: (none)
adapts: java.lang.String
uses: (none)

ModificationDecls:
```

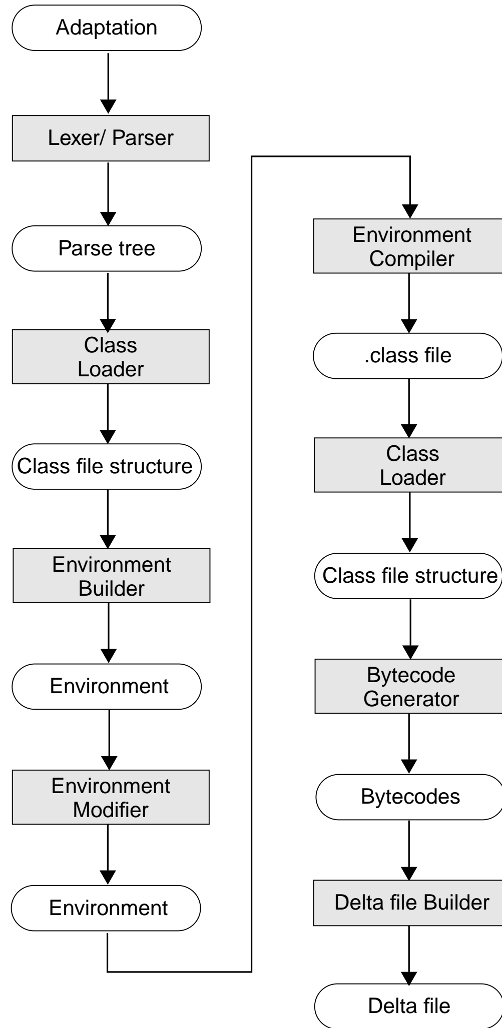
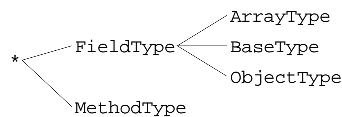


Figure 2. Delta file compilation

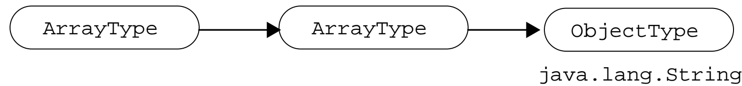
```
0: AddFieldDecl:
    AccessFlagDecls: static
    field type: BaseType: float
    field name: f
1: AddMethodDecl:
    AccessFlagDecls: public
    return type: BaseType void
    method name: print
    parameterdecls:
    throws:
    body: { System.out.println ("hello"); }
```

The parse tree represents the adaptation specification in an easy-to-access structure. The parser recognizes access modifiers and places them in a list of individual modifiers. Types used in the adaptation are represented in the following way:



For each instance of a type the parser build a corresponding type graph. Object types and base types (primitive types) are leaf nodes in the type graph. Array types contain a reference to their element type. A field type is any one of an object type, an array type, or a base type. A method type contains a list of field types for its formal parameters and another field type for this return type. For convenience, the `void` type has been added to the set of primitive types (even though this does not conform to the JVM specification). The `void` type is only legal to indicate that a method has no return type.

For example the type `java.lang.String[][]` is represented by the following type graph:



3.2 Building the Environment

For simplicity, we use `javac`, the standard Java compiler, to generate bytecodes for new methods. Since `javac` requires a Java source file as its input, we generate a Java declaration (what we call *environment*) of the class to be adapted and then extend it by the new methods. The environment serves as a wrapper so that new methods can be compiled using the standard Java compiler. The delta file compiler invokes `javac` to compile the environment into a binary class file from which it extracts the new bytecodes.

The environment is initially generated from the type information found in the original class file. The delta file compiler loads the original class file and generates a textual representation of the class. For our example the initial environment looks like this:

```
package java.lang;

class String extends java.lang.Object implements java.io.Serializable {
  private char[] value;
  private int offset;
  private int count;
  private static final long serialVersionUID = -6849794470754667710L;
  // ...

  public String() { super(); };
  public String(java.lang.String p0) { super(); };
  // ...

  public native int length();
  public void getChars(int p0, int p1, char[] p2, int p3) { };
  // ...

}
```

Note that the initial environment does not include implementations for methods and constructors. Instead methods with no return type and constructors have an empty body. Other methods are defined `native` so that no body is required for compilation. In Java every constructor either invokes another constructor in the same class, or a constructor in the direct superclass. If the first statement of a constructor body is not an explicit invocation of another constructor, then the constructor is implicitly assumed to begin with a “`super();`”, an invocation of the constructor of its direct superclass that takes no arguments. Since the direct superclass may not have a default constructor, the environment builder also loads the direct superclass of the initial environment. It then looks up a constructor and generates a `super` statement taking null values as arguments.

3.3 Modifying the Environment

The environment modifier incorporates all changes described in the adaptation specification into the initial environment. For example, if the adaptation specification contains a modification to include a new method, the environment modifier adds a corresponding method declaration (including a body for the Java code) to the environment. It also adds fields to the environment because a new method may depend on it.

For the example adaptation, the environment after incorporating the changes looks as follows:

```
package java.lang;

class String extends java.lang.Object implements java.io.Serializable {
  private char[] value;
  private int offset;
  private int count;
  private static final long serialVersionUID = -6849794470754667710L;

  static float f;
  // ...

  public String() { super(); };
  public String(java.lang.String p0) { super(); };
  // ...

  public native int length();
  public void getChars(int p0, int p1, char[] p2, int p3) { };
  // ...

  public void print() { System.out.println ("hello"); } ;
}
```

3.4 Compiling the Environment

The environment compiler translates the final environment into a class file by invoking the standard Java compiler. If the compilation succeeds, `javac` generates a class file containing the bytecodes for new methods and constructors. A bytecode generator loads the generated class file and extracts the bytecodes from the class file.

3.5 Representation of Bytecodes

The bytecode generator retrieves the bytecodes from the compiled class and then converts each instruction into a constant-pool-independent representation. First let's take a look how the compiled bytecodes for the `print` method in the environment look like:

```
magic: 0xcafebabe
minor: 3 major: 45
constant pool count: 113
1: StringConstant string: 85
2: ClassConstant name: 88
7: ClassConstant name: 93
9: FieldrefConstant class: 7 nameandtype: 14
10: MethodrefConstant class: 2 nameandtype: 15
14: NameAndTypeConstant name: 97 desc: 68
15: NameAndTypeConstant name: 99 desc: 39
39: "(Ljava/lang/String;)V"
68: "Ljava/io/PrintStream;"
85: "hello"
88: "java/io/PrintStream"
93: "java/lang/System"
97: "out"
99: "println"
...

Method: public print()V
  CodeAttribute
    maxStack: 2
    maxLocals: 1
```

```
    code length: 9
        0x0000: b2 00 09 12 01 b6 00 0a b1
      disassembled instructions:
      0: getstatic 9
      3: ldc 1
      5: invokevirtual 10
      8: return
```

Since the bytecodes contain references to entries that are specific to the constant pool for the compiled class, the bytecodes must be transformed into a constant-pool-independent format in order to plug them into an arbitrary class file.

We use two passes to obtain a constant-pool-independent format of bytecodes:

1. The bytecode generator widens all instructions so that only wide constant pool references are used.
2. Each reference to the constant pool is resolved and stored in a table.

3.5.1 Instruction Widening

The JVM supports both one- and two-byte constant pool addressing. Patching 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 the operand that addresses the constant pool. Inserting bytes into the code moves jump targets, and therefore jump addresses need corrections. Furthermore, instructions requiring 4-byte operand alignment (e.g., `tableswitch` and `lookuptable)` need pad byte adjustments.

To prevent these problems, we require that any code added to a class uses 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. At runtime, the modifier walks through a table of constant pool references and updates each two-byte index with the actual index into the constant pool of the modified class.

The instruction widening is implemented by two passes: The first pass calculates the new instruction layout by converting every `ldc` into `ldc_w` and realigning instruction operands by adjusting pad bytes. The offsets of the changed instructions are stored in a table. The second pass generates the bytecodes as determined in the previous pass and recalculates jump targets.

After instruction widening, the code array for the `print` method looks as following:

```
Method: public print()V
  CodeAttribute
    maxStack: 2
    maxLocals: 1
    code length: 10
        0x0000: b2 00 09 13 00 01 b6 00 0a b1
      disassembled instructions:
      0: getstatic 9
      3: ldc_w 1
      6: invokevirtual 10
      9: return
```

Note that `ldc` is replaced by `ldc_w` and that the code length increased by one byte.

3.5.2 Reference Resolver

In the second pass, the bytecode generator resolves every constant pool reference and stores the address of the constant pool entry together with the constant value in a table. After resolving all constants, the bytecode representation for method `print` looks as follows:

```
maxStack: 2
```

```
maxLocals: 1
code length: 10
  0x0000: b2 00 00 13 00 00 b6 00 00 b1
reference table:
 (1, CpFieldRef: CpClassRef: java/lang/System out Ljava/io/PrintStream;)
 (4, CpStringRef: "hello")
 (7, CpMethodRef: CpClassRef: java/io/PrintStream println (Ljava/lang/String;)V)
disassembled instructions:
  0: getstatic
  3: ldc_w
  6: invokevirtual
  9: return
```

4. Internal Representation of the Delta File Format

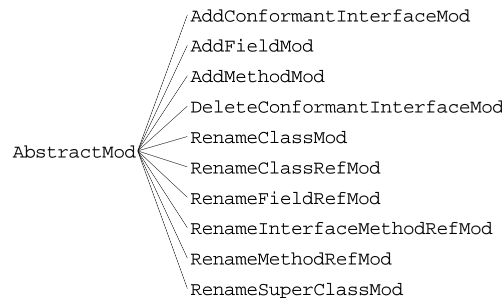
The delta file contains a list of modifications:

```
class DeltaFile {
private:
  DynamicArray<AbstractMod>* _modtable;
  ...
};
```

Each modification describes an individual action that the modifier must perform on the class file (such as adding a new field or renaming a method). `AbstractMod` is the abstract base class from which every modification derives:

```
class AbstractMod {
protected:
  Precond* _precond;
...
};
```

Currently, the following modifications exist:



The precondition field is part of every modification. It defines the property that the class must fulfill such that the modifier is allowed 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. Possible preconditions include a *specific class name*, *class conforms to a specific interface*, or simply *true* if every class needs modification (e.g., to update references after renaming).

Since the layout of the constant pool cannot be determined until the class is actually modified, bytecodes must be delivered in a constant pool independent format. For that reason the reference table annotates all constant pool references by the explicitly resolved constant. The bytecode representation has the following structure:

```
class CodeRep {
private:
  u2  _max_stack;
  u2  _max_locals;
  u4  _code_length;
  u1* _code;
```

```
  DynamicArray<ReferenceEntry>* _reference_table;
  DynamicArray<Exception>*     _exception_table;
  ...
};
```

The reference table stores the byte offset of each missing constant pool index together with the constant. At load time, the modifier can then update all constant pool reference in the code array.

For the example adaptation show in section 3 the delta file looks as following:

```
modification table count: 2
0: AddFieldMod: NamedClassPrecondition: java/lang/String static f F
1: AddMethodMod: NamedClassPrecondition: java/lang/String public print ()V throws: -
   maxStack: 2
   maxLocals: 1
   code length: 10
     0x0000: b2 00 00 13 00 00 b6 00 00 b1
   reference table:
    (1, CpFieldRef: CpClassRef: java/lang/System out Ljava/io/PrintStream;)
    (4, CpStringRef: "hello")
    (7, CpMethodRef: CpClassRef: java/io/PrintStream println (Ljava/lang/String;)V)
   disassembled instructions:
     0: getstatic
     3: ldc_w
     6: invokevirtual
     9: return
```

This delta file includes two modifications, namely `AddFieldMod` and `AddMethodMod`. Modification `AddFieldMod` describes a new field `f` for class `java.lang.String`. Modification `AddMethodMod` defines a method `print` including the bytecodes for the implementation and a reference table for all constant pool entries.

5. BCA Core System

This section describes our implementation of binary component adaptation for Java in detail. We designed our implementation for easy integration into Sun's JDK VM rather than for ultimate speed. The BCA system specifies a simple application programming interface (API). It reads the original class file from a memory buffer and returns a memory buffer representing the adapted class file. This interface allows the BCA system to be easily integrated into any system (such as a virtual machine or compiler) since it is independent of any in-memory representation of class files. Specifically, we do not depend on the in-memory class file structure used in the JDK VM. This allows to integrate BCA into future releases of the VM even if data structures changes. On the downside, since the BCA system does not depend on any intermediate class file structure, it must provide its own class loader.

5.1 Loading the Class File

The class loader retrieves the Java class file and stores the complete information in an in-memory representation. The original structure of the class file format remains unchanged and is not further processed. The class file keeps the constant pool and indices into the constant pool (i.e., the constant pool is not *resolved*). Furthermore, the class loader preserves type descriptors in their representation as UTF-8 strings.

The class loader is architecture independent. Numerical values are handled correctly regardless whether the underlying architecture supports big- or little-endian byte ordering.

A class file represents either a Java class or interface type. It includes the name of the class and superclass, the access rights, a list of implemented interfaces, a method and field table, attributes and a constant pool. The mapping of the class file format [LY96] to C++ structures is straightforward. Each component such as a constant pool entry, interface, field, method, or attribute is represented by a separate C++ class. The class file structure looks as follows:

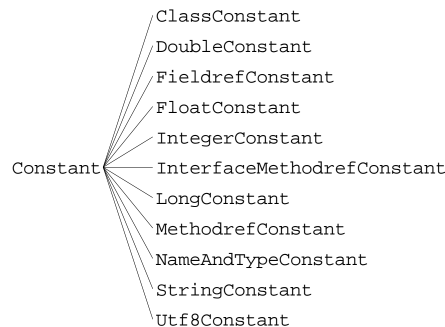
```
class Classfile {
```

```
private:
  u4 _magic;
  u2 _major_version;
  u2 _minor_version;

  u2 _access_flags;
  u2 _this_class;
  u2 _super_class;

  DynamicArray<Constant> _constant_pool;
  DynamicArray<Interface> _interfaces;
  DynamicArray<Field> _fields;
  DynamicArray<Method> _methods;
  DynamicArray<Attribute> _attributes;
  ...
};
```

The constant pool is a table representing various string constants, class names, field names, and other constants that are referred to within the class file structure and its substructures. Each entry in the constant pool is a subclass of `Constant`. The constant hierarchy looks as follows:



The set of interfaces to which a classfile conforms is stored in interface structures. Each interface structure includes a single constant pool reference to the name of the interface:

```
class Interface {
private:
  u2 _nameindex;
  ...
};
```

The field table contains all class or instance variables that the class file declares. Each field structure is composed of the access modifiers, field name and type descriptor, and attributes. The only attribute defined for fields is the `ConstantValue` attribute used for field initialization.

```
class Field {
private:
  u2 _access_flags;
  u2 _name_index;
  u2 _descriptor_index;
  DynamicArray<Attribute> _attributes;
  ...
};
```

Each method and each constructor is described by a method structure. A method structure includes the access modifiers, the method name and signature, and various attributes.

```
class Method {
private:
  u2 _access_flags;
  u2 _name_index;
```

```
  u2 _descriptor_index;
  DynamicArray<Attribute> _attributes;
  ...
};
```

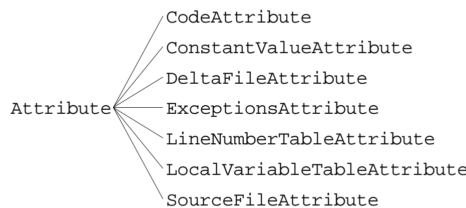
A method can have any number of optional attributes associated with it. Methods that are neither abstract nor native contain a single code attribute:

```
class CodeAttribute : public Attribute {
private:
  u2 _max_stack;
  u2 _max_locals;
  u4 _code_length;
  u1* _code;

  jvm_DynamicArray<Exception> _exception_table;
  jvm_DynamicArray<Attribute> _attributes;
  ...
};
```

A code attribute contains the Java Virtual Machine instructions as well as auxiliary information about the method (such as maximum operand stack depth and number of locals variables, exception handlers and optional attributes for debugging information). The actual JVM instructions are represented as bytes in the code array.

Attributes are used in various structures and substructures within the class file. The `DeltaFileAttribute` is an attribute type introduced for BCA's version control mechanism to resolve name conflicts. It represents a list of delta files that the class used for compilation. The `Exceptions` attribute is used to indicate which checked exceptions a method may throw.



5.2 Modifying the Class File

Modifying the class file structure involves adding new parts (i.e., 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 constant pool (or adds them if not present) and sets the indices in the field structure.

Adding a method to a class is similarly simple: the modifier extends the method table by a method structure and extends the constant pool by the method name and descriptor. If the method is neither abstract nor native, the modifier also includes a code attribute.

JVM instructions reference the constant pool. However, the exact constant pool layout is unknown until the class is modified. Therefore, constant pool references must be determined at load-time. For that reason, the delta file contains all explicitly resolved constants in a reference table. At load time, the modifier walks through the constant pool reference table, adds constants to the constant pool if required and updates the index for the JVM instructions.

However, one complication remains: the JVM supports one- and two-byte constant addressing. Updating an instruction operand that is only one-byte, may lead to an overflow if the constant pool already has 255 or more entries. For that reason, the modifier expects that added byte codes use only two-byte constant pool addressing, ensuring that the index can always be updated. Instruction widening is described in section 3.5.1.

5.3 Linearizing the Class File

After the modifications, the changed class representation is stored in a memory buffer in the standard class file representation. A native class loader (e.g., VM loader) can then retrieve the class file from this memory buffer.

6. Integrating BCA into the Virtual Machine

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 to the native JDK loader. This organization is simple to implement since modifications are invisible to the standard virtual machine (VM). Thus the VM needs only minor modifications to insert BCA into the loading process.

The main disadvantage of this approach is its higher overhead; the class file is actually parsed twice, once by the BCA system 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 retrieves the modified class. Clearly, the performance of 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., porting BCA to newer versions of the JDK).

The BCA system provides a stub function with the following interface:

```
void bcastub(u1* rdbuf, u4 rdbufsize, u1* wrbuf, u4* wrbufsize, char* deltas);
```

The BCA system retrieves the class file from a memory buffer beginning at address `rdbuf` and length `rdbufsize`. The parameter `deltas` contains a list of delta files separated by colons. After modification, the BCA system returns the adapted class in a buffer located at `wrbuf` and length `wrbufsize`.

To integrate the BCA into the loading process, we modified the standard VM class loader so that it calls the BCA system before it generates its intermediate class representation. This step is accomplished by calling `bcastub` in the file `classloader.c`.

```
bool_t
createInternalClass(unsigned char *ptr, unsigned char *end_ptr, ClassClass *cb,
                    struct Hjava_lang_ClassLoader *loader, char *name, char **detail)
{
  struct CICcontext context_block;
  struct CICcontext *context = &context_block;
  struct Classjava_lang_Class *ucb = unhand(cb);

  u1* wrbuf;
  u4 wrbufsize;

  bcastub(ptr, end_ptr - ptr, wrbuf, &wrbufsize, globalDeltas);

  ptr = wrbuf;
  end_ptr = wrbuf + wrbufsize;

  /* now create internal class representation */
  ...
}
```

The function `bcastub` retrieves the original class from the memory buffer, modifies it according to the delta files given by `globalDeltas`, and writes a new class file into a memory buffer. The native class loader obtains the modified class from that memory buffer.

7. 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.

7.1 Using BCA to Modify javac

We integrated BCA into the standard Java compiler `javac` by modifying its class loader. 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. Since `javac` is written in Java, we can use BCA to make the corresponding modifications on `javac`'s class files. Therefore, we wrote two adaptation specifications such that `javac` accepts a new command line argument `-deltas` and passes every class to the adaptation system before it retrieves type information.

The following adaptation specification changes `sun.tools.javac.Main` such that it recognizes the `-deltas` command line argument:

```
import java.io.FileOutputStream;
import java.util.*;

delta Main adapts class sun.tools.javac.Main {
  add field public static java.lang.String deltasString;
  rename method boolean compile(java.lang.String[] argv) to compileold;
  add method public synchronized boolean compile(java.lang.String[] argv) {

    deltasString = "";
    String[] newargv = argv;
    for (int i = 0 ; i < argv.length ; i++) {
      if (argv[i].equals("-deltas")) {
        if ((i + 1) < argv.length) {
          deltasString = argv[i+1];
          newargv = new String[argv.length - 2];
          for (int j=0; j < i; j++) {
            newargv[j] = argv[j];
          }
          for (int j=i+2; j < argv.length; j++) {
            newargv[j-2] = argv[j];
          }
          i++;
        } else {
          error("-deltas requires argument");
        }
      }
    }
    return compileold(newargv);
  };
}
```

This adaptation specification adds a new variable `deltasString` to class `Main` to store the delta command line argument. It also renames method `compile` to `compileold` and then adds a new method `compile` which parses the arguments and stores the delta argument in the variable `deltasString`. After the argument parsing the original implementation of `compile` (now called `compileold`) is invoked.

Integrating the adaptation system into the loading process is relatively simple as well. It requires only a few lines that invoke the BCA system via a native method call. The adaptation for class `sun.tools.java.BinaryClass` looks as following:

```
import java.io.*;
import java.util.*;

delta BinaryClass adapts class sun.tools.java.BinaryClass uses Main {
  add method
  public static native byte[] calldynloader(byte[] rdbuf, java.lang.String deltas);
```

```
    rename method
    sun.tools.java.BinaryClass load (sun.tools.java.Environment env,
                                     java.io.DataInputStream in,
                                     int mask) to old_load;

    add method
    public static sun.tools.java.BinaryClass
    load(sun.tools.java.Environment env, java.io.DataInputStream in, int mask)
      throws java.io.IOException {

      System.loadLibrary("jci");

      String deltas = sun.tools.javac.Main.deltasString;
      ByteArrayOutputStream byteout = new ByteArrayOutputStream();

      try {
        for (;;) {
          byteout.write(in.readByte());
        }
      } catch (EOFException e) {
        byte outbuf[] = calldynloader(byteout.toByteArray(), deltas);

        ByteArrayInputStream bytein = new ByteArrayInputStream(outbuf);
        return old_load(env, new DataInputStream(bytein), mask);
      }
    };
  }
```

This adaptation specification first adds a native method `calldynloader`, the stub function for the BCA system. The implementation for this method is wrapped into `libjci.so`, a dynamic library that contains the BCA system. In Java, the library is then loaded by a call to `System.loadLibrary`[1].

Then, the adaptation specification renames method `load` to `loadold` and adds a new `load` method that inserts the adaptation system into the loading process before it invokes `loadold`. Note that this adaptation specification uses the new variable `deltasString` introduced by the `Main` adaptation. This dependency is expressed by the `uses` clause.

The adaptation specifications contain some relatively simple changes for the classes `sun.tools.javac.Main` and `sun.tools.java.BinaryClass`. Since the changes are decoupled from javac's classes, the delta files can also be used for a newer release of javac.

References

[Ber97] Elliot Berk. *JLex: A lexical analyzer generator for Java*. Department of Computer Science, Princeton University, 1997.

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

[Hud97] Scott E. Hudson. *CUP User's Manual.* Graphics Visualization and Usability Center. Georgia Institute of Technology, 1997.

[KH97] Ralph Keller and Urs Hölzle. Supporting the Integration and Evolution of Components Through Binary Component Adaptation. *Technical Report TRCS97-15*, Department of Computer Science, University of California, Santa Barbara, September 1997.

[KH98] Ralph Keller and Urs Hölzle. Binary Component Adaptation. *Proceedings of ECOOP'98,* Lecture Notes in Computer Science, Springer Verlag, July 1998.

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

[1] `libjci.so` must be included in the library search path (`LD_LIBRARY_PATH`).

Appendix A Adaptation Specification Grammar

The following subsection presents a grammar for the adaptation specification.

A.1 Grammar Notation

For the grammar, we use the same notation as in chapter 19 of the Java Language Specification [GJS96]. Terminal symbols of the grammar are shown in `fixed-width` font. Nonterminal symbols are shown in *italic* type. The definition of a nonterminal is introduced by the name of the nonterminal, followed by a colon. One or more alternative right-hand sides for the nonterminal then follow on succeeding lines. Nonterminals that are not specified are identical to the Java Language Specification.

A.2 Productions from Adaptation Specification

CompilationUnit:

ImportDeclsopt `delta` *SimpleName* `adapts` *SpecifierDecl UsesDeclopt* { *ModificationDeclsopt* }

A.2.1 Import Declarations

ImportDecls:

ImportDecls ImportDecl `;`

ImportDecl:

SingleTypeImportDecl

TypeImportOnDemandDecl

SingleTypeImportDecl:

`import` *NameDecl*

TypeImportOnDemandDecl:

`import` *NameDecl* `. *`

A.2.2 Class files Specifier Declaration

SpecifierDecl:

ClassSpecifierDecl

InterfaceSpecifierDecl

ImplementsSpecifierDecl

ClassSpecifierDecl

`class` *NameDecl*

InterfaceSpecifierDecl

`interface` *NameDecl*

ImplementsSpecifierDecl

`implements` *NameDecl*

A.2.3 Uses Declaration

UsesDecl:

`uses` *ClassTypeListDecl*

A.3 Productions from Modification Declarations

ModificationDecls:

ModificationDecls ModificationDecl `;`

ModificationDecl:

AddFieldDecl

AddMethodDecl

RenameFieldDecl

RenameMethodDecl

RenameFieldRefDecl

RenameMethodRefDecl

AddImplementsDecl

AddFieldDecl:

`add` `field` *AccessFlagDeclsopt TypeDecl Identifier*

AddMethodDecl:

`add` `method` *AccessFlagDeclsopt TypeDecl Identifier* `(` *FormalParameterListDeclopt* `)` *ThrowsDeclopt MethodBodyDeclopt*

RenameFieldDecl:

`rename` `field` *Identifier* `to` *Identifier*

RenameMethodDecl:

`rename` `method` *TypeDecl Identifier* `(` *FormalParameterListDecl* `)` `to` *Identifier*

RenameFieldRefDecl:

`rename` `reference` `field` *NameDecl Identifier* `to` *Identifier*

RenameMethodRefDecl:

`rename` `reference` `method` *NameDecl TypeDecl Identifier* `(` *FormalParameterListDecl* `)` `to` *Identifier*

AddImplementsDecl:

`add` `implements` *NameDecl*

A.3.1 Productions from AddMethodDecl

FormalParameterListDecl:

ProperFormalParameterListDecl

ProperFormalParameterListDecl:

ProperFormalParameterListDecl `,` *FormalParameterDecl*

FormalParameterDecl

FormalParameterDecl:

TypeDecl Identifier

ThrowsDecl:

`throws` *ClassTypeListDecl*

ClassTypeListDecl:

ClassTypeListDecl `,` *ClassTypeDecl*

ClassTypeDecl

ClassTypeDecl:

NameDecl

MethodBodyDecl:

{ *BlockDecl* }

BlockDecl: any legal Java method body

A.4 Productions from Access Modifers

AccessFlagDecls:

  AccessFlagDecls AccessFlagDecl

AccessFlagDecl: one of

```
public protected private
static
abstract final native synchronized transient volatile
```

A.5 Productions from Type Declarations

TypeDecl:

  BaseTypeDecl

  ReferenceTypeDec

BaseTypeDecl: one of

```
byte char double float int long short boolean void
```

ReferenceTypeDecl:

  ArrayTypeDecl

  ClassOrInterfaceTypeDecl

ClassOrInterfaceTypeDecl:

  NameDecl

ArrayTypeDecl

  BaseTypeDecl `[ ]`

  NameDecl `[ ]`

  ArrayTypeDecl `[ ]`

A.6 Productions from Names

NameDecl:

  SimpleNameDecl

  QualifiedNameDecl

SimpleNameDecl:

  Identifier

QualifiedNameDecl:

  NameDecl `.` *Identifier*