USENIX Association

Proceedings of the Third Virtual Machine Research and Technology Symposium

San Jose, CA, USA
May 6–7, 2004

USENIX
THE ADVANCED COMPUTING SYSTEMS ASSOCIATION

MCI-Java: A Modified Java Virtual Machine Approach to Multiple Code Inheritance

Maria Cutumisu, Calvin Chan, Paul Lu and Duane Szafron
Department of Computing Science, University of Alberta
{meric, calvinc, paullu, duane}@cs.ualberta.ca

Abstract

Java has multiple inheritance of interfaces, but only single inheritance of code via classes. This situation results in duplicated code in Java library classes and application code. We describe a generalization to the Java language syntax and the Java Virtual Machine (JVM) to support multiple inheritance of code, called MCI-Java. Our approach places multiply-inherited code in a new language construct called an *implementation*, which lies between an interface and a class in the inheritance hierarchy. MCI-Java does not support multiply-inherited data, which can cause modeling and performance problems. The MCI-Java extension is implemented by making minimal changes to the Java syntax, small changes to a compiler (IBM Jikes), and modest localized changes to a JVM (SUN JDK 1.2.2). The JVM changes result in no measurable performance overhead in real applications.

1 Introduction

Three distinct language concepts are used in object-oriented programs: interface, code and data. The motivation for separate language mechanisms to support these concepts has been described previously [14]. The goal of the research described in this paper is to explicitly support each of these three mechanisms in an extended Java language, to evaluate the utility of concept separation, and to potentially increase demand for separate language constructs in future languages.

Researchers and practitioners commonly use the terms *type* and *class* to refer to six different notions:

- a real-world concept (**concept**)
- a programmatic interface (**interface**)
- the code for an interface (**implementation**)
- an internal machine data layout (**representation**)
- a factory for creation of instances (**factory**)
- a maintainer of the set of all instances (**extent**)

Unfortunately, most object-oriented programming languages do not separate these notions. Each language models the **concept** notion by providing language constructs for various combinations of the other five notions. Java uses *interface* for **interface**. However, it combines the notions of **implementation**, **representation** and **factory** into a *class* construct. Smalltalk and C++ have less separation. They use the same *class* construct for **interface**, **implementation**, **representation** and **factory**.

In this paper we will focus on general-purpose programming languages, so we will not discuss the **extent** notion, which is most often used in database programming languages [15]. We also combine **concept** with **interface**, to enforce encapsulation. Finally, we combine **representation** and **factory**, since **representation** layout is required for creation. This reduces the programming language design space to three dimensions:

- An *interface* defines the legal operations for a group of objects (**interface**) that model a **concept**.
- An *implementation* associates generic code with the operations of an interface (**implementation**) without constraining the data layout.
- A *representation* defines the data layout (**representation**) of objects for an **implementation**, provides data accessor methods and a mechanism for object creation (**factory**).

We use the generic term *type* to refer to an *interface*, an *implementation* or a *representation*.

Inheritance allows properties (*interface*, *implementation* or *representation*) of a type to be used in child types called its *direct subtypes*. By transitivity, these properties are inherited by all *subtypes*. If type B is a subtype of type A, then A is called a *supertype* of type B. If a language restricts the number of direct supertypes of any type to be one or less, the language has *single inheritance*. If a type can have more than one direct supertype, the language supports *multiple inheritance*.

Interface inheritance allows a subtype to inherit the *interface* (operations) of its supertypes. The principle of *substitutability* states that if a language expression contains a reference to an object whose static type is A, then an object whose type is A or any subtype can be used. Interface inheritance relies only on substitutability and does not imply code or data inheritance.

Table 1 Support for single/multiple inheritance and concept separation constructs in some existing languages.

Language	*interface*	*implementation*	*representation*
Java	multiple / interface	single / class	single / class
C++	multiple / class	multiple / class	multiple / class
Smalltalk	single / class	single / class	single / class
Eiffel	multiple / class	multiple / class	multiple / class
Cecil/BeCecil	multiple / type	multiple / object	multiple / object
Emerald	implicit multiple / abstract type	none / object constructor	none / object constructor
Sather	multiple / abstract class	multiple /class	multiple / class
Lagoona	multiple / category	none / methods	single / type
Theta	multiple / type	single / class	single / class

Implementation (code) inheritance allows a type to reuse the *implementation* (binding between an operation and code) from its parent types. Code inheritance is independent of data representation, since many operations can be implemented by calling more basic operations (e.g. accessors), without specifying a representation, until the subtypes are defined. Java and Smalltalk only have single code inheritance, but C++ has multiple code inheritance.

Representation (data) inheritance allows a type to reuse the *representation* (data layout) of its parent types. This inheritance is the least useful and causes considerable confusion if multiple data inheritance is allowed. Neither Java nor Smalltalk support multiple data inheritance, but C++ does.

Table 1 shows the separation and inheritance characteristics of several languages: Java, C++, Smalltalk, Eiffel [17], Cecil [6] and its descendant BeCecil [5], Emerald [20], Sather [22], Lagoona [12] and Theta [9]. This list is not intended to be complete. A more general and extensive review of type systems for object-oriented languages has been compiled [15]. Although there is growing support for the separation of *interface* from *implementation/representation*, the concepts of *implementation* and *representation* are rarely separated at present. We intend to change this. The major research contributions of this paper are:

- The introduction of a new language construct called an *implementation* into the Java programming language. This construct completely separates the two orthogonal concepts of *implementation* (code) and *representation* (data).
- A new multi-super call mechanism that generalizes current Java semantics, rather than using C++ multi-super semantics.
- The first implementation of multiple code inheritance in Java, based on localized modifications to the SUN JDK 1.2.2 JVM, along with minor changes to the syntax of Java and to the IBM Jikes 1.15 compiler. Existing programs still work and suffer no performance penalties.
- A demonstration that multiple code inheritance reduces duplicated and similar code, so program construction and maintenance are simplified.

Our modified multiple code inheritance compiler (*mcijavac*), modified JVM (*mcijava*), the code for the scenarios in this paper, and the code for the `java.io` example are available on-line as MCI-Java [16].

2 Motivation for Multiple Code Inheritance in Java

In this Section we motivate the use of multiple code inheritance using some classes from the `java.io` library. Java currently supports multiple interface inheritance. Consider the Java class `RandomAccessFile` (from `java.io`) that implements the interfaces `DataInput` and `DataOutput`, as shown in Figure 1[1]. Since Java supports substitutability, any reference to a `DataInput` or `DataOutput` can be bound to an instance of `RandomAccessFile`.

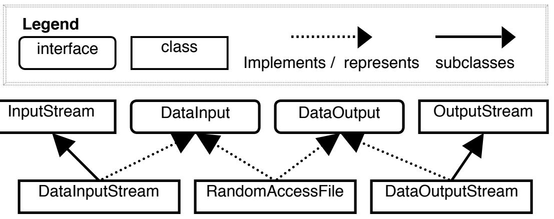

Figure 1. The inheritance structure of some classes and interfaces from the `java.io` library.

However, Java does not support multiple code inheritance. Much of the code that is in `RandomAccessFile` is identical or similar to code in

[1] There are actually two other classes in `java.io`, `FilterInputStream` and `FilterOutputStream`, that are not included in Figure 1, Figure 2, or Figure 5. They have been omitted for simplicity and clarity, since they do not affect the abstractions described in this paper.

DataInputStream and DataOutputStream. Although it is possible to refactor this hierarchy to make RandomAccessFile a subclass of either DataInputStream or DataOutputStream, it is not possible to make it a subclass of both, since Java does not support multiple code inheritance.

This causes implementation and maintenance problems. One common example is that duplicate code appears in several classes. This makes programs larger and harder to understand. In addition, code can be copied incorrectly or changes may not be propagated to all copies. There are many examples of code copying errors in various contexts. For example, code is often cloned (cut-and-pasted) in device drivers for operating systems [7]. If a bug is found in the repeated code, a fix must be applied to each clone. However, if the same code is refactored into a single *implementation* in an object-oriented inheritance hierarchy, then any bug fix or new functionality would only have to be done once.

Two alternative techniques for reducing code duplication are *mixins* [1] and *traits* [21]. These approaches are discussed in Section 9, where they are contrasted with our multiple code inheritance technique.

2.1 Duplicate Method Promotion

The methods writeFloat(float) and writeDouble(double) are examples of duplicate methods that appear in both DataOutputStream and RandomAccessFile. There are also four methods that have identical code in DataInputStream and RandomAccessFile.

Once these duplicate methods have been found, how can code inheritance be used to share them? Figure 1 shows that we need a common ancestor type of DataInputStream and RandomAccessFile to store the four common read methods and a common ancestor type of DataOutputStream and RandomAccessFile to store the two common write methods. To share code, this ancestor cannot be an interface. It also cannot be a class, since we would need multiple code inheritance of classes and Java does not support it. In fact, it should be a multiple inheritance *implementation*. Figure 2 shows the common code factored into two *implementations*: InputCode and OutputCode. In this approach, an *implementation* provides common code for multiple classes.

The benefit of using *implementations* to promote *duplicate methods* may seem questionable to re-use only six methods. However, it is not only such duplicate methods that can be promoted higher in the inheritance hierarchy. Multiple code inheritance can also be used to factor some non-duplicate methods, if they are abstracted slightly. We have used three additional code promotion techniques to factor non-duplicate methods.

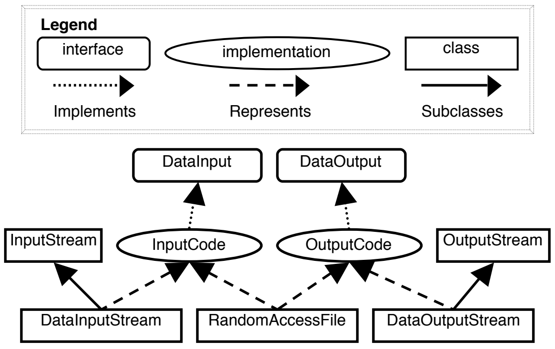

Figure 2. Adding *implementations* to Java, for multiple code inheritance.

2.2 Prefix Method Promotion

The first technique is called *prefix method promotion*. It applies when a class dependent computation is done at the start of the method and the rest of the method is identical. Consider the readByte() methods shown in Figure 3, from classes DataInputStream and RandomAccessFile. These methods differ only by a single line of code.

```
// This method is in DataInputStream
public final byte readByte() throws IOException
{
   int ch = this.in.read();
   if (ch < 0)
      throw new EOFException();
   return (byte)(ch);
}

// This method is in RandomAccessFile
public final byte readByte() throws IOException
{
   int ch = this.read();
   if (ch < 0)
      throw new EOFException();
   return (byte)(ch);
}

// This method replaces the previous two, and
// is in InputCode
public final byte readByte() throws IOException
{
   Source in = this.source();
   int ch = in.read();
   if (ch < 0)
      throw new EOFException();
   return (byte)(ch);
}
```

Figure 3. An example of using the prefix technique to promote methods.

Figure 3 shows a single common method promoted to an *implementation* called InputCode, which replaces both. This abstraction requires the creation of a new interface, Source, that contains one abstract method, read(). It also requires a method called source() to be declared in InputCode. A default method with code, `return this`, is created in `InputCode` and it is inherited by class `RandomAccessFile`. The implementation of this method is overridden in class `DataInputStream` with the code, `return this.in`. This prefix technique can be used to promote seven other methods in the same two classes.

2.3 Super-Suffix Method Promotion

The second technique is called *super-suffix method promotion*. It can be used to move similar methods from `DataOutputStream` and `RandomAccessFile` to the common *implementation*, `OutputCode`. Consider methods for `writeChar(int)` that appear in classes `DataOutputStream` and `RandomAccessFile`, as shown in Figure 4.

```
// The original method in DataOutputStream
public final void writeChar(int v) throws
IOException {
   this.out.write((v >>> 8) & 0xFF);
   this.out.write((v >>> 0) & 0xFF);
   incCount(2);
}

// The original method in RandomAccessFile
public final void writeChar(int v) throws
IOException {
   this.write((v >>> 8) & 0xFF);
   this.write((v >>> 0) & 0xFF);
}

// The method in OutputCode that replaces the
// one in RandomAccessFile and does most of the
// computation for the one in DataOutputStream
public final void writeChar(int v) throws
IOException {
   Sink out = this.sink();
   out.write((v >>> 8) & 0xFF);
   out.write((v >>> 0) & 0xFF);
}

// The new method in DataOutputStream
public final void writeChar(int v) throws
IOException {
   super(OutputCode).writeChar(v);
   incCount(2);
}
```

Figure 4. Examples of super-suffix methods that can be reused in different classes.

To replace these methods by a common method, we substitute each first line by a common abstracted line, analogous to the previous example. This abstraction requires the creation of a new interface, `Sink`, that contains one abstract method, `write(int)`. It also requires a method called `sink()` to be declared in `OutputCode`.

However, there is another problem that must be solved before we can promote `writeChar(int)`. This method has an extra line of code in class `DataOutputStream`, which does not appear in class `RandomAccessFile`. Fortunately, we can promote all lines except for this *suffix* line into a common method in `OutputCode`. This eliminates `writeChar(int)` from `RandomAccessFile`. However, in `DataOuputStream` we need to include the missing last line using a classic refinement technique that makes the super call shown in Figure 4.

Note that `super(OutputCode)` is **not** standard Java. It calls a method in the *superimplementation*, `OutputCode`, instead of calling a method in a superclass. In general, since there may be multiple immediate *superimplementations*, the super call must be qualified by one of them. This is one of the standard approaches to solving the super ambiguity problem of multiple inheritance and it will be discussed later in this paper. We can use this super-suffix technique to promote a total of six similar methods that appear in `DataOuputStream` and `RandomAccessFile`.

2.4 Static Method Promotion

The third technique for promoting non-duplicate methods is called *static method promotion*. For example, both `DataInputStream` and `RandomAccessFile` implement `readUTF()`. The class implementers must have realized that the two implementations were identical, so rather than repeating the code, they created a static method called `readUTF(DataInput)` and moved the common code to this static method in class `DataInputStream`. Then they provided short one line implementations of `readUTF()` in `DataInputStream` and `RandomAccessFile` that call the static method. Now that we have provided a common code repository (`InputCode`) that both `DataInputStream` and `RandomAccessFile` inherit from, we can eliminate the static method by moving its code to `InputCode` and eliminate the short methods that call this common code, since both classes now share this common instance method. This is an example where we did not actually remove repeated code. Instead, we replaced one code sharing abstraction (static sharing), that can cause maintenance problems, by a better code sharing mechanism (inheritance).

We conducted an experiment to determine how much code from the stream classes of the `java.io` libraries could be promoted, if Java supported multiple code inheritance. Table 2 and Table 3 show a summary of the method promotion and lines of code promotion respectively for each of our code promotion techniques.

Table 2 Method decrease in the Java stream classes using multiple code inheritance. The number marked with a * indicates that all lines of code (except for one line) in the method were promoted, so a single line method remained.

Class	duplicate	prefix	super-suffix	static elimination	total promoted	method decrease
DataInputStream	4 of 19	8 of 19	0 of 19	1+1*	14 of 19	74%
DataOutputStream	2 of 17	0 of 17	6* of 17	0	8 of 17	47%
RandomAccessFile	6 of 45	8 of 45	6 of 45	1	21 of 45	47%

Table 3 Executable code line decrease in the Java stream classes using multiple code inheritance. All extra lines of executable code from the extra classes, Source and Sink, are also included in the third column.

Class	initial lines	extra lines for prefix, super-suffix and static elimination	net lines after all code promotion techniques	line decrease
DataInputStream	127	2	42	67%
DataOutputStream	84	7	67	20%
RandomAccessFile	158	0	93	41%

For example, from Table 2 we see that there are 19 methods in class `DataInputStream`. Of these 19 methods, 4 were promoted since there are duplicate methods in class `RandomAccessFile`. An additional 8 methods out of 19 were promoted using the prefix technique illustrated in Figure 3.

Finally, one method was eliminated using static elimination. The instance method was promoted to `InputCode` and one static method was reduced from 40 lines to 1 line. Table 2 shows that the super-suffix technique resulted in 6 promoted methods out of 45 in class `RandomAccessFile`. The corresponding 6 methods in `DataOutputStream` (marked by an asterisk in Table 2) were not completely promoted. A shorter method was retained to make the suffix super call and to execute one or more additional lines of code.

For the line counts, we only counted executable lines and declarations, not comments or method signatures. However, more important than the size of the reductions is the lower cost of understanding and maintaining the abstracted code. Note that even though most of the method bodies of six methods move up from `DataOutputStream` to `OutputCode`, small methods remain that make super calls to these promoted common "prefix" methods. In Table 3, the third column indicates the lines that were added for an abstraction (`Sink out = this.sink();`) or a multi-super call (`super(OutputCode).writeChar(v);`). All executable lines of code in *implementations* `InputCode` and `OutputCode` are included in column 4 of Table 3.

Note that this abstraction required the creation of another new interface, `Source`, which is analogous to the interface, `Sink`, which was described earlier. The resulting inheritance hierarchy for the Stream classes is shown in Figure 5.

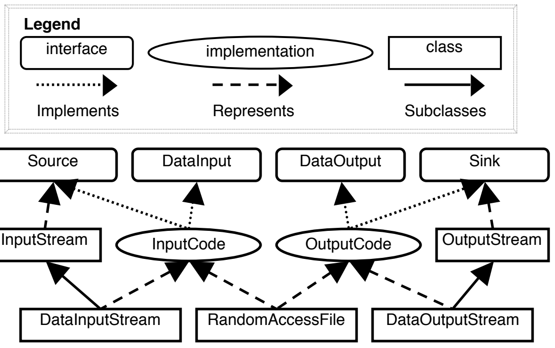

Figure 5. The revised Stream hierarchy to support multiple code inheritance.

The new interfaces `Source` and `Sink` only contain declarations of the `read()` and `write()` methods, so they contain no lines of executable code. They only exist so that they can be used as the static type of the variables `in` and `out` in the *implementations* `InputCode` and `OutputCode`.

Table 2 and Table 3 show that the use of multiple inheritance in Java can result in a significant reduction in the number of duplicate lines of code in library classes. This reduction can result in fewer errors during library maintenance and library extension and can therefore reduce maintenance costs [4].

3 Supporting *Implementations* in Java

Since Java has no concept of an *implementation*, we have three choices as to how to introduce it into Java: as a class (probably abstract), as an interface, or as a new language feature. We actually need to make this decision twice: once at the source code level and once at the JVM level. It is not necessary for the choices at these two levels to be the same.

At the source code level, an abstract class seems to be an obvious choice to represent an *implementation*. However, Section 2 clearly indicates the utility of multiple code inheritance. If *implementations* were represented by classes (abstract or concrete), we would need to modify Java to support multiple inheritance of classes. This would have the undesirable side-effect of providing multiple data inheritance, since classes (even abstract classes) are also used for data. Interfaces have the multiple inheritance we want but, if we use interfaces to represent *implementations* at the source code level, we would lose the use of interfaces for their original intent – specifications with no code.

Our solution is to introduce a new language construct, called an *implementation* at the source code level. However, at the JVM level, we decided to make use of the fact that interfaces already support multiple inheritance. Therefore, we did not introduce a new language concept at the JVM level. Instead, we generalized interfaces to allow them to contain code.

To implement our solution, we made independent localized changes to the compiler and to the Java Virtual Machine (JVM). Our compiler (*mcijavac*) compiles each *implementation* to an interface that contains code in the .class file. Our modified JVM (*mcijava*) supports execution of code in interfaces and multiple code inheritance. In addition, the JVM modifications to support multiple code inheritance are executed at load-time and the changes that affect multi-super are call-site resolution changes. Therefore, the performance of our modified JVM is indistinguishable from the original JVM. In fact, the SUN JDK 1.2.2 JVM uses an assembly-language module for fast dispatch and no changes were made to this module so the fast dispatch was preserved.

Our approach decouples language syntax changes from the JVM support required for code in interfaces and multiple code inheritance. For example, someone could propose a different language construct and syntax at the source code level and make different compiler modifications. In fact, our first implementation used a source-to-source translation approach with standard Java syntax and special comments to annotate interfaces that should be treated as *implementations* [8].

As long as a compiler or translator produces code in interfaces, our modified JVM can be used to execute the code. Similarly, someone can provide an alternate JVM that supports code in interfaces and use our language syntax and compiler to support *implementations*.

Although *implementations* support multiple code inheritance, they do not support multiple data inheritance, since they cannot contain data declarations. Multiple data inheritance causes many complications in C++. For example, if multiple inheritance is used in C++, an offset for the `this` pointer must be computed at dispatch time [11]. This is not necessary for multiple code inheritance. At first glance, it may appear that the opportunities for multiple code inheritance without multiple data inheritance are few. However, examples such as the one in Section 2 exist in the standard Java libraries and many application programs.

4 The Semantics of *Implementations*

To support *implementations*, we made two fundamental changes to the language semantics: the first to support multiple code inheritance and the second to support multi-super calls.

4.1 Semantics of Multiple Code Inheritance

The twenty-two scenarios and sub-scenarios in Figure 6 represent the common situations for inheriting code from *implementations*, including multiple code inheritance (note the Legend at the top of the Figure). The circled numbers and the letter ϕ can be ignored for now, since they are related to JVM modifications that are discussed in Section 8. Other more complex scenarios can be composed from these scenarios. When the method `alpha()` is shown in a class or *implementation*, the scenario also holds if that method is inherited from a parent type. For example, in scenario 5, the `alpha()` method in class `A` may actually be inherited from a parent class or *implementation*. The semantics are consistent, regardless of whether a method is declared explicitly in a type or inherited from a supertype.

Some scenarios have two sub-scenarios that differ only in the order of supertypes, such as scenario 7a and scenario 7b. Syntactically, this is accomplished by varying the lexical order of the *implementations*. We have defined a semantics that is symmetric with respect to order, so the results are the same for both scenarios. In some languages with multiple code inheritance, such as CLOS [3], the order is significant. In our semantics, the order is not significant. However, the order-dependent sub-scenarios are included in Figure 6 so that the interested reader can trace the JVM modifications described in Section 8 to confirm that our algorithm produces symmetric inheritance semantics.

Note that when a method `alpha()` appears in a superclass, that superclass may actually represent the class `Object`. For example, scenario 5 can be used to illustrate the situation where class `A` represents the `Object` class and the `alpha()` method represents the `toString()` method.

For the scenarios in Figure 6, consider a call-site where the static type of the receiver is any type (*implementation* or class) shown in the scenario, the dynamic type of the receiver is class `C`, and the method signature is `alpha()`.

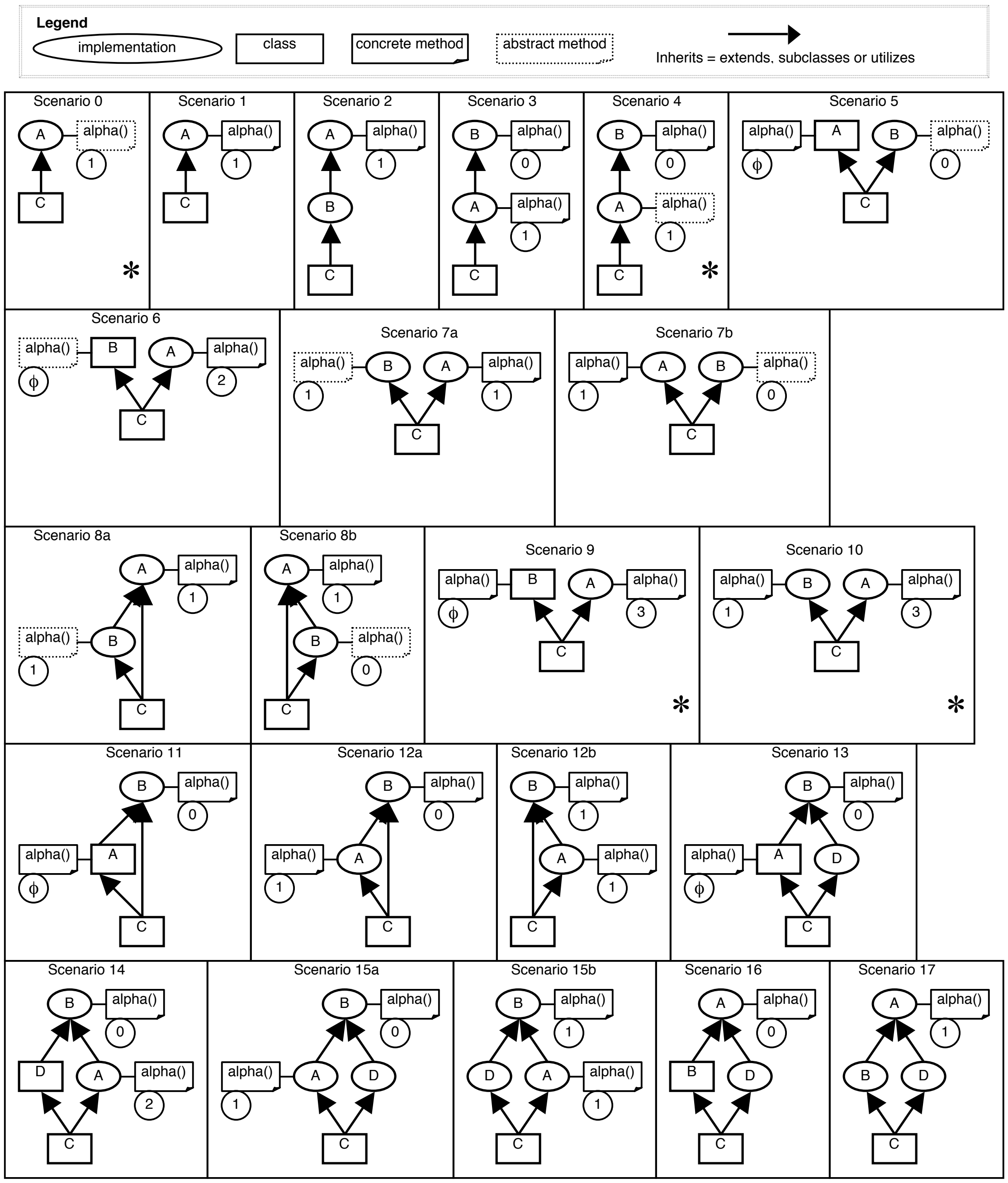

Figure 6. Inheritance scenarios for multiple code inheritance. The circles are explained in Section 8.

Scenario 0 mirrors the traditional case, where an abstract method (no code) in an interface or class is inherited in class C. Since our scenarios assume that the receiver is an instance of class C, some of the scenarios actually produce compiler errors. Such scenarios are marked with an asterisk (*) in the lower right corner. For example, scenario 0 would produce a compiler error, since it would force class C to be abstract and generate an error when the code tries to create an instance of class C. However, it is important to support such scenarios in the JVM with the correct semantics, since these scenarios can occur at runtime, if classes are recompiled in a specific order.

For example, scenario 0 can occur if *implementation* A is compiled with a non-abstract `alpha()` method, then class C is compiled and then *implementation* A is recompiled after changing `alpha()` to be abstract. If class C is not recompiled (legal in Java), then the JVM must throw an exception indicating that the code tried to execute an abstract method.

For scenarios 1 through 3, the code from *implementation* A is dispatched. The semantics mirror the single code inheritance semantics used by classes. Scenario 4 is an example of code inheritance suspension, where an abstract method in *implementation* A blocks class C from inheriting the code from *implementation* B. Scenario 4 mirrors the semantics for code inheritance from classes.

In scenarios 5 through 8, class C inherits code from a class or *implementation* along one inheritance path and an abstract method (no code) along a second path. In this case we define the code inheritance semantics for class C to inherit the code (in particular from A). Notice that scenario 5 directly mirrors the classic case where a class inherits code for a method from a superclass and implements an interface containing an abstract method with identical signature. Scenarios 8a and 8b illustrate an important principle of code inheritance suspension – an abstract method in a type can only suspend code inheritance along a path from a parent type through that type; it cannot suspend code inheritance along *all paths* from its parent type.

In each of scenarios 9 and 10, a *multiple code inheritance ambiguity* exists between two different implementations of `alpha()` in two parent types of class C. Therefore, the programmer is required to supply a local implementation of method `alpha()` in class C to clear this ambiguity. If the method in one of the parent types is desired, a method that makes a single super call to the appropriate parent can be used. Again, the order of inheritance is ignored and there is no preference for inheritance from a superclass over inheritance from a *superimplementation*.

Scenarios 11 through 15 are quite interesting cases. Two different multiple inheritance semantics could be defined for our language extension [19]. *Strong multiple code inheritance semantics* requires these scenarios to be inheritance ambiguities, since for each scenario, class C can inherit different code along different code inheritance paths. However, *relaxed multiple code inheritance semantics* states that if two types serve as potentially ambiguous code sources and are related by an inheritance relationship, the code in the child type overrides the code in the parent type. The code in the child type is called the *most specific method*. With these semantics, the inherited code in class C is the code provided by type A, for scenarios 11 through 15. We have implemented relaxed multiple code inheritance semantics in our compiler and JVM. It would be simple to implement strong semantics instead. In this case, a further decision would be required to resolve scenarios 16 and 17, since they inherit the *same code* along multiple paths. However, since we used the relaxed definition of multiple code inheritance semantics, these are not ambiguous scenarios and the code from *implementation* A is inherited by class C.

4.2 Semantics of Multi-super Calls

Even if a method is overridden, it is often desirable to invoke the original method in a supertype. However, due to multiple code inheritance, we have to choose among multiple supertypes in a super call. We saw an example of this in Figure 4. The method `writeChar(int)` in class `DataOutputStream` needed to call the overridden method code from its *superimplementation* `OutputCode`, as opposed to calling the method in its superclass, `OutputStream`. We call this generalization a *multi-super call*. We refer to a super call to a superclass as a *classic* super call to differentiate it from a multi-super call.

In C++, each multi-super call is a direct jump to a particular superclass that is specified lexically and it is resolved at compile-time (*e.g.* `A::alpha()`). If subsequent changes to the code result in a new class being inserted between the class that contains the call-site and the target class of the multi-super call, then any code in the intervening classes is ignored. This can result in a logic error if one or more of the inserted classes adds a method that performs some additional computations that are desired.

In the spirit of Java, we have defined a more dynamic semantics for multi-super than the static semantics defined for C++. For example, Java does not force the recompilation of sub-classes when a super-class is recompiled. Our modified Java compiler ensures that only a direct parent supertype can be used in the multi-super call. These multi-super semantics are consistent with Java's classic super mechanism, where the lookup always starts from the closest superclass and searches upwards for appropriate method code. If a new superclass is added, the call-site code does not need to be changed to take advantage of any "value-added" code that is inserted in new intervening classes.

Figure 7 shows the basic inheritance scenarios that define the semantics of multi-super. In each scenario, assume that a class or *implementation* (not shown in the scenario) is a direct subtype of *implementation* C and makes a multi-super call to *implementation* C. The scenarios in Figure 7 can be derived from the scenarios of Figure 6 by changing class C into *implementation* C. However, several of the scenarios have been excluded after this transformation, since it is impossible to have a class that is a supertype of an *implementation*. Therefore, scenarios 5, 6, 9, 11, 13, 14 and 16 from Figure 6 are excluded.

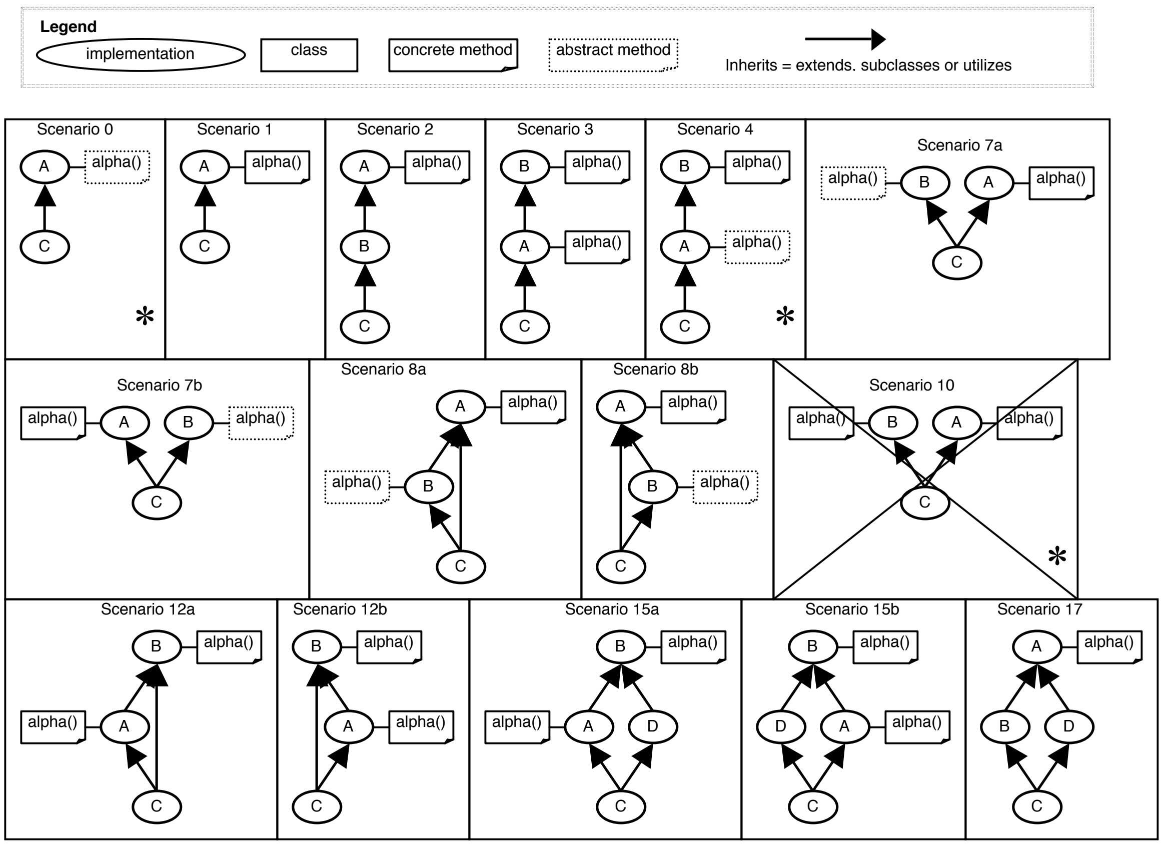

Figure 7. Inheritance scenarios for multi-super.

In addition, scenario 10 cannot occur, since an ambiguous method exception would be generated when *implementation* `C` was loaded. In other words, ambiguous super calls can never happen. In each scenario of Figure 7, the code for method `alpha()` in *implementation* `A` is executed. Of course, if this `alpha()` is abstract, then an exception is thrown. As in Figure 6, any scenario marked with an asterisk will not compile without an error, so a series of recompilations is necessary to generate the scenario at runtime.

Note that Figure 7 does not contain any scenarios where a classic super call is made to a superclass, even though the code in this superclass may actually be inherited from an *implementation*. In fact, there are many such scenarios, since each scenario in Figure 7 could have `C` as a class instead of an *implementation*. In all of these cases, the same semantics hold as if `C` is an *implementation*.

4.3 Classic Super Calls in *Implementations*

It does not make sense to have a classic super call in an *implementation*, since an *implementation* cannot have a superclass. However, if the same method appears in two classes that share a *superimplementation* and if that method contains a classic super call, promotion of this method to the *superimplementation* would result in a classic super call in this *superimplementation*. For example, assume that there is a common method in `DataInputStream` and `RandomAccessFile` of Figure 5, and that this common method contains a classic super call. The super call should invoke code in `InputStream` if the common method is invoked on an instance of `DataInputStream` and should invoke code in the `Object` if the common code is invoked on an instance of `RandomAccessFile`. However, it is illegal to put a classic super call in an *implementation*, since an *implementation* cannot have a superclass. There are three solutions to this problem in MCI-Java.

If the programmer wants to promote a method to an *implementation* that contains a classic super call, the classic super call should be replaced by a call to a new method. For example, a call `super.alpha()` should be replaced by a call, `this.superalpha()`. The new method should be implemented in each of the subclasses to contain a single line classic super call to the original method. For example, the single line in `superalpha()` would be `super.alpha()`.

A second solution is to modify MCI-Java so that including a classic super call in an *implementation* is legal and the semantics are defined as follows. At runtime, when a classic super call is made in an *implementation* (for example, `InputCode`), the JVM looks down the calling stack to the first stack frame that is a class, rather than an *implementation* (for example, `DataInputStream`) and then starts looking for code in the superclass of this class (for example, `InputStream`).

The third solution defines the same semantics as the second, but uses a different approach. Each *implementation* that contains a method with a classic super call is marked as it is loaded. When a class is loaded that inherits from such a marked *implementation*, the method is treated as though the method was local to the class, instead of being inherited from the *implementation*. As indicated in Section 9, copying a code pointer (but not the code) is equivalent to the approach taken for all methods (not just methods that contain a super call) in *traits* [21].

In MCI-Java, an *implementation* cannot have a superclass, so allowing a classic super call would be a poor choice. Therefore, we use the first solution.

5 Syntax and Compiler Changes

We made three minor syntax changes to the language to support the *implementation* language construct. First, we added the keyword `implementation` to mark an *implementation.* For example, the first line of the `OutputCode` *implementation* shown in Figure 5 is:

```
public implementation OutputCode implements
   DataOutput, Sink {
```

Second, we added the keyword `utilizes` to mark a `class` that inherits from an `implementation`. For example, the first line of the `RandomAccessFile` class shown in Figure 5 is:

```
public class RandomAccessFile utilizes
   InputCode, OutputCode {
```

and the first line of the `DataOutputStream` class shown in Figure 5 is:

```
class DataOutputStream extends
   OutputStream utilizes OutputCode {
```

Third, we modified the syntax of the `super` method call to implement the multi-super call. We specify one of many potential *implementations* that can contain code or inherit code from other *implementations*, as an argument. For example, Figure 4 shows a multi-super call from the method `writeChar(int)` in class `DataOutputStream` to the *superimplementation* of this method in *implementation* `OutputCode`.

If no code for a method is contained in a referenced *superimplementation* and it has not inherited code from one of its *superimplementations*, then the compiler generates an error, similar to the case of finding no code in a superclass for a normal super call.

Note that each class still has a unique superclass, so the syntax for a normal super call is unchanged. For example, for any method in the class `DataOuputStream`, the call `super.alpha()` would still call an implementation of `alpha()` in the superclass of `DataOutputStream`, which is `OutputStream`.

To accommodate these three syntax changes, and to report ambiguous method declarations, as described in Section 4, we modified the open-source IBM Jikes compiler, version 1.15 [13]. There were originally 243 C++ classes in the compiler. We modified 17 methods in 10 of these classes and added an additional 41 methods to them. We also added 2 classes used in building abstract syntax trees. Each of these added classes consisted of 1 constructor, 1 destructor and 3 accessor functions.

All of the changes were straightforward. However, one of the changes was especially interesting: when compiling a message expression whose receiver is the pseudo-variable, `this`, the standard compiler always generates an *invokevirtual* instruction. However, if such a message expression appears in an *implementation*, it must generate an *invokeinterface* instruction instead.

6 Correctness and Performance Experiments

This Section provides an overview of tests and experiments conducted during the process of verifying our modifications to the Jikes compiler and SUN JVM JDK 1.2.2. The first goal of our validation was to show that our multiple code inheritance implementation preserves the semantics and performance of existing single inheritance code. The second goal was to show that both our basic multiple code inheritance and the multi-super call mechanism execute correctly in multiple inheritance programs.

We first compiled and ran three large existing Java programs (javac, jasper and javap) using our modified compiler and JVM. In all three of these tests, we obtained correct results and there was no measurable change in the execution times, between the original and modified JVMs [8].

We then conducted tests to verify the correctness of our JVM and compiler modifications for multiple inheritance programs. We constructed test programs for each scenario described in Section 4 and they produced the desired results. The scenarios shown in Figure 6 test all paths through the modified class loader code shown in Figure 8. The scenarios in Figure 7 test all paths through the modified multi-super resolution code.

Finally, we conducted an experiment to evaluate the runtime performance of the refactored I/O classes described in Section 2 that used multiple inheritance, compared to the I/O classes from the standard library. These refactored library classes exercise all of the modifications that we made to support multiple code inheritance, including the use of the pseudo-variable `this` in an *implementation*. The test program ran without errors and with unmeasurable time penalties for multiple code inheritance. We used two different configurations. The first used an AMD Athlon XP 2400+ running Red Hat linux version 7.2. The second used a SUN Ultra-60 running Solaris version 9.

This test program starts by creating an instance of `RandomAccessFile` and writing a series of double precision values, int values, char values and strings to it. This exercises methods specified in the interface `DataOutput` whose code appears in the *implementation* `OutputCode`. The data file is then closed and reopened as an instance of `DataInputStream`. All of the data is read, using methods specified in the interface `DataInput`, whose code appears in the *implementation* `InputCode`, with help from a few methods that remain in `DataInputStream`. As the data is read, it is written to a second file using an instance of `DataOuputStream`. These writes exercise methods specified in `DataOutput` and implemented in `OuputCode`. Finally, this file is read using an instance of `RandomAccessFile`, exercising methods specified in `DataInput` and implemented in `InputCode`.

7 Dispatch in the Unmodified JVM

To implement multiple code inheritance, we modified SUN's JVM 1.2.2 [23] to execute code in interfaces. We know of no elegant way to implement multiple code inheritance in Java without JVM modifications. Although the approach of using inner classes [18] is interesting, its use of delegation inheritance along the interface chains is not very appealing from the language consistency perspective. Inner classes make interface inheritance second class. We have previous success in modifying JVM dispatch to support multi-method dispatch [10]. Our changes to support multiple code inheritance are concise and localized and should transfer to other JVMs.

In this Section we briefly review how a method call-site is dispatched in the unmodified SUN JVM and the standard data structures that are used. A more complete description of these data structures has appeared [8].

At compile-time, a call-site is translated to a JVM instruction whose bytecodes depend on the static type of the receiver object. If the static type is a class, then the generated opcode is *invokevirtual*. If the static type is an interface, then the opcode is *invokeinterface*. In either case, a method reference is also stored as an instruction operand – an index into the constant pool. The method reference contains the signature of the method and the static type of the receiver object.

In the SUN JVM, the dispatch of *invokevirtual* uses three data structures: *method block* (MB), *method table* (MT) and *virtual method table* (VMT). The dispatch process for *invokeinterface* requires one additional data structure, *interface method table* (IMT).

At runtime, the compiled code for each method is referenced using a *method block* (MB) that contains complete information for the method, including its signature, a pointer to its bytecodes and an offset that is used during dispatch. For interfaces, the bytecode pointer is `null`, since in standard Java there can be no code in interfaces. In the SUN JVM 1.2.2 distribution, method dispatch consists of the following three steps:

S1. Method resolution: generates a *resolution method block* .

S2. Method quicking (or pre-execution): replaces the opcode with one of its quick counterparts and computes a reference to an *execution method block*.

S3. Method execution: executes the quicked bytecode using the referenced execution method block.

A *method table* (MT) is an array of MBs that are declared (not inherited) in a class or interface. To resolve an *invokevirtual* instruction (S1), the JVM uses the bytecode's method reference to obtain the static class and a method signature. It then searches the MT of this static class for an MB whose signature matches. If no match is found, it searches the MTs along the superclass chain. The compiler guarantees that a match is found and the match is the resolution MB.

The resolved MB will not necessarily be executed. However, it will contain an offset that can be used as an index into another data structure called the *virtual method table* (VMT), which contains a pointer to the execution MB. A reference to this execution MB is used in the quick bytecodes that are generated in step 2 (S2) of the method dispatch process. We make no modifications to steps S2 or S3.

When a class is loaded, the loader constructs an MT and VMT for the class. It constructs a VMT by first copying the VMT of its superclass and then extending the VMT for any new methods that have been declared in the class, whose signatures are different from the signatures of inherited methods. During class loading, the loader may discover that the class needs a VMT slot for an abstract method for which it does not declare any code or inherit any code. In this case, the loader extends the VMT by providing a slot for this method, allocates an MB in another table called the *Miranda Method Table* (MMT) and sets the VMT slot for the method to point to this new MB. At the end of this process, every method that can be invoked on an instance of the loaded class has a unique VMT table entry that points to an MB. We refer to a VMT entry that points to an MB in the class's MT or MMT as a *local VMT entry*. It is also possible that a VMT entry points to an MB in the MT or MMT of a superclass. Such an entry is called a *non-local VMT entry*. This distinction is critical to support code in interfaces.

Resolution of *invokeinterface* (S1) is similar to resolution of *invokevirtual*, except the method reference uses an interface instead of a class. Resolution starts at the *interface method table* (IMT) of the interface.

The IMT provides an extra level of indirection that solves the problem of inconsistent indexing of interface methods between classes. This extra level of indirection is analogous to the way C++ implements multiple inheritance using multiple virtual function tables.

An IMT has one entry for each interface that is extended or implemented (directly or indirectly) by its class or interface. This entry contains a pointer to the interface and a pointer to an array of VMT offsets, where there is one array entry for each method declared in the interface.

During resolution, the JVM starts with the zero'th entry of the interface's IMT, which contains a pointer to the interface itself. The MT of this interface is searched for a matching method. If one is not found, the MTs of subsequent interfaces in the IMT are searched. The compiler guarantees a signature match.

As with the *invokevirtual* bytecode, the resolution MB may not be the execution MB. In the *invokeinterface* case, the resolution MB contains a local MT offset instead of a VMT index. To use this offset, the JVM first finds the IMT entry for the static interface type of the receiver object. This entry contains an array of VMT indexes. The offset from the resolution MB selects the array element that contains the appropriate index into the VMT of the receiver's class. This VMT entry is the execution MB and a reference to it is used in the quick bytecodes that are generated in step 2 (S2) of the dispatch process. A good description of alternate approaches to implementing *invokeinterface* , including class object search, itable search, indexed itables and the alternate IMT scheme used in the Jikes RVM (formerly Jalapeño) has appeared [2].

8 JVM Modifications for Multiple Code Inheritance

In this Section we describe the localized changes we made to the SUN JVM to support multiple code inheritance. Recall that at the source code level, we support multiple code inheritance by placing code into a new construct called an *implementation*. However, our compiler produces a .class file that represents each *implementation* by an interface with method code. Therefore, our changes to the JVM are based on supporting code in interface .class files.

8.1 Code in Interfaces

Since code from interfaces has to be reachable from the class pointer of the receiver object, we modify the IMT construction for a class to copy the code from the interfaces to a data structure accessible from the class. Since this change only affects JVM class loading code and does not change any code that is executed at a call-site, its runtime overhead is small.

In the current JVM, when constructing the IMT of the loaded class, the JVM iterates over each superinterface of the class. For each interface, the JVM iterates over each declared method. Besides the normal actions taken in the classic JVM, for each method in an interface, our modified JVM takes some additional actions. The algorithm that implements these extra actions is shown in Figure 8. Each scenario from Figure 6 is marked in the algorithm to show where it is handled. Each method in Figure 6 has a circled number that shows which action from Figure 8 is taken when its MB is processed. The circled ϕ indicates that no action is taken, since that method is in a class instead of an interface. In the algorithm, the symbol < is used to indicate a proper subtype and >= indicates a supertype.

Creating a new MB on the C-heap is necessary when a class inherits code from an interface that overrides code in a non-local MB. The alternative of changing the code pointer for a non-local MB can result in the wrong code being executed. For example, consider scenario 14 from Figure 6. At the time when class `C` is being loaded, its VMT entry for `alpha()` points to a non-local MB in the MMT of class `D`. While building the IMT in class `C`, the JVM encounters the method `alpha()` in interface `A`. If it copied the MB for `alpha()` from interface `A` to the MB for `alpha()` in class `C` (stored in the MMT of class `D`), dispatch would work properly for any `alpha()` message sent to an instance of class `C`. However, consider the message `alpha()` sent to an instance of class `D`. Its VMT entry for `alpha()` points to the modified MB in its MMT, which points to the code in interface `A` (instead of the code from interface `B`).

```
Let c be the class being loaded.
Let imb = the MB of the method being processed.
Let mb = the current MB pointed to by the VMT
   entry of c with the same signature as the
   method being processed.
if (mb.codepointer == null)
   if(mb is not local(c))
      Action 2 //scenarios: 6A
   else if (imb.type > mb.type) and there is no
      path from c.type to imb.type that does
      not go through mb.type
      Action 0 // scenarios: 4B
   else
      Action 1 // scenarios: 0, 1, 2A, 3A,4A,
              // 7aB, 7aA, 7bA, 8aB, 8aA, 8bA,
              // 10B, 12aA, 12bB, 15aA, 15bB
              // 17A
else // mb.codepointer != null
   if (imb.codepointer == null)
      Action 0 // scenarios: 5B, 7bB, 8bB
   else
      if (imb.type < mb.type)
         if (mb is local(C))
               Action 1 // scenarios:
                        // 12bA, 15bA
         else
               Action2 // scenarios: 14A
      else if (imb.type >= mb.type)
         Action0 // scenarios 3B, 11B, 12aB,
                 // 13B, 14B, 15aB, 16A,
      else // imb.type unrelated to mb.type
         Action 3 // scenarios: 9A,10A

Action 0: do nothing.
Action 1: imb is copied onto mb, but the offset
of mb (index back to the VMT) is retained.
Action 2: Create a new MB on the JVM C-heap,
copy imb to the new MB, change the current VMT
entry (the index of this entry is in the offset
of mb) to point to the new MB and change the
offset of the new MB to this VMT index.
Action 3: Throw an ambiguous method exception.
```

Figure 8. The JVM modifications to support code in interfaces.

It is important to note that resolution and dispatch of *invokevirtual* and *invokeinterface* bytecodes proceed in exactly the same way as with the unmodified JVM, but the change in the class loading code allows the code in the interface to be found and executed. With the design choices we made, no other JVM changes were required to support code in interfaces. That is, we modified the IMT construction algorithm for a class to:

1. detect and report potential ambiguities, and
2. copy the code from interfaces to classes.

8.2 Multi-super Calls

To support multi-super calls, we changed the resolution code that is executed the first time a multi-super call-site is encountered. There are no changes to the dispatch code, so any multi-super call-site that is executed more than once uses the standard JVM code for subsequent executions. For example, the JVM actually has an assembly language module that does dispatch (instead of using C code). No change to this assembly language dispatch module is required to support our multi-super extension.

A super call is compiled into an *invokespecial* bytecode, where the method reference contains a target interface instead of a class. The JVM can recognize a multi-super call, since it is the only case where the compiler generates an *invokespecial* bytecode and the method reference is an interface instead of a class.

Our modified JVM uses a custom resolution algorithm to find a resolution MB. The algorithm traverses all of the interfaces in the IMT of the target interface. It finds all MBs whose signature matches the signature in the method reference. It then computes the most-specific MB as the resolution MB. Our JVM then finishes by performing the same bytecode quicking operation as the unmodified JVM, where the call-site is replaced by an *invokenonvirtualquick* bytecode. No change is made to the way this quicked bytecode is executed (in the assembly language module).

Since resolution happens only once at each call-site, the overhead of our change is insignificant in the running time of a program, as supported by the performance experiment described in Section 6.

9 Related Work

In Section 1, we surveyed the related work in multiple inheritance on a per-language-basis. Researchers, including our group, have prototyped a variety of new approaches to multiple inheritance without the new features ever being a standard part of a language. In terms of programming language concepts, the most-closely related work to MCI-Java's *implementations* are *mixins* [1] and *traits* [21].

The main differences between the three approaches are (1) the base programming language used to prototype the feature, (2) compiler support, (3) the semantics of the inheritance, and (4) the semantics of dynamic code handling (e.g., compile-time versus load-time versus runtime). The most important differences are related to semantics, but the details of the base language and compilation are also significant. For example, since MCI-Java is implemented within a full-fledged Java VM, our work has had to solve a number of practical issues related to dynamic code, where the source code for the classes is not available to the VM. In contrast, *traits* have been prototyped within Smalltalk, which allows for all affected code to be recompiled when necessary.

The *traits* paper itself has an excellent overview of *mixins*, including a good description of how they address multiple inheritance [21]. *Mixins* have been investigated using Java as well, but *mixins* provide compositional inheritance semantics, as opposed to MCI-Java's hierarchical inheritance semantics. Consequently, *mixins* have three main problems: ordering, dispersal of glue code, and fragile hierarchies. In the interests of space, we do not repeat all of the observations made in the *traits* paper here. However, to summarize: in MCI-Java, inheritance is symmetric so ordering does not apply. There is no glue code in MCI-Java, so this is not a problem. MCI-Java solves most (but not all) of the problems with fragile hierarchies. However, MCI-Java always finds the appropriate method to use at runtime by not copying any method pointers at compile-time. In addition, all errors due to hierarchy changes that can be determined at load-time are reported at load-time and errors that cannot be determined at load-time are handled as exceptions at runtime. There are no unexpected executions with MCI-Java.

We now focus on comparing MCI-Java's *implementations* with *traits*. Our work with MCI-Java has evolved over a couple of years to include compiler support for *implementations*, more localized changes to the Java VM, and better handling of dynamic code loading. For example, an earlier version of MCI-Java, reported in Cutumisu's Master's thesis (2002) [8], used scripts to handle Java source language changes instead of a modified Java compiler. In spirit, the goals of *traits* are closely aligned with our language goals, since in addition to solving the code reuse problem, *traits* also provides a separate language component for code that is independent of language features for interface and representation.

There are many similarities between an *implementation* and a *trait*. Each is a collection of methods (code). Each can be used by a client class to augment its natively-defined (locally-defined) methods. Each can invoke methods that are abstract until they are natively defined in a client class that uses it. Each contains no representation information (instance variables). A class can use more than one *implementation* or *trait*, which can lead to inheritance conflicts. The semantics of inheritance conflicts and the resolution mechanisms are different in *traits* and MCI-Java and they will be discussed later. There are also many differences between an *implementation* and a *trait*. However, since there is not enough space to cover all of the differences, we will focus only on the fundamental distinctions.

The most fundamental difference between an *implementation* and a *trait* is that when a client class that uses a *trait* is compiled, the non-overridden methods from the *trait* are *flattened* into the client class. This means that the methods can be viewed as if they were defined natively in the client class. This does not mean that the code is copied to the client class, since the main goal is to allow the code to be shared by different client classes. Sharing is accomplished by extending the Smalltalk method dictionary (similar to a symbolic virtual method table) for each client class that uses a *trait* by one entry for each of the non-overridden methods. However, the entries in all method dictionaries that use a *trait* point to common code stored in the *trait* itself, where the *trait* is a "hidden class".

The extension of the method dictionary occurs when the client class is compiled and this is the source of many of the most important differences between *implementations* and *traits*. In Smalltalk, if a superclass or a *trait* used by a client class is recompiled, the client class is automatically recompiled. This can be done, since all of the classes and *traits* are in the same Smalltalk image. In Java, if a superclass or an *implementation* used by a client class is recompiled, the client class is not automatically recompiled. In Java, a class decides at load-time (not compile-time) which methods it will put in its method table. As a simple example, consider scenario 2 from Figure 6. Assume `A` and `B` are both *implementations* and scenario 2 applies when class `C` is compiled. However, assume that before runtime, *implementation* `B` is recompiled so that it contains a method for `alpha()`. Even though class `C` is not recompiled, the correct code in *implementation* `B` is executed, since the method table for class `C` does not contain any methods (or pointers to methods) that were copied to it when it was compiled.

This example also applies if `B` and `C` are classes. In fact, *implementations* were designed to behave as classes in this respect and mirror the semantics of Java. This is just one example of why changing method tables at compile-time (i.e., flattening) and hiding the code source at runtime is problematic in Java. *Implementations* survive at runtime as first-class language features, not as hidden entities that serve only as repositories for shared code. Again, this is the most fundamental distinction between *traits* and *implementations*.

More generally, the *traits* paper [21] has an excellent description of the three major problems with multiple inheritance: "conflicting features", "accessing overridden features" and "factoring out generic wrappers". Fortunately, MCI-Java has solved all three of the problems.

As indicated in the *traits* paper, the "conflicting features" problem is not really a problem if data is not multiply-inherited and multiple-data inheritance is disallowed in MCI-Java. One difference between *traits* and MCI-Java is the conflict resolution semantics. MCI-Java and *traits* both solve the "diamond" problem (scenario 17 of Figure 6) by declaring no conflict. However, MCI-Java adopts a more relaxed definition of inheritance conflict [19] than *traits*. Consequently, scenarios 12a, 12b, 14, 15a and 15b, which are not inheritance conflicts in MCI-Java, would be inheritance conflicts with *traits*, if the *implementations* were replaced by *traits* and no glue was used. The conflict resolution can also result in different methods being selected by MCI-Java and *traits*. If the *implementations* in scenarios 11 and 13 were replaced by *traits*, there would still be no inheritance conflicts, but *traits* would select the method in B instead of A for each case. This is because methods from *traits* take precedence over methods from superclasses, whereas in MCI-Java, inheritance from superclasses and inheritance from *superimplementations* are treated the same.

It is also correctly pointed out in the *traits* paper that explicitly naming an arbitrary superclass in the source code (as done in C++) makes the code fragile with respect to changes in the architecture of the class hierarchy. That is why MCI-Java requires every explicit multi-super call to be made to a direct *superimplementation* and inheritance is then used to find the appropriate method. For example, in scenario 17, the programmer must decide whether a super call in *implementation* (or class) C should be along the B inheritance chain or D inheritance chain, rather than explicitly specifying *implementation* A . If the inheritance hierarchy is changed above the direct superclasses, the original intent to inherit along the B or D inheritance chain will survive. Note that radical changes to the inheritance hierarchy above the immediate superclasses can be made in Java without recompiling type C and no unexpected consequences will arise. However, the only way to change the inheritance relationships between type C and its immediate *superimplementations* is to recompile type C. In this case, if the original super call is made invalid due to a direct *superimplementation* being removed, the compiler will generate an error, so no unexpected consequences will arise.

The *traits* paper also has a pertinent example of the third problem with multiple inheritance: factoring out generic wrappers. However, once again, this problem has been solved in MCI-Java. The problem can be reduced to solving the problem of what to do with classic super calls that get promoted to *implementations* as described in Section 0.

In summary, MCI-Java addresses almost all of the weaknesses attributed to *mixins* and, more generally, various aspects of multiple inheritance [21]. Although there are many similarities between MCI-Java and *traits*, the fundamental difference is the choice, for *traits*, of inheritance semantics based on flattening. The compile-time-based technique of flattening is difficult to support in Java, given the dynamic semantics of the VM and the inaccessibility of source code at load-time and runtime. The dynamic aspects of Java led to many scenarios and implementation problems (e.g., Sections 4 and 8) that we solved for MCI-Java, which are not issues for *traits* under Smalltalk.

10 Conclusions

We have shown why multiple code inheritance is desirable in Java. We have defined a mechanism, called *implementations*, which supports multiple code inheritance and a super call mechanism that allows programmers to specify an inheritance path to the desired *superimplementation*. We have defined simple syntactic Java language extensions and constructed a modified Jikes compiler (*mcijavac*) to support these extensions. We have constructed a modified JVM (*mcijava*) to support multiple code inheritance. Our modifications are small and localized. The changes consist of:

1. Class loader changes to support code in interfaces.
2. Method block resolution changes to support multi-super.

Our JVM modifications do not affect the running time of standard Java programs and they add negligible overhead to programs that use multiple inheritance.

11 Acknowledgement

This research was supported by grants from the Natural Sciences and Engineering Research Council (NSERC) of Canada. We would like to thank the referees and the programming committee for several helpful suggestions that improved the manuscript. Finally, we would like to thank the shepherd for this paper, Michal Cierniak, for his help.

References

[1] D. Ancona, G. Lagorio and E. Zucca, Jam – A Smooth Extension of Java with Mixins, *14th European Conference on Object-Oriented Programming (ECOOP)*, Cannes France, pp 145-178, June 2000.

[2] B. Alpern, A. Cocchi, S. Fink, D. Grove and D. Lieber, Efficient Implementation of Java Interfaces: InvokeInterface Considered Harmless, *16th ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications OOPSLA*, Tampa U.S.A., pp 108–124, October 2001.

[3] B. Bobrow, D. DeMichiel, R. Gabriel, S. Keene, G. Kiczales, D. Moon, Common Lisp Object System Specification, X3J13 Document 88-002R, June 1988.

[4] E.L. Burd, M. Munro, Investigating the Maintenance Implications of the Replication of Code, *Proceedings of the International Conference on Software Maintenance (ICSM)*, Bari Italy, pp 322 – 329, October 1997.

[5] C. Chambers and G. T. Leavens. BeCecil, A Core Object-Oriented Language With Block Structure and Multimethods: Semantics and Typing, *Fourth International Workshop on Foundations of Object-Oriented Languages (FOOL)*, Paris France, January 1997.

[6] C. Chambers and G. T. Leavens, Typechecking and Modules for Multimethods. *A C M Transactions on Programming Languages and Systems*, 17(6), pp 805–843, November 1995.

[7] A. Chou, J. Yang, B. Chelf, S. Hallem, and D. Engler, An Empirical Study of Operating Systems Errors. *18th ACM Symposium on Operating System Principles (SOSP)*, Banff Canada, pp 73–88, October 2001.

[8] M. Cutumisu, Multiple Code Inheritance in Java, M.Sc. Thesis, University of Alberta, http://www.cs.ualberta.ca/~systems/mci/thesis.pdf, December 2002.

[9] M. Day, R. Gruber, B. Liskov, and A. C. Myers, Subtypes vs Where Clauses: Constraining Parametric Polymorphism, *10th ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications OOPSLA*, Austin U.S.A., pp 156–168, October 1995.

[10] C. Dutchyn, P. Lu, D. Szafron, S. Bromling and W. Holst, Multi-Dispatch in the Java Virtual Machine: Design and Implementation, *Proceedings of 6th Usenix Conference on Object-Oriented Technologies and Systems (COOTS)*, San Antonio U.S.A., pp 77-92, January 2001.

[11] M. Ellis and B. Stroustrup, *The Annotated C++ Reference Manual*, Addison Wesley, New Jersey, 1990.

[12] M. Franz, The Programming Language Lagoona — A Fresh Look at Object-Orientation, *Software — Concepts and Tools*, 18, pp 14–26, 1997.

[13] IBM Research Jikes Compiler Project http://www.ibm.com/developerworks/opensource/jikes/

[14] Y. Leontiev, M. T. Özsu, and D. Szafron, On Separation between Interface, Implementation and Representation in Object DBMSs, *26th Technology of Object-Oriented Languages and Systems Conference (TOOLS USA)*, Santa Barbara U.S.A., pp 155 – 167 August 1998.

[15] Y. Leontiev, T. M. Özsu and D. Szafron, On Type Systems for Database Programming Languages. *ACM Computing Surveys*, 34(4) pp 409 – 449 December 2002.

[16] MCI-Java. http://www.cs.ualberta.ca/~systems/mci.

[17] B. Meyer, Object-Oriented Software Construction, Second Edition, Prentice-Hall, 1997.

[18] M. Mohnen, Interfaces with Skeletal Implementations in Java, *14th European Conference on Object-Oriented Programming (ECOOP 2000) - Poster Session*, June 12th - 16th 2000, Cannes, France, http://www-i2.informatik.rwth-aachen.de/~mohnen/PUBLICATIONS/ecoop00poster.html has a link to an unpublished full paper.

[19] C. Pang, W. Holst, Y. Leontiev and D. Szafron, Multi-Method Dispatch Using Multiple Row Displacement, *13th European Conference on Object-Oriented Programming (ECOOP)*, Lisbon Portugal, 304-328, June 1999.

[20] R. K. Raj, E. Tempero, H. M. Levy, A. P. Black, N. C. Hutchinson, and E. Jul, Emerald: A General-Purpose Programming Language. *Software Practice and Experience*, 21(1), pp 91–118, January 1991.

[21] N. Schärli, S. Ducasse, O. Nierstrasz and A. Black, Traits: Composable Units of Behavior, *17th European Conference on Object-Oriented Programming (ECOOP)*, Darmstadt Germany, pp 248-274, July 2003.

[22] D. Stoutamire, and S. Omohundro, The Sather 1.1 specification. Tech. Rep. TR-96-012, International Computer Science Institute, Berkeley, August 1996.

[23] Sun Microsystems Inc. Java[tm] 2 Platform. http://www.sun.com/software/communitysource/java2/download.html.