Relaxed MultiJava: Balancing Extensibility and Modular Typechecking

Todd Millstein, Mark Reay, and Craig Chambers
Department of Computer Science and Engineering
University of Washington
Box 352350
Seattle, WA 98195-2350 USA
{todd,mreay,chambers}@cs.washington.edu

ABSTRACT

We present the rationale, design, and implementation of Relaxed MultiJava (RMJ), a backward-compatible extension of Java that allows programmers to add new methods to existing classes and to write multimethods. Previous languages supporting these forms of extensibility either restrict their usage to a limited set of programming idioms that can be modularly typechecked (and modularly compiled) or simply forego modular typechecking altogether. In contrast, RMJ supports the new language features in a virtually unrestricted form while still providing mostly-modular static typechecking and fully-modular compilation. In some cases, the RMJ compiler will warn that the potential for a type error exists, but it will still complete compilation. In those cases, a custom class loader transparently performs load-time checking to verify that the potential error is never realized. RMJ's compiler and custom loader cooperate to keep load-time checking costs low. We report on qualitative and quantitative experience with our implementation of RMJ.

Categories and Subject Descriptors

D.3.3 [**Programming Languages**]: Language Constructs and Features — *classes and objects*; D.3.4 [**Programming Languages**]: Processors — *compilers*.

General Terms

Algorithms, Design, Languages

Keywords

Relaxed MultiJava, external methods, multimethods, modular typechecking, class loader

1. INTRODUCTION

The design of a programming language must balance several competing goals. One important goal is the ability to organize software into separate modules, each of which can be reasoned about (e.g. typechecked and compiled) separately from the implementations of other modules. This kind of modular checking allows software components to be developed and checked for correctness once and then reliably reused in many future contexts.

Another important goal is the ability to easily extend existing software with new capabilities, without requiring the existing software to be modified. Standard object-oriented languages gain great expressive power by allowing a class to be defined as an extension (i.e., a subclass) of an existing class, without modifying the existing class or any of its clients. More advanced object-oriented languages, including Common Lisp [46, 42], Dylan [44], Cecil [13, 14], AspectJ [26], and Hyper/J [24, 41], support several additional forms of extensibility, such as the ability to add new methods to existing classes, add statements before or after existing methods, add new superclasses to existing classes, and/or write methods that dynamically dispatch on the run-time classes of their arguments (which is a kind of extension to those argument classes).

Unfortunately, modular reasoning is in conflict with flexible extensibility. In general, the more a module can be extended from the outside, the fewer properties can be proven about the module separately from those extensions. For example, traditional statically typed object-oriented languages check that each operation is properly implemented on a class-by-class basis. This checking ensures that dispatch errors such as "message not understood" and "message ambiguous" can never occur on message sends at run time. But if new methods can be added to existing classes in an unrestricted manner, then it is easy to introduce message dispatch errors that elude modular detection [34].

Because of these conflicts, each language design represents a particular tradeoff between the amount of extensibility allowed and the amount of modular typechecking supported. Most existing languages have been biased toward one or the other extreme. For example, standard object-oriented languages support modular class-by-class typechecking but only support subclassing-based extensibility. At the opposite end of the spectrum, the advanced languages listed earlier support several additional kinds of extensibility. However, the cost of this greater extensibility has been a loss of modular static reasoning; these languages require whole-program information to perform typechecking (if they support static typechecking at all) and perhaps to perform compilation as well.

In previous work with other colleagues, we developed MultiJava [18, 17, 36], an extension to Java [4, 22] that augments Java's subclassing-based extensibility with the ability to add methods (called *external methods*) to existing classes (called *open classes* [15]) and the ability to write methods (called *multimethods*) that

OOPSLA'03, October 26-30, 2003, Anaheim, California, USA.

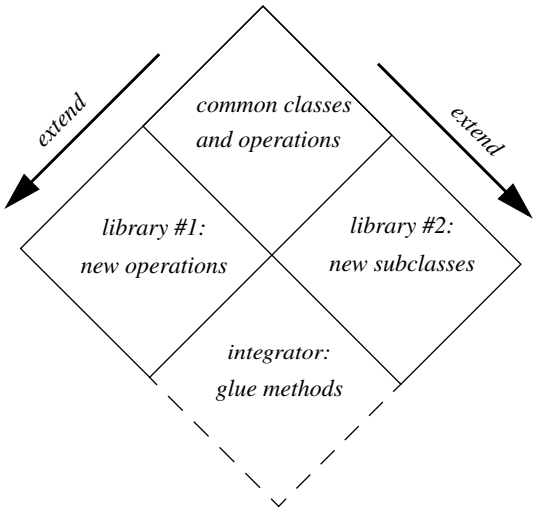

Figure 1: Completing the extensibility diamond

can dispatch on argument classes in addition to the receiver class. MultiJava supports these additional features while retaining Java's modular typechecking and compilation schemes. To do so, MultiJava restricts the ways in which the new language features may be used to a particular set of extensibility idioms that are compatible with modular checking.

As a consequence of MultiJava's insistence on fully modular typechecking, there are several useful forms of extensibility that are simply disallowed. For example, MultiJava requires all external method declarations that belong to the same operation to be written in a single file. This is because, given a strictly modular view, it would not otherwise be possible to guarantee the absence of duplicate or ambiguous external method declarations for that operation. Because there is the *potential* for an ambiguity given only partial program information, MultiJava conservatively rejects "free-standing" external method declarations. However, it is also possible that free-standing external methods are completely safe, and in practice there are programming situations that need them. For example, a client of two independently developed libraries may need to provide implementations of operations defined in one library for concrete classes defined in the other library; in other words, the client needs to "complete the diamond" set up by the two independent extensions, as illustrated in Figure 1. We sometimes refer to free-standing methods as "glue methods," since they serve to combine two separate libraries. Even if the programmer ensures that glue methods do not cause ambiguities, MultiJava will still reject this programming idiom.

In this paper we present the design and implementation of Relaxed MultiJava (RMJ). Like MultiJava, RMJ augments Java with external methods and multimethods, and it provides modular typechecking and compilation. At the same time, RMJ supports nearly arbitrary usage of the new features. These properties are achieved by giving programmers explicit control over the tradeoff between extensibility and modular reasoning, instead of having the language legislate one or the other extreme.

The key technical principle underlying RMJ's design is to treat the modular detection of the *potential* for a message dispatch error as producing merely a compile-time *warning*. For any operation flagged at modular compile time as potentially incompletely or ambiguously implemented, the programmer can choose to resolve the problem and acquire a guarantee of modular type safety. Alternatively, the programmer can retain the extra expressiveness that triggered the warning. In that case, the operation will undergo more checking at load time, to ensure that the operation is in fact properly implemented. We employ a custom class loader to perform this load-time checking. RMJ's strategy allows the expression of many more idioms than are expressible in MultiJava, but it still ensures that (a) all message dispatch errors are detected no later than load time, and (b) the programmer is always aware at modular compile time of the potential for any load-time errors. MultiJava's type system falls out as a special case of RMJ, corresponding to a scenario in which all compile-time message dispatch warnings are treated as errors by the programmer.

RMJ has the following novel collection of characteristics:

- RMJ is strictly more expressive than MultiJava, which in turn is strictly more expressive than Java. Aside from a few compilation challenges discussed later, RMJ allows arbitrary usage of external methods and multimethods.
- RMJ provides the same modular static assurances as MultiJava, because RMJ modularly and statically identifies and reports to the programmer the same problems as MultiJava. If MultiJava would report no errors to the programmer, then RMJ will report no errors to the programmer, and no errors can occur, even at load time. But where MultiJava would reject a program, RMJ might instead warn of a potential problem, allowing the programmer to take responsibility for avoiding it.
- For each compile-time warning, the RMJ class loader will check at class load time whether the potential error actually occurs for the program being linked. This check can be viewed as a natural augmentation of the normal class verification check in the standard Java class loader. If a class or external method loads successfully, then there can be no message dispatching errors involving that class or method. Such load-time checking is qualitatively better than run-time checking of each message send, even when (as in Java's case) class loading can occur at run time. Run-time checking can never prove that some future message send won't fail, whereas load-time checking guarantees that, for those classes that are loaded, there cannot be any message send, on any future execution path, that fails.
- RMJ's load-time checking typically occurs incrementally as a program runs, because of Java's lazy class loading style, with the exact set of loaded classes possibly dependent on program inputs. Load-time checking therefore guarantees the type safety of a particular set of loaded classes, but it may miss classes that can be loaded on other program runs. RMJ also includes a *preloader* tool that statically checks an application for load-time errors in all statically reachable classes. In this way, the preloader provides assurances of type safety no matter what subset of the statically reachable classes are actually loaded when the application is later run.
- As with Java and MultiJava, RMJ source code is compiled into standard Java class files modularly, one file at a time. Therefore, RMJ source and compiled files interoperate seamlessly with Java source and compiled files.
- RMJ's compiler and class loader collaborate to make the necessary load-time checking efficient, incremental, and mostly a "pay-as-you-go" proposition.

An implementation of RMJ is freely available for download and experimentation [36].

```
package ShapePackage;
import OutputPackage.*;
public abstract class Shape {
  ... generic operations on shapes ...
  public abstract void draw(OutputDevice d);
}
```

```
package RectanglePackage;
import ShapePackage.*;
import OutputPackage.*;
public class Rectangle extends Shape {
  ... implementations of Shape methods ...
  public void draw(OutputDevice d) {
    ... code for drawing a Rectangle ...
  }
}
```

```
package CirclePackage;
import ShapePackage.*;
import OutputPackage.*;
public class Circle extends Shape {
  ... implementations of Shape methods ...
  public void draw(OutputDevice d) {
    ... code for drawing a Circle ...
  }
}
```

Figure 2: `Shape` and two implementations

The next section presents the design of the RMJ language. Section 3 describes our implementation strategy, including compiler support and the structure of the RMJ class loader. Section 4 assesses our work, presenting the results of some qualitative experience using the language and quantitative performance experiments. Section 5 describes previous work on increasing the modular extensibility of traditional object-oriented languages. Section 6 concludes with a discussion of future work.

2. LANGUAGE DESIGN

This section informally describes the RMJ language, which extends MultiJava's support for expressing external methods and multimethods in Java. Both RMJ and MultiJava are explicitly designed to extend Java as little as possible, to make it easier for programmers to learn and adopt the new features. However, these syntactically small extensions offer significant new abilities to programmers. (More complete descriptions of the MultiJava language can be found elsewhere [18, 17].)

Throughout this section we will use a running example, inspired by an example due to Krishnamurthi [28]. Imagine that one author develops an abstract `Shape` class, and two independent developers each provide concrete implementations for `Rectangle` and `Circle`, as shown in Figure 2. The `draw` method relies on the abstract `OutputDevice` class in the `OutputPackage` package (not shown).

2.1 Operations and Message Dispatch

Before illustrating RMJ's features, we describe a view of Java's methods and dispatch semantics that generalizes naturally to the RMJ setting. It is useful to consider the methods in a Java program to be implicitly partitioned into a set of *operations* (sometimes referred to as *generic functions* [35, 7, 42]). Each operation is a collection of methods that have the same name and type signature. A method that does not override any other method introduces a new operation, and all methods that override that *introducing method* belong to its operation. For example, the `draw` method in `Shape` of Figure 2 introduces an operation, and the other two `draw` methods in the figure also belong to it.

Each syntactic call site *s* in a program invokes a single operation's methods. The mapping from *s* to its associated operation *o* is determined statically, based on the static types of the receiver and other arguments to the call. When a message send occurs at *s* dynamically, the *unique most-specific applicable method* belonging to *o* is chosen. A method is applicable to the message send if the class of the actual receiver is either the method's receiver class or some subclass. The unique most-specific applicable method is the single applicable method that overrides all other applicable methods.

If there were no static typechecking, two kinds of message dispatch errors would be possible dynamically. If a message send had no applicable methods, then a "message not understood" error would occur. If a message send had several applicable methods but no single most-specific one, then a "message ambiguous" error would occur. Java's static typechecking guarantees that these errors can never occur by ensuring that each operation is *properly implemented*: it has a unique most-specific applicable method for every possible type-correct concrete receiver. For example, a static error would be signaled if `Rectangle`'s `draw` method in Figure 2 were removed, as that omission could cause a run-time "message not understood" error to occur.

2.2 External Methods

RMJ and MultiJava allow new methods to be added to existing classes from the outside. For example, if a client of the `Shape` library wishes to view `Shapes` as providing an `area` operation, the client can program such an extended view of `Shapes` by writing a new file containing one or more *external method declarations* for `area`, as shown in the top of Figure 3. In this example, the client knows about the `Rectangle` and `Circle` subclasses of `Shape` and provides appropriate `area` methods for them, along with a default method that handles any other `Shape` subclasses that might exist.

We call the `area` operation *external* because its introducing method is external. In RMJ and MultiJava, external operations are scoped. To use the new `area` operation, client code must import it, just as classes are imported. For example, the second `import` declaration at the bottom of Figure 3 provides `AreaUser` access to the `area` operation. Once imported, `area` is treated just like any other operation on `Shapes`. As shown in the figure, `area` can be invoked using Java's normal message-send syntax; there is no distinction to clients between the "original" operations of `Shape` (like `draw`) and the externally added ones (like `area`). Methods of external operations can also be overridden in subclasses, like other methods. For example, in Figure 4 the `Triangle` subclass of `Shape` includes an overriding implementation of `area` as a regular method inside its class declaration.

```
package AreaPackage;

import ShapePackage.*;
import RectanglePackage.*;
import CirclePackage.*;

public double Shape.area() {
  ... default implementation ...
}

public double Rectangle.area() {
  return width() * height();
}

public double Circle.area() {
  return Math.PI * radius() * radius();
}
```

```
package AreaUserPackage;

import ShapePackage.*;

import AreaPackage.area;

public class AreaUser {

  public void usesArea(Shape s) {
    double d = s.area();
    ...
  }

}
```

Figure 3: External `area` methods and a sample client

```
package TrianglePackage;

import ShapePackage.*;
import OutputPackage.*;

import AreaPackage.area;

public class Triangle extends Shape {

  ... implementations of Shape methods ...

  public void draw(OutputDevice d) {
    ... code for drawing a Triangle ...
  }

  public double area() {
    return base() * height() / 2;
  }

}
```

Figure 4: Subclass `area` method

The ability to add methods to existing classes is a powerful and recurring idiom. The visitor design pattern [21] was developed in part to overcome the inability of existing mainstream languages to add new "visiting" operations to existing classes. The separation of code into multiple, orthogonal concerns, as in role-based programming [3, 47, 45], subject-oriented programming [24, 41], and aspect-oriented programming [27, 26], is also dependent on the ability to organize methods not by class but by concern, and then to add these methods to the underlying classes from the outside. Even when it would be possible to put all methods into their class, such as when developing an application from scratch, it may still be desirable to modularize some of the source code by operation.

A key strength of MultiJava is that each of the files in Figures 2-4 can be typechecked modularly, given access only to the *visible* classes and external operations, which are those that are referenced by a given file. For example, the `area` methods in Figure 3 are typechecked in the context of the classes in Figure 2, but without

```
package AreaPackage;

import ShapePackage.*;
import RectanglePackage.*;
import CirclePackage.*;

public abstract double Shape.area();

public double Rectangle.area() {
  return width() * height();
}

public double Circle.area() {
  return Math.PI * radius() * radius();
}
```

Figure 5: Abstract external methods in RMJ

access to `Triangle` (which may not even have been written yet). If typechecking passes on each file, then every operation in the program is guaranteed to be properly implemented, so run-time message dispatch errors will not occur. In order to make this strong guarantee, MultiJava imposes significant limitations on the kinds of external methods that can be written. RMJ provides the same modular checking as MultiJava but does not impose the associated limitations, instead transparently providing additional load-time safety checks as necessary. The following subsections describe two extensions that RMJ makes to MultiJava's external methods.

2.2.1 Abstract External Methods

It is natural to allow external methods of abstract classes to be abstract. For example, it may be desirable to declare `Shape`'s `area` method abstract; Figure 5 illustrates how this is programmed in RMJ. Abstract external methods allow the programmer to document the requirement that all concrete subclasses should provide an appropriate `area` implementation.

However, it is difficult to preserve fully modular typechecking in the face of abstract external methods. For example, suppose the `Triangle` class of Figure 4 did not import `area` nor include an overriding `area` method. If the version of the `area` operation in Figure 5 is used, we will get a "message not understood" error at run time if `area` is ever invoked on a `Triangle`. Since neither `Triangle` nor `area` is visible to the other, modular typechecking is not able to detect this error.

MultiJava addresses this problem by simply disallowing abstract external methods, thereby ensuring that each external operation has a default method implementation. Unfortunately, a reasonable default implementation of an operation does not always exist. Unless the set of operations available on `Shape` is very rich, it is unlikely that any useful `area` default implementation can be written. Therefore the default implementation's body will probably be forced to simply throw an exception. Such a default implementation satisfies MultiJava's modular typechecker, but only by creating the potential for a run-time error which is not much different than the "message not understood" error that the default implementation is written to prevent! (MultiJava's approach works well for operations where overriding methods merely provide more efficient or customized implementations of a default algorithm, such as the union of two sets, but not for operations where the overriding methods define the appropriate behavior of the operation for each subclass.)

In contrast, RMJ allows abstract external methods to be written, signaling only compile-time *warnings* rather than compile-time *errors*. When the file containing the `area` methods in Figure 5 is compiled, the programmer will be issued a warning about the *potential* for `area` to be incompletely implemented, but the file

```
package TriangleAndAreaGluePackage;

import TrianglePackage.*;
import AreaPackage.area;

public double Triangle.area() {
  return base() * height() / 2;
}
```

Figure 6: Glue external methods in RMJ

will be compiled successfully. As long as all concrete subclasses of `Shape` loaded into the program define or inherit a concrete implementation of `area`, the program will be correct and the potential for `area` to be incomplete will not have been realized. However, if a concrete subclass of `Shape` is loaded that does not override the abstract `area` method declaration, then a load-time verification error will be reported. RMJ's combination of compile-time and load-time checking is sufficient to ensure that all operations are properly implemented. Therefore, a program that passes RMJ's compile-time and load-time checks will never generate any dispatching errors when messages are sent at run time.

2.2.2 Glue Methods

Suppose again that the area and triangle libraries are two independent augmentations to the original shape hierarchy, so that the `Triangle` class wouldn't know about the `area` operation and wouldn't have an `area` method inside it. As described above, if both the `area` operation in Figure 5 and the revised `Triangle` class are loaded into the same RMJ program, a load-time error will be triggered. To resolve this problem, the integrator of the two independently developed libraries must be able to provide additional "glue" code that makes the libraries work together, i.e., that "completes the diamond" as illustrated in Figure 1. For example, Figure 6 shows a new file that defines the external method enabling the `area` operation of Figure 5 to interoperate with `Triangle`, without modifying either library (or retypechecking either library or even having source access to either library).

We refer to the `area` method in Figure 6 as a *glue method*, because it glues together an existing class with an existing operation. More precisely, a glue method is an external method that does not reside in the same file that introduces the method's associated operation. The `area` method in Figure 6 is a glue method because it belongs to the operation that was introduced in Figure 5.[1]

Unfortunately, it is difficult with a purely modular view to ensure that there are not any duplications or ambiguities between the glue method in Figure 6 and the other methods in the `area` operation. For example, although the glue method in Figure 6 is not ambiguous with the `area` methods in Figure 5, if an unseen file contains another `area` glue method for `Triangle`, at run time a "message ambiguous" error will occur when `area` is invoked on a `Triangle` instance. Because of these kinds of problems, MultiJava does not allow glue methods to be written. It instead requires all the external methods for a particular operation to be written in the file that introduces the operation, allowing an operation's external methods to be typechecked as a unit, thereby preserving modular typechecking.

1. Even if `area` had a default implementation, as in Figure 3, glue methods would still be useful, allowing clients to customize the integration of the area and triangle libraries.

In contrast, RMJ allows glue methods to be written but issues a compile-time warning that there is the potential for duplicate or ambiguous methods to appear in other files. RMJ will still compile the glue methods successfully. The class loader will then verify as glue methods are loaded that there are no duplicates or ambiguities.

An unusual issue in the design of glue methods is the need to determine how they interact with Java's lazy loading capabilities. A class is typically loaded in Java implementations upon first reference (e.g., when an instance is created). Similarly, in RMJ and MultiJava, an external operation is loaded simply by referencing it by name (e.g., in a message send to that operation). Referencing an external operation has the effect of loading the operation's introducing method, as well as all overriding methods declared in the same file. However, RMJ's glue methods are written separately from their operations, so they will never be loaded by this scheme. Furthermore, individual methods are never named directly in programs: a method is always invoked indirectly via message sends to its associated operation.

To address this problem, our custom class loader accepts a list of the names of files containing glue methods to be included in a given program. Before loading the program's first class (the one containing the `main` method), the class loader records the existence of each glue method, but it does not load any glue methods. Each glue method will be loaded as soon as it is *reachable*, meaning that the method's operation, receiver, and argument types have all been loaded. This mechanism for finding glue methods is somewhat analogous to Java's existing mechanism for finding classes, which relies on the *classpath* list provided to the class loader.

Our strategy of loading a glue method only when it is reachable ensures that the method is not loaded until it is capable of being invoked, in keeping with Java's lazy loading scheme. At the same time, the strategy still maintains a kind of monotonicity in the meaning of operations: the method chosen by invoking an operation with a given receiver and arguments cannot change during the course of a program's execution, even if new methods are added to the operation through later loading. Implementation details of our strategy for loading glue methods are provided in Section 3.

While RMJ supports glue methods belonging to external operations like `area`, it currently does not support glue methods belonging to regular "internal" operations like `draw`. Glue methods for external operations allow us to integrate separately developed class hierarchies and external operations, which was our goal. However, supporting glue methods on internal operations would enable additional kinds of useful expressiveness, particularly in the presence of multimethods (described in Section 2.3). Unfortunately, it is challenging to modularly compile glue methods belonging to internal operations in a way that is efficient and that interoperates seamlessly with existing Java source and compiled files. We leave this to future work.

2.3 Multimethods

RMJ and MultiJava also extend Java by allowing message dispatch to depend upon the run-time classes of the arguments of the message in addition to the receiver; this is called *multiple dispatching* (as opposed to the *single dispatching* of traditional receiver-based method lookup). To exploit multiple dispatching, a (possibly external) method can add a *specializer* to one or more of its arguments, which restricts the method to only apply to message sends whose arguments are instances of the specializing classes (or

```
package RectanglePackage;

import ShapePackage.*;
import OutputPackage.*;

public class Rectangle extends Shape {

  ... implementations of Shape methods ...

  public void draw(OutputDevice d) {
    ... code for drawing a Rectangle ...
  }

  public void draw(OutputDevice@BWPrinter p)
  { ... code for drawing a Rect. on a b&w printer ...
  }

}
```

```
package CirclePackage;

import ShapePackage.*;
import OutputPackage.*;

public class Circle extends Shape {

  ... implementations of Shape methods ...

  public void draw(OutputDevice d) {
    ... code for drawing a Circle ...
  }

  public void draw(OutputDevice@BWPrinter p)
  { ... code for drawing a Circle on a b&w printer ...
  }

}
```

```
package DrawUserPackage;

import ShapePackage.*;
import OutputPackage.*;

public class DrawUser {

  public void usesDraw(Shape s) {
    OutputDevice od = ...;
    s.draw(od);
  }

}
```

Figure 7: `draw` multimethods and a sample client

their subclasses); methods with argument specializers are called *multimethods*.

For example, consider the `draw` operation for `Shapes` in Figure 2. It may be useful to have special drawing functionality for particular kinds of output devices. Figure 7 shows revised `Rectangle` and `Circle` classes, each with a new multimethod for drawing on black-and-white printers. To specify a specializer, a formal argument is declared using the syntax *`StaticType@SpecializerClass FormalName`*. In the `Rectangle` class, the second `draw` method is applicable only if the dynamic class of the receiver is `Rectangle` (or a subclass) *and* the dynamic class of the argument is `BWPrinter` (or a subclass). The static argument types of a method identify to which operation the method belongs, allowing multimethods to smoothly interact with Java's static overloading mechanism.

As illustrated at the bottom of Figure 7, operations containing multimethods are invoked using Java's regular message-send syntax. At run time, the unique most-specific applicable method is invoked, as described in Section 2.1. In the presence of multimethods, a method *m* overrides a method *n* if *m*'s receiver is *n*'s receiver or a subclass, and for each argument position *i*, *m*'s *i*th specializer is *n*'s *i*th specializer or a subclass. In our example, if `draw` is sent to a `Rectangle` and a `BWPrinter`, then both `Rectangle draw` methods are applicable and the second one is chosen, since the methods have the same receiver but the second's argument specializer is more specific (an unspecialized argument is equivalent to one specialized on the static type). Sending the `draw` message to a `Rectangle` and a `ColorPrinter`, however, will invoke the first `Rectangle draw` method, since the second one is not applicable.

In addition to operations like `draw`, multimethods are natural for binary operations like equality, addition, and set union, which accept two arguments of the same type. Multimethods allow the arguments of a binary operation to be treated symmetrically and allow algorithm selection to be sensitive to the representations of both arguments. We have also found multimethods to be quite useful in event-based systems, where components register themselves to be notified when an event occurs. Notification consists of the invocation of a component's `handleEvent` operation, passing the event as an argument. To define how events are dispatched, a component defines some number of `handleEvent` multimethods, each of which specializes its event argument to the particular subclass of event to be handled.

As with external methods, MultiJava is able to modularly typecheck and compile files containing multimethods. If typechecks succeed on all files, then MultiJava guarantees that each operation is properly implemented. In the context of multimethod dispatch, this means that the operation has a unique most-specific applicable method for every possible type-correct tuple consisting of a concrete receiver and concrete arguments. As with external methods, however, MultiJava imposes restrictions on how multimethods are written to ensure this ability to check multimethods modularly. Each concrete class is required to define or inherit a *singly dispatched* implementation of each operation that it supports. For example, in the `Rectangle` class in Figure 7, the first `draw` method, which doesn't specialize on its argument, is required. If it were omitted, MultiJava would issue a compile-time error, because `draw` could be incompletely implemented if there exist output devices other than `BWPrinter`, for example `ColorPrinter`. Unfortunately, as with the earlier `area` operation, it may be difficult to write a default implementation of the `draw` operation that does not simply throw an exception.

RMJ treats the absence of singly dispatched default methods as a compile-time warning rather than a compile-time error. In our example, the default `draw` methods in `Rectangle` and `Circle` can be omitted, leaving only the `draw` multimethods, as in Figure 8. When each of the `Rectangle` and `Circle` files is compiled, as long as `draw` is implemented for all visible concrete subclasses of `OutputDevice`, the RMJ compiler will issue only a warning that there is the potential for `draw` to be incompletely implemented, but the file will be compiled successfully. If `BWPrinter` and its subclasses are the only concrete kinds of output devices loaded into the program, the program will be correct and the potential for an incomplete implementation will not be realized. However, if a different concrete subclass of `OutputDevice` is loaded, then a load-time verification error will be reported. As before, RMJ's combination of compile-time and load-time checking is sufficient to ensure that all operations are properly implemented. Therefore, a program that passes RMJ's compile-time and load-time checks will never generate any dispatching errors when messages are sent at run time.

```
package RectanglePackage;

import ShapePackage.*;
import OutputPackage.*;

public class Rectangle extends Shape {

  ... implementations of Shape methods ...

  public void draw(OutputDevice@BWPrinter p)
  { ... code for drawing a Rect. on a b&w printer ...
  }

}
```

```
package CirclePackage;

import ShapePackage.*;
import OutputPackage.*;

public class Circle extends Shape {

  ... implementations of Shape methods ...

  public void draw(OutputDevice@BWPrinter p)
  { ... code for drawing a Circle on a b&w printer ...
  }

}
```

Figure 8: `draw` multimethods in RMJ

The current RMJ implementation does not support the omission of singly dispatched default methods for internal operations like `draw`. This idiom has not been necessary for our case studies, but it could become useful when RMJ supports glue methods for internal operations. In that case, glue methods could be used to fill the "gaps" caused by the lack of singly dispatched default methods, without modifying existing code.

2.4 Dispatching on Interfaces

It is common practice for a third-party library to expose only interfaces to clients, rather than the classes implementing those interfaces. In order for clients to write multimethods that dispatch on such library classes, it is necessary to allow interfaces as specializers. However, this idiom poses a challenge for modular typechecking, because interfaces support multiple inheritance: two multimethods whose arguments specialize on different interfaces may appear to be unambiguous at modular compile time but cause a run-time "message ambiguous" error if an unseen class implements both interfaces [34].

MultiJava handles this modularity problem by simply disallowing interface specializers. For example, if the various output devices in `OutputPackage` were defined as interfaces rather than classes, then the second `draw` method in `Rectangle` and the second `draw` method in `Circle` of Figure 7 would be disallowed. In contrast, RMJ allows arbitrary usage of interface specializers, but it issues a compile-time warning about the potential for multiple-inheritance ambiguities and performs load-time ambiguity checking to ensure that the potential ambiguities never occur in practice.

Because RMJ allows interface specializers, it makes sense to also allow external methods on interfaces. Such methods are useful for the same reasons that interface specializers are useful. For example, if the shapes in `ShapePackage` were exported as interfaces, then external methods on interfaces would be required to implement the `area` functionality in Figure 3. As expected, external methods on interfaces in RMJ lead to compile-time warnings about potential ambiguities, backed up by load-time checking to ensure safety.

2.5 Discussion

In RMJ we have identified those restrictions of MultiJava that reflect only the potential for a message dispatch error and replaced them with warnings. RMJ still reports all actual and potential errors as each file is modularly compiled. This contrasts with other systems that have comparable support for external methods or multimethods, which require whole-program information in order to typecheck and/or compile a file. For example, a class in AspectJ [26] may be typechecked and compiled only when given all of the *aspects* that add external methods (among other things) to the class. RMJ's modular checking is particularly important when compiling library files whose clients are not yet known.

External methods and multimethods are the realization in the Java context of a single underlying idiom: the ability to write methods that dynamically dispatch on existing classes. External methods can dispatch on an existing receiver class, and multimethods can dispatch on existing argument specializer classes. With these two constructs, RMJ allows essentially arbitrary expression of this underlying idiom. Methods can be written and organized in any grouping desired. Those organizations that satisfy MultiJava's modular typechecking restrictions are proven safe entirely modularly. The remainder must be confirmed with a load-time check, but in many cases, the only feasible alternatives to load-time checking are methods that simply throw run-time exceptions.

RMJ's type system can be viewed as a variant of the *soft typing* approach [11], but with load-time checks instead of run-time checks. Soft typing systems attempt to perform as much static checking as possible on programs written in a dynamically typed language like Scheme. Our work takes the opposite perspective: we relax a language that supports modular static typechecking to allow more expressiveness, inserting load-time checks where needed to ensure safety. RMJ can serve as a general platform for experimenting with modular type systems for languages with external methods and multimethods. Different modular type systems can be evaluated without changing the underlying expressiveness of the language. Rather, what changes is simply the factoring of the checks between compile time and load time.

RMJ's approach is orthogonal to the particular details of external methods and multimethods. The factoring of checks between modular compile time and load time would be equally useful for other kinds of expressive languages that pose a challenge for modular typechecking. Obvious candidates are aspect-oriented and related languages, which typically forego modular guarantees in favor of increased expressiveness. Using RMJ's approach, these languages can retain their current expressiveness while still providing as much early feedback as possible to users. More early feedback can be incorporated as new research identifies opportunities for better modular reasoning in these languages.

3. COMPILATION AND CLASS LOADING

RMJ source code compiles into regular Java bytecode classes, which are loaded by a custom class loader running on a standard Java virtual machine. This section explains how bytecode for RMJ is generated, loaded, and verified. We begin by briefly reviewing how MultiJava's language extensions are compiled, then explain the additional compilation techniques used for RMJ, and finally describe RMJ's custom class loader. A key feature of our compilation strategy is that the custom class loader only performs

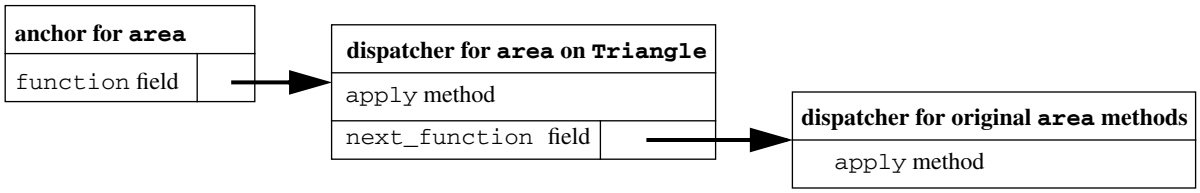

Figure 9: Structure of the implementation of `area` from Figures 3 and 4

extra checking on an operation if the compiler was forced to emit a warning message about potential incompleteness or ambiguity problems for that operation. If the compiler can verify the correctness of all classes modularly, then the custom class loader will perform no load-time checking.

3.1 Compile-Time Extensions in MultiJava

RMJ is able to reuse most of the compilation techniques of MultiJava. To implement multiple dispatching, MultiJava merges all the methods that have the same receiver class but different argument class specializers into a single bytecode method. This method consists of an `if` statement that uses a series of `instanceof` tests to dispatch to the right branch, each of which contains the body of one of the original multimethods. Clients continue to invoke operations containing multimethods in exactly the same way as before, thereby shielding clients from whether or not some operation has multimethods. This design also allows a MultiJava class to extend a regular Java class and override a regular Java method with a MultiJava multimethod, all transparently to the Java class and its existing Java clients.

To implement external operations, MultiJava generates an *anchor class* representing the operation. The external methods declared in the file that introduces the external operation are merged into a single bytecode method named `apply`, in which the original receiver has been converted into an additional argument. This `apply` method selects the right branch using a series of `instanceof` tests of the original receiver plus any other arguments with specializer classes.

As shown in Figure 4, subclasses of the receiver of an external operation can import the operation and then add additional methods to it. To allow an external operation to be extended by later classes in this way, the `apply` bytecode method for the group of methods on an external operation from a single source file is put in a *dispatcher class*; the external operation's anchor class then maintains a linked list of dispatcher class instances, in order of most specific to least specific. If a subclass adds one or more new methods to an existing external operation, the subclass methods are compiled into an `apply` method in their own dispatcher class, which is added to the front of the anchor class's dispatcher list as part of the subclass's static initialization code.

For example, consider the `area` external operation defined in Figure 3. The operation gets its own anchor class, and the three methods are merged into an `apply` method in a new dispatcher class. When `Triangle` of Figure 4 is compiled, a dispatcher class for its `area` method is created as well. When `Triangle` is initialized, its static initialization code adds this dispatcher to the front of the anchor class's dispatcher list, resulting in the structure illustrated in Figure 9. When `area` is invoked, the head of the anchor class's list of dispatchers is fetched, and its `apply` method is invoked. If none of the methods of the head dispatcher is applicable, then the `apply` method fetches the next dispatcher in the chain and invokes its `apply` method recursively. Eventually, an applicable method will be found, because modular static typechecking has verified that the operation is completely implemented.[2]

More details on the implementation techniques of MultiJava are available in earlier papers [18, 17].

3.2 Compile-Time Extensions in RMJ

The RMJ compiler has two code-generation tasks beyond what the MultiJava compiler does. First, the RMJ compiler must generate appropriate bytecode for the additional features not supported by MultiJava. Second, the RMJ compiler must record information in the resulting class files to tell the RMJ class loader what checks to perform when each class is loaded. This information is conveyed through the extensible annotation mechanism supported by Java's class file format [Lindholm & Yellin 97].

3.2.1 Compiling RMJ Extensions

RMJ allows the method introducing an external operation to be abstract, as illustrated in Figure 5. Bytecode generation for abstract methods is simple: the abstract method is treated as if it has a body that simply throws a `RuntimeException`. A similar technique can be used to generate code when a concrete class lacks a singly dispatched method for some operation, as illustrated in Figure 8. Bytecode for the implicit abstract method can be generated, with a body that throws a `RuntimeException`.

RMJ allows the receiver or an argument specializer of a method to be an interface. Bytecode generation is unaffected by this relaxation of MultiJava, since the existing strategy of testing for a method's applicability using `instanceof` tests works for interfaces as well as classes.

Finally, RMJ allows a method to be declared in a separate file from both its receiver class and its (external) operation. We compile each such glue method into an `apply` method in its own dispatcher class; this enables each glue method to be loaded separately by the RMJ class loader, as it becomes reachable. A source file containing several glue methods is itself compiled into a *glue anchor class* akin to an external operation's anchor class. The glue anchor class is used to provide annotations to the class loader about the new glue methods, as described below.

In RMJ, it is possible for glue methods to override some existing methods and to be overridden by other existing methods. For example, suppose `C` is a subclass of `B`, which is a subclass of `A`. An external operation could initially declare methods for receivers `A`

2. This "chain of responsibility" [21] style works correctly and can be generated completely modularly and statically, but it is not as efficient as regular method invocation in Java. An alternative strategy worth investigating in the future would generate a more efficient custom dispatcher method at load time, based on the current set of loaded dispatchers. One version of this strategy is used in the implementation of the Runabout [23], a variation on the visitor design pattern.

and C in one file, with a later glue method implementing the operation for B. The MultiJava implementation would merge the external methods for A and C, which were declared in the same file, into a single dispatcher class with a single apply method. However, this strategy would not allow "insertion" of methods, like that for B, into the middle of the specificity order. To resolve this problem, RMJ compiles each method of an external operation into its own dispatcher class with its own apply method. In this way, all methods of external operations are treated as if they were glue methods, for the purposes of compilation. This does have the side-effect of slowing down dispatch of external operations somewhat, as shown in Section 4.2.

3.2.2 Bytecode Annotations

The compiler must inform the class loader whenever load-time completeness or ambiguity checking is required for an operation. The compiler must additionally provide the loader with information about the methods declared on that operation, to enable the checking to be performed. Both of these tasks are accomplished via *method annotations*. The RMJ compiler produces a method annotation for each source method that belongs to an external operation. Each method annotation indicates the operation that the method is part of, the receiver and argument specializers (if any), the fully qualified names of its anchor and dispatcher classes, and whether or not the method is abstract. The annotation for a method declared in the file introducing the method's external operation (e.g., an area method in Figure 3) is placed in the operation's anchor class bytecode. The annotation for an internal method added to an external operation (e.g., the area method in Figure 4) is placed in the bytecode for the new method's receiver. Finally, the annotation for a glue method (e.g., the area method in Figure 6) is placed in the associated glue anchor class bytecode.

Method annotations provide enough information for the loader to perform the necessary checking on external operations. For example, if a method annotation for an abstract external method is observed, then the loader will know to perform completeness checking. This checking relies on the other method annotations of the operation being checked, to decide whether the operation is fully implemented. Similarly, the appearance of a method annotation for a glue method or a method that specializes on an interface indicates that the associated operation requires ambiguity checking.

Method annotations are also generated for methods of regular internal operations, in order for the loader to check their completeness and ambiguity if necessary. It would be sufficient to generate an annotation for each method in the program, but this would be a large number of annotations. Worse, it wouldn't allow existing class files compiled by a regular Java compiler to be used seamlessly (e.g., subclassed from) in RMJ programs.

Fortunately, the RMJ compiler can safely generate annotations for methods of internal operations on demand. First, if a concrete class adds multimethods to an internal operation but does not declare or inherit a singly dispatched method for that operation, the compiler generates a method annotation for the implicit abstract singly dispatched method, in addition to the annotations for each of the multimethods. These annotations are sufficient for the loader to safely and precisely check completeness.

Second, the RMJ compiler must generate annotations to allow ambiguity checking of an internal operation containing a method that specializes on an interface. When such a method is observed, method annotations are created for it and for all other methods on its operation declared in the current class. For proper ambiguity checking of the operation, annotations are also needed for all methods of the operation in any subclasses and superclasses of the current class. Therefore, the existence of the method specializing on an interface triggers the compiler to generate appropriate method annotations in each subclass when it is compiled. However, the superclasses have already been compiled, so they will in general not contain such annotations. Instead, we include the annotations for superclass methods in the bytecode for the first subclass that defines a method specializing on an interface. In this way, we generate the proper method annotations to enable load-time ambiguity checking, without either requiring existing code to be recompiled or generating method annotations for operations that do not require load-time checking.

3.3 Load-Time Extensions in RMJ

RMJ uses a custom class loader, named RMJClassLoader, that subclasses Java's standard ClassLoader class, to load the classes used in an RMJ program. This class loader observes each class loaded into the program and examines it for RMJ method annotations.

The RMJ class loader is invoked in the following manner:

```
% java -Drmj.glue=<glue> RMJClassLoader
      <Main> <args>
```

As described earlier, the class loader accepts a list of the glue files to be included in the current program; this is set via the rmj.glue property. Glue methods are processed in two phases: the first phase *registers* the existence of a glue method, and the second phase *loads* the glue method's dispatcher class and checks for duplicate or ambiguously defined glue methods. The loader performs the first phase immediately, using the method annotations in the files named in the rmj.glue property. Each glue method is not actually loaded until it becomes reachable: its operation's anchor class, receiver, and argument specializers have been loaded. This strategy ensures that each glue method is only loaded if necessary and that it gets inserted in the appropriate place in the chain of dispatchers. Details on registering and loading glue methods are provided in Section 3.3.2.

Once all the glue is registered, the loader starts the RMJ program by loading the <Main> class and invoking its main method with the given <args>. RMJClassLoader will be the defining class loader [30] for <Main>, which means that any classes referenced from that class will also be loaded with RMJClassLoader, transitively.

The key method of RMJClassLoader is loadClass, which takes the fully qualified name of a class to load, finds the bytecode implementation of the class, performs necessary RMJ checks on the class, and creates and returns the Class object representing the loaded class. (This same process applies to interfaces as well. From the perspective of the virtual machine, interfaces are simply a special kind of abstract class. We adopt this perspective throughout the rest of this section, referring to both classes and interfaces generically as classes.) The overall procedure of RMJClassLoader's loadClass method is sketched in Figure 10.

RMJClassLoader cannot be the defining loader for system classes, or else the classes will not be able to be passed to system methods. In the current implementation of RMJClassLoader, any class in the java package is loaded by the regular system class loader. Otherwise, we use the normal Java mechanisms to find the class and install it in the JVM using the inherited

```
Class loadClass(String fullName) {
  Class c;
  if (fullName.startsWith("java.")) {
    c = findSystemClass(fullName);
    registerSuperclasses(c);
  } else {
    String fileName = asFileName(fullName);
    URL url = getResource(fileName);
    byte[] bytes = ..read contents of url..;
    c = defineClass(bytes);
  }
  registerClass(c);
  loadReachableMethods(c);
  verifyCompleteness(c);
  return c;
}
```

Figure 10: `RMJClassLoader`'s `loadClass` method

`defineClass` method, with `RMJClassLoader` as the defining loader. The boldface operations in the `loadClass` method support RMJ's load-time checking and are described in the rest of this section.

Java's custom class loader mechanisms have enabled us to include additional load-time checking in the Java virtual machine. However, custom class loaders were intended to support multiple namespaces, not as a way for language designers to implement language extensions [10], and they do not gracefully support all that we and other language designers might like. For example, custom class loaders for different extensions cannot be composed nicely. We view the design of a more flexible mechanism in Java for composable load-time checkers and code transformers to be an interesting area for future work.

3.3.1 Registering Classes

In order to perform completeness and ambiguity checking incrementally as classes are loaded, the loader maintains a number of data structures, which are described as needed in this section. The `registerClass` method updates these data structures appropriately whenever a new class is loaded. Aside from registering the existence of the new class, `registerClass` also reads any method annotations in the class and updates the data structures to reflect their existence.

The RMJ class loader will not be the defining class loader for a system class. Consequently, classes referenced by the system class, such as its ancestor classes, may not be observed by the RMJ class loader. To partially account for this omission, the `registerSuperclasses` method calls `registerClass` on each of a system class's superclasses, allowing the RMJ class loader's data structures to reflect their existence. However, it is still possible for some relevant system classes to be missed, which can cause the loader to perform fewer checks than necessary to ensure correctness. An improved composable class loader mechanism would provide a way for custom class loaders to at least observe that these internal system classes have been loaded.

3.3.2 Registering and Loading Methods of External Operations

As mentioned above, the RMJ class loader examines each file listed in the `rmj.glue` property for method annotations and registers any that are found. Registration consists in the creation of an *external method descriptor* for each method, which includes the fully qualified names of the external method's anchor class, dispatcher class, receiver class, and argument specializer classes (or static types for arguments that are unspecialized). As mentioned earlier, all methods of external operations are treated as if they were glue methods for the purposes of compilation. Therefore, when `registerClass` finds an annotation in a newly loaded class for a method added to an external operation (either a method in the file introducing the operation or a method in a subclass of the operation's receiver), the method is registered exactly as glue methods are registered.

Methods of external operations will not be loaded until they are reachable. Therefore, the loader maintains an *external method registry*, which maps not-yet-loaded anchor, receiver, and argument specializer class names to the external method descriptors that are awaiting their loading. As part of a method's registration, the registry is updated to reflect the classes upon which the new method is waiting. Finally, to speed external method loading (described next), each external method descriptor also stores a count of the number of distinct not-yet-loaded classes that it is waiting on. For example, when the glue method in Figure 6 is registered via the `rmj.glue` property, it initially is waiting for the `area` operation's anchor class and the `Triangle` class. The external method registry is therefore updated to reflect these dependencies, and the method's descriptor gets a count of two.

The `registerClass` method, described earlier, is responsible for updating the external method registry to reflect the loading of a new class. That is, any mappings from the new class's name in the registry are removed, and the mapped-to external method descriptors have their counts decremented. The `loadReachableMethods` operation then loads any method that has now become reachable, i.e., whose associated descriptor's count is zero. Multiple methods of an external operation can become reachable simultaneously. In that case, `loadReachableMethods` loads the dispatcher classes of less-specific external methods and prepends them to the operation's dispatcher chain *before* those of more-specific external methods, to ensure that overriding methods always end up in front of their overridden methods on the chain.

3.3.3 Verifying Completeness

The `verifyCompleteness` method is used to ensure that operations remain complete in the face of abstract external methods and concrete classes that implicitly contain abstract singly dispatched methods. The loader must ensure that, for each such abstract method, for each type-correct tuple of concrete receiver and argument classes, the abstract method is overridden by some loaded concrete method that is applicable to the tuple. To reduce the load-time work that is performed, only tuples consisting of *top concrete classes* of the abstract method's receiver and argument types need be considered. A concrete class *C* is a top concrete class of an abstract class *D* if there is no other concrete class *E* that is a superclass of *C* and a subclass of *D*.[3] If an operation has an incompleteness, it will be revealed by a tuple of top concrete classes. By similar reasoning, only *top concrete methods* of the abstract method need to be considered for applicability to these tuples. A top concrete method is a concrete method that directly overrides the abstract method, without any intervening overriding concrete methods.

3. Recall that throughout this subsection we are treating interfaces as special kinds of abstract classes.

Completeness checking in `verifyCompleteness` uses an incremental algorithm that works as each abstract method annotation and concrete class is loaded, without any redundant checking. When a new abstract method is loaded that needs completeness checking, the loader constructs all the type-correct tuples of top concrete classes, based on the set of classes currently loaded, and checks that each has an applicable loaded method that overrides the abstract method. Conversely, when a new concrete class *C* is loaded, the loader finds all loaded abstract methods that need completeness checking and have a receiver or argument type for which *C* is a top concrete class. For each such abstract method, the loader constructs all tuples of top concrete classes that contain *C* in some position and ensures that each has an applicable loaded method that overrides the abstract method.

For example, suppose the `area` methods in Figure 5 are loaded in an RMJ program. Assuming the `Shape`, `Rectangle`, and `Circle` classes have already been loaded, `verifyCompleteness` will check for the existence of `area` methods applicable to `Rectangle` and `Circle`, as each is a top concrete class of `Shape`. The method annotations in the `area` operation's anchor class allow this checking to succeed. When the `Triangle` class is later loaded, `verifyCompleteness` will check for the existence of an `area` method applicable to `Triangle`. If there is a subclass `area` method for `Triangle`, as in Figure 4, or a glue method for `Triangle`, as in Figure 6, it will have already been loaded by `loadReachableMethods` and will therefore be properly accounted for.

Our incremental completeness algorithm resembles the Rapid Type Analysis algorithm [5]. Both algorithms maintain information about a set of reachable classes and a set of reachable operations. Whenever either set is extended, the new element is checked against all the existing elements of the other set. The algorithm is guaranteed at every point in time to have checked all pairs in the cartesian product of the two sets, without any redundant checking.

The class loader maintains several data structures to make the checking of `verifyCompleteness` efficient. They are updated incrementally by `registerClass` as each class is loaded. The data structures are as follows:

- a mapping from each loaded abstract class to its set of loaded top concrete subclasses,
- a mapping from each loaded abstract method needing completeness checking to its set of top concrete methods,
- a mapping from each loaded abstract class *C* to the set of abstract methods for which that class is the receiver or an argument type.

For maximum flexibility, our `verifyCompleteness` implementation treats a completeness error as a non-fatal warning, and still allows the program to continue execution. If the incompleteness ever actually occurs at run time, then the exception that was compiled as the body of the abstract method will be thrown. It would be straightforward to parameterize the loader to allow users to specify different ways of treating load-time errors.

3.3.4 Verifying Unambiguity

As with completeness checking, the loader performs ambiguity checking on an operation incrementally, as each class and reachable method is loaded. The heart of the loader's algorithm for incremental ambiguity checking is a routine that checks a pair of methods for ambiguity with one another. This algorithm can be used equally well to perform ambiguity checking at compile time, on the visible methods of an operation [29]. First, the receiver and argument specializers (or static argument types, where unspecialized) $(C_1,...,C_n)$ and $(D_1,...,D_n)$ for each of the two methods are retrieved. For now, we assume that the receivers and specializers are all classes; the generalization to interfaces is presented below. The algorithm checks several cases:

- If $(C_1,...,C_n) = (D_1,...,D_n)$, then the two methods are duplicates, and an ambiguity error is reported.
- Otherwise, if each C_i is equal to or a subclass of the corresponding D_i, or vice versa, then one method overrides the other, and the methods are not ambiguous.
- Otherwise, if for each i, C_i and D_i are *related*, meaning that one is equal to or a subclass of the other, then the two methods may be ambiguous, because they are applicable to overlapping sets of argument tuples. This overlap is succinctly characterized by their *intersection tuple* $(\text{int}(C_1,D_1),...,\text{int}(C_n,D_n))$, where $\text{int}(C_i,D_i)$ returns whichever of C_i or D_i subclasses from the other. The methods' overlap is not a problem as long as there exists a third method whose receiver and argument specializers form exactly the intersection tuple: the third method *resolves* the ambiguity of the first two. If such a method has been loaded, then the original two methods are unambiguous, and otherwise an ambiguity error is reported.
- Otherwise, the methods are *disjoint*: they are applicable to disjoint sets of argument tuples, and so they are unambiguous.

As a simple example, consider the `area` methods in Figure 3. All three pairs of methods pass the above check. The methods for `Rectangle` and `Circle` are disjoint from one another, because neither receiver is a subclass of the other. Further, each of these methods overrides the method for `Shape`. To illustrate intersection tuples, suppose that the `Shape` class of Figure 2 also contained a method for drawing black-and-white printers:

```
public void draw(OutputDevice@BWPrinter p)
{ ... code for drawing a Shape on a b&w printer ... }
```

The first `draw` method of `Rectangle` in Figure 7 would overlap with the above method, and the intersection tuple would be (`Rectangle`, `BWPrinter`). Without the second `draw` method in `Rectangle`, whose receiver and argument specializer form exactly the intersection tuple, the other two methods would cause an ambiguity error to occur if `draw` were invoked on the intersection tuple.

As discussed in Section 2, an operation must undergo load-time ambiguity checking if either the operation has glue methods or has methods that specialize on interfaces. We discuss each situation in turn.

3.3.4.1 Glue Methods

The loader records the set of methods that have been loaded for each operation. Then, just before `loadReachableMethods` loads the dispatcher class for a method belonging to an external operation, the new method is checked for ambiguity against each of the previously loaded methods with which it may be ambiguous, using the algorithm described above. It would be conservative for the loader to check the new method for ambiguity against each of the previously loaded methods on the same operation. However, there is no need to recheck a pair of methods for ambiguity if their unambiguity was already established at modular compile time by the RMJ compiler. Any pair of methods that were simultaneously visible by the RMJ compiler during its compile-time checks on some file need not be rechecked at load time. Therefore, the only

load-time checking that is required is between pairs of methods where one method is a glue method and the other method is either another glue method or a method written inside a class (such as the `area` method in Figure 4).

To exploit this observation, each external operation's list of previously loaded methods is partitioned into three separate lists, based on whether the method came from the source file introducing the external operation (a *base method*), the source file of a class that adds a method to the external operation (a *subclass method*), or a glue method source file (a *glue method*); the method's annotation indicates which category the method is in. Whenever a glue method is loaded by `loadReachableMethods`, it is checked for ambiguity with those methods on the glue and subclass lists. Whenever a subclass method is loaded, it is checked for ambiguity with those methods on the glue list. No other combinations need load-time checking. For example, when each of the methods in Figure 3 is loaded, no ambiguity checking is performed. When the glue method in Figure 6 is loaded, it is checked for ambiguity against any subclass methods or other glue methods, but it is not checked against the base methods from Figure 3, as that checking already occurred at compile time.

3.3.4.2 Interface Specializers

Each operation containing methods that specialize on interfaces must be checked for unambiguity at load-time. To do so, we first generalize the routine described above for checking pairwise ambiguity of methods, to properly handle multiple inheritance. We now take into account the fact that two interfaces (or one interface and one class) can have a common subclass without themselves being related. Only the second-to-last case in the earlier routine needs to be modified. First, the case should apply when for each i, either C_i and D_i are related, as before, or there exists a loaded concrete class that inherits from both C_i and D_i. In the latter case, we define int(C_i,D_i) to be the set of loaded concrete classes that inherit from both C_i and D_i. Finally, there can now be multiple intersection tuples, formed by taking the n-way cartesian product of the int(C_i,D_i) sets, each of which requires a loaded resolving method.

The revised routine depends both on the set of currently loaded methods (in order to find resolving methods) and on the set of currently loaded concrete classes (in order to compute int(C_i,D_i)). Similarly to incremental completeness checking, the loader checks unambiguity of an operation containing methods with interface specializers incrementally as each of these sets grows. If the operation is external, unambiguity of a new method with respect to previously loaded methods is checked by `loadReachableMethods`, before the new method is loaded. If the operation is internal, unambiguity of a new method is checked when the method's annotation is found by `registerClass`. Finally, the custom class loader maintains a mapping from each loaded interface to the loaded methods that have that interface as its receiver or as an argument specializer; this mapping is updated by `registerClass` as classes are loaded. When a concrete class that implements an interface is loaded and registered, each method on the interface's list is retrieved and rechecked for ambiguity with respect to the other loaded methods of its operation.

As with the ambiguity checking of operations containing glue methods, we can optimize which pairs of methods need to be checked for operations containing interface-specializing methods. Only pairs of methods where at least one has an interface specializer need to be checked for ambiguity by the RMJ loader. All other pairs are guaranteed to be unambiguous because of the RMJ compiler's modular checks. This optimization still requires that a method specializing on an interface be checked against *all* previously loaded methods, including base methods. Although each method was checked against the base methods modularly by the RMJ compiler, the compile-time checks may have missed ambiguities caused by unseen concrete subclasses of interfaces.

3.3.4.3 Run-time Ambiguity Checking

As with completeness errors, to give programmers increased flexibility, ambiguity errors are treated by our RMJ class loader as non-fatal warnings, and the program is allowed to continue. Whenever a load-time ambiguity error is reported for some operation, a special ambiguity dispatcher class, whose `apply` method throws a `RuntimeException`, is instantiated and prepended to the operation's dispatcher list. If the ambiguity is caused by duplicate methods, then the ambiguity dispatcher's `apply` method has the same receiver and argument specializers as each of the duplicates. If the ambiguity is caused by the lack of a resolving method for an intersection tuple, then the ambiguity dispatcher's `apply` method has the same receiver and argument specializers as the intersection tuple. The ambiguity dispatchers ensure that an exception will be thrown whenever a run-time ambiguity occurs. This design only works for external operations; if the load-time ambiguity error for an internal operation is ignored by the programmer and the ambiguity is encountered at run time, one of the ambiguously defined methods will be invoked arbitrarily.

3.4 RMJ Preloader

When developing an application in RMJ, the programmer may wish to exploit expressiveness that cannot be checked purely modularly, at the cost of taking on the responsibility of avoiding load-time incompletenesses and ambiguities. The RMJ class loader checks for these problems as the program is run on some input. It would also be useful to know whether or not a program can incur load-time errors at all, for *any* possible input. For example, a developer of shrink-wrapped software might wish to verify, once and for all, that no load-time errors can occur for a program. As another example, a programmer integrating two libraries may wish to find places where those libraries require glue in order to avoid load-time errors.

To assist in this kind of checking, we have developed a *preloader* tool. The preloader is invoked like the custom class loader, except that no arguments are given:

```
% java -Drmj.glue=<glue> RMJPreLoader
      <Main>
```

The preloader starts by registering all the glue listed in the `rmj.glue` property, just as `RMJClassLoader` does. The preloader then exhaustively explores all classes statically referenced by the `<Main>` class or some other referenced class, transitively, ignoring the application's actual flow of control. The preloader performs all the RMJ load-time checks as it visits each class. The preloader does not visit classes loaded only through reflective mechanisms such as `Class.forName`. Many applications only reference classes statically, and most others only rarely reference classes through reflection, so this limitation should not greatly hinder the preloader's effectiveness at finding errors. It would be straightforward to extend the preloader to accept a list of dynamically reachable classes to visit in addition to the statically reachable ones.

Even though classes may be visited by the preloader in a different order than in a real execution, and more classes may be visited by the preloader than in a real execution, the preloader is guaranteed to discover all potential load-time hazards of a real execution, with the one caveat about reflection described above. If the preloader reports a hazard, then the programmer is given early warning about a situation needing attention. If the preloader reports no hazards, then load-time errors will not occur when the program is run.

4. EXPERIENCE

We have developed an implementation of RMJ. We extended the MultiJava compiler to handle the additional RMJ language features and code generation strategy, and we implemented `RMJClassLoader` and `RMJPreloader` as described in the previous section. Our implementation is freely available for download as a part of the regular MultiJava system [36]. The next subsection describes a case study using our RMJ implementation, and the second subsection reports on some performance experiments from this study.

4.1 A Case Study

We have experimented with rewriting parts of Barat [8], a Java front-end written by others. Barat builds an abstract syntax tree (AST) from a set of Java source files, which can then be used to perform various static analyses. Barat is itself written in Java, and the AST nodes are represented by a class hierarchy, with root interface `Node`. Barat uses the visitor design pattern in order for clients to perform their desired analyses without modifying the node classes directly. To write an analysis, clients create a new class implementing the `Visitor` interface, which has a `visit`*N* method for each node named *N*. To run the analysis, clients invoke an AST node's `accept` method, passing an instance of the new visitor.

Barat comes with several predefined visitors. One of them, the `OutputVisitor`, outputs a source-code representation of the given AST nodes. We re-implemented this functionality using RMJ, by writing an external operation, `output`, on the AST node classes. As opposed to the visitor pattern, which requires "hooks" (the `accept` method) inside the node classes, the implementer of Barat did not need to plan ahead to allow us to implement our revised `output` operation. Further, it was natural to define the `output` operation to take parameters, for example the current indentation and the stream to which the output should be directed. Because the `OutputVisitor` has to conform to the `Visitor` interface, these parameters must instead be simulated via fields in the `OutputVisitor` class. Finally, clients can invoke the `output` operation via ordinary message sending syntax, as if it were defined in the original node classes.

Those benefits would be obtained via an `output` external operation in regular MultiJava, but the `output` operation also benefits from the new features of RMJ. In MultiJava, the `output` operation would be forced to contain a default implementation for the root interface `Node`, to handle any unseen concrete subclasses. However, there is no reasonable default behavior in this case, so the default method would be forced to simply throw a run-time exception. In RMJ, `output` contains an abstract method for `Node`. During modular static typechecking, a warning is signaled, and any visible subclasses are checked for completeness. Our custom class loader then checks at load time to ensure that all subclasses of `Node` do indeed have an appropriate implementation of `output`.

The `output` operation also naturally employs RMJ's glue methods. One way Barat has been used is to experiment with extensions to Java (e.g. [2]). Clients add their own subclasses of `Node` to represent the new syntax and update the Barat parser appropriately. Unfortunately, the client extensions break all existing visitors, which do not know how to visit the new nodes. If clients wish to use the `OutputVisitor`, it must first be modified in place to contain methods for visiting the new nodes, and then retypechecked and recompiled. MultiJava would allow each new node to contain an overriding `output` method as a regular internal method. However, if the `output` operation were not known when the new nodes were written, MultiJava would require modification of existing code to later add `output` methods for the new nodes.

In RMJ, the `output` operation can be updated to handle the new nodes, without requiring source access to the original `output` external methods or to the new nodes. We simply create a new file containing glue methods that provide `output` functionality for the new nodes. As an example, we created a version of the `output` operation that does not support Barat's `Cast` and `Instanceof` nodes, representing Java's run-time cast and `instanceof` test, respectively. Therefore, the `output` operation is only well-defined on the subset of Java programs that do not perform explicit run-time type manipulation. To handle the "extension" allowing casts and instanceof tests, we then created two `output` glue methods handling the new nodes, without modifying the original `output` code or the new nodes. Clients of `output` whose Java programs employ run-time type manipulation add the glue files to their `rmj.glue` property to make the two independent extensions to the `Node` hierarchy (the `output` operation and the new node subclasses) work together.

A final use of RMJ's expressiveness is required by Barat's use of interfaces as the sole external view of its functionality. Barat implements its AST nodes using two parallel hierarchies: as a set of interfaces, and as an associated set of classes implementing those interfaces. The intent is that clients never interact directly with the implementation classes, but only with the interfaces. `Node`, `Cast`, `Instanceof`, and all other `Node` kinds are public Java interfaces; internal concrete classes like `CastImpl` and `InstanceofImpl` implement these public interfaces. RMJ allows the various `output` methods to be defined directly on the public interfaces and ensures that there are no multiple-inheritance ambiguities at load time.

One benefit of the original visitor implementation is that it can be inherited for use by other visitors. For example, a `LoggingOutputVisitor` could subclass from `OutputVisitor` and override a few of the `visitN` methods to print some extra logging information in those cases, while inheriting the rest of the `visitN` methods. Writing a logging external operation in RMJ that forwards to our `output` operation would not work, since recursive calls would all go to `output` instead of back to the logging operation.

If inheritance of visitors is desired, an alternative strategy in RMJ is to implement the `OutputVisitor` as an `Output` class that contains an operation accepting the node being visited as an argument, as shown in the top of Figure 11. When the `apply` operation is invoked, multimethod dispatch is used to provide the appropriate implementation for each node. As shown in the middle of the figure, the `Output` class can then be extended by a `LoggingOutput` class, analogous to the `LoggingOutputVisitor`. The `LoggingOutput` class has an overriding `apply` multimethod for each kind of node for which

Table 1: Execution Times

version	1. Java `OutputVisitor`				2. MultiJava `output` on classes				3. RMJ `output` on classes		4. RMJ `output` on interfaces	
loader	**a. RMJ full**	**b. RMJ no-chk**	**c. RMJ no-chk no-glue**	**d. Java**	**a. RMJ full**	**b. RMJ no-chk**	**c. RMJ no-chk no-glue**	**d. Java**	**a. RMJ full**	**b. RMJ no-chk**	**a. RMJ full**	**b. RMJ no-chk**
small test	7.7	7.7	7.4	7.2	8.1	8.2	7.9	7.6	8.7	8.6	8.8	8.2
large test	19.6	19.5	19.1	18.4	20.5	20.8	20.3	18.9	20.5	20.5	21.7	20.6

```
public class Output {
  public void apply(Node@IfNode n) {
    ... code for outputing an if statement ... }
  public void apply(Node@WhileNode n) {
    ... code for outputing a while statement ... }
  ...
}
```

```
public class LoggingOutput extends Output {
  public void apply(Node@WhileNode n) {
    ... code for outputing and logging a while stmt ... }
  ...
}
```

```
public void Node.output() {
  new Output().apply(this);
}
```

Figure 11: An alternative to visitors in RMJ

logging is desired, while inheriting the rest of its functionality from `Output`. Unlike the visitor-based approach, the `Output` class using multimethods requires no advance planning from the implementer of the node hierarchy. Additionally, `apply` multimethods within the `Output` class can inherit from one another, unlike the various `visit`*N* methods of the `OutputVisitor` class. Finally, an external operation named `output` can be written as a wrapper around a call to `Output`'s `apply` method, as shown in the bottom of Figure 11, so that clients can use their normal calling sequence to invoke the operation.

4.2 Performance Experiments

RMJ adds overhead to perform its load-time checking and run-time invocation of glue methods. To gauge RMJ's performance cost, we studied four different versions of the output functionality described above. The first version is the original `OutputVisitor` class provided with Barat. The second is an external `output` operation using regular MultiJava. Because of MultiJava's restrictions for modular safety, this version includes a concrete default implementation for `Node`. In addition, all of the `output` methods are declared in a single file, and they are defined directly on the internal classes (e.g., `CastImpl` and `InstanceofImpl`) rather than the external interfaces (e.g., `Cast` and `Instanceof`). The third version is an RMJ version of the external `output` operation, which uses an abstract method for `Node` and uses glue methods for the `output` methods for casts and `instanceof` tests, but the `output` methods are still defined on the internal classes. The fourth version is the "ideal" RMJ version, which is like the third version except that the `output` methods are defined on the external interfaces. The first two versions can be run with either Java's regular class loader or with the RMJ class loader, but the third and fourth versions can only be run using RMJ's custom loader.

Table 1 presents the raw results of our performance experiments. We invoked each version of the output functionality on two inputs: a small input that's a single Java source file 662 lines in length, and a large input that's 20 Java source files 7476 total lines in length. Barat parses the file(s), creates the associated AST nodes, and then invokes the output functionality (either the `OutputVisitor` or the `output` operation) to print out the source-code representation of the nodes. All reported times are the median value in seconds of the user time of five runs, measured on a 500 MHz, Pentium III PC with 128MB RAM running RedHat Linux 7.3 and Sun Java SDK1.4.1.

Our experiments used three variants of the RMJ class loader. The "full" RMJ loader is the one described in the previous section. The "no-chk" and "no-chk no-glue" variants help identify RMJ's load-time costs. The "no-chk" variant is RMJ's class loader with all completeness and ambiguity checking disabled. The loader still loads each glue method when it becomes reachable. The "no-chk no-glue" variant is like "no-chk" but additionally does not load glue; it simply emulates the ordinary Java class loader. This variant can therefore not be used on versions 3 and 4 of the output functionality, which require glue methods.

The "passive" overhead for using the RMJ class loader instead of the regular Java class loader (the difference between columns a and d for the first two versions of the output functionality) is 7% for the small input and 7-8% for the larger input. The negligible difference between the "full" and "no-chk" loaders for the first two versions indicates that almost none of this overhead is due to the cost of maintaining data structures for any potential completeness and ambiguity checking. With the additional cost of loading glue as it becomes reachable (the difference between "full" RMJ's overhead and "no-chk no-glue" RMJ's overhead in the first two versions), the overhead is 3-4% for the small input and 1-3% for the large input. The rest of the passive overhead is simply the cost of using a class loader other than Java's default one, possibly because that

Table 2: RMJ's Run-time Cost

version	2. MultiJava `output` on classes	3. RMJ `output` on classes	4. RMJ `output` on interfaces
loader	**a. RMJ full**	**a. RMJ full**	**a. RMJ full**
small test x8	9.8	10.4	10.8
small test x16	11.4	12.1	12.6
small test x24	12.9	13.7	14.2
small test x36	15.4	16.3	16.6

loader uses native methods to load classes. Because the overhead of RMJ's class loader is incurred only when a class is loaded, the impact of the load-time costs on performance becomes less important as programs run longer.

The cost of abstract external methods and glue methods, without external methods on interfaces, is illustrated by the difference between columns 2a and 3a in Table 1. Overhead for the small input is 7%, and there is no noticeable overhead for the large input. The negligible difference between the "full" and "no-chk" columns in version 2 and in version 3 indicates that the incurred overhead is largely caused by the run-time cost of having each external method reside in its own dispatcher class.

To corroborate this observation, we ran versions 2 and 3 of the output functionality on test cases consisting of 8, 16, 24, and 36 copies of the small input file, as shown in Table 2. These test cases isolate RMJ's run-time cost, since the RMJ class loader's work is identical in each case. Our earlier observation is supported by the results: as the number of copies increases, version 3 continues to take 6% longer than version 2. Therefore, the overhead is incurred throughout execution. A more efficient compilation strategy for external operations with glue methods would reduce this overhead and is a key area for future work, as described in Section 6.

The cost of external methods on interfaces is illustrated by the difference between columns 3a and 4a in Table 1. The overhead is 1% for the small input and 6% for the large input. The overhead is largely the load-time cost of performing ambiguity checking in the presence of interfaces. This is illustrated by the fact that the difference between versions 3 and 4 disappears in the "no-chk" variant. It is also corroborated by Table 2, in which the absolute cost of version 4 over version 3 remains roughly constant (between 0.3 and 0.5 seconds) as the number of copies increases.

5. PREVIOUS WORK

The inspiration for the features in MultiJava and RMJ comes from previous languages based on multimethods, including CLOS [46, 42], Dylan [44], and Cecil [13, 14]. These languages support arbitrary multimethods and external methods (indeed, all methods are written external to their classes). However, CLOS and Dylan are dynamically typed, and Cecil requires global typechecking to ensure type safety of message sends [31]. Vortex [19], the compiler for Cecil (and other languages), employs a global compilation strategy that makes heavy use of whole-program optimization.

Parasitic methods [9] and Half & Half [6] are both extensions to Java that support *encapsulated* multimethods [12], which are akin to internal multimethods in RMJ; neither language supports external (multi)methods. Like RMJ, both languages support the use of interfaces as specializers in multimethods. Because it is difficult to modularly check multimethod ambiguity in the presence of interface specializers, parasitic methods modify the multimethod dispatch semantics so that ambiguities cannot exist, employing the textual order of methods to break ties. Half & Half resolves the problem by performing ambiguity checking on entire packages at a time, rather than on individual classes. For such package-level checking to be safe, Half & Half must also limit the visibility of some interfaces to their associated packages, thereby disallowing outside clients from employing them as specializers. In contrast, RMJ ensures that operations are unambiguous without either modifying the multimethod dispatching semantics or imposing restrictions on the usage of interface specializers. Instead, RMJ requires incremental ambiguity checking at class load time.

Recently several languages have emerged that provide direct support for separation of concerns. For example, AspectJ [26] is an aspect-oriented extension to Java, whose *aspects* can extend existing classes in powerful ways. Hyper/J [41] is a subject-oriented extension to Java that provides *hyperslices*, which are fine-grained modular units that are composed to form classes. Both languages support external methods; for example, this ability corresponds to AspectJ's *introduction* methods. The languages additionally support many more flexible extensibility mechanisms than RMJ. For example, AspectJ's *before* and *after* methods provide ways of augmenting existing methods externally. To cope with this level of expressiveness, these languages employ non-modular typechecking and compilation strategies. For example, AspectJ's compiler *weaves* the aspects into their associated classes; only when all aspects that can possibly affect a class are available for weaving can that class's typechecking and compilation be completed.

Binary Component Adaptation (BCA) [25] allows programmers to define *adaptation specifications* for their classes, which can include the addition of new methods, thereby supporting external methods. Adaptation specifications can also include modifications not supported by RMJ, like the declaration that an existing class implements a new interface. The typechecking and compilation strategy is similar to the aspect-weaving approach described above, requiring access to all adaptation specifications that can affect a given class in order to typecheck and compile the class. The authors describe an on-line implementation of BCA, whereby the weaving is performed dynamically using a specialized class loader.

Jiazzi [33] is an extension to Java that provides components with a powerful external linking semantics, including recursive linking. The authors show how to use these features to encode an *open class pattern*, whereby a component imports a class and exports a version of that class modified to contain a new method or field. Open classes in RMJ (and MultiJava) allow two clients of a class to augment the class in independent ways, without having to be aware of one another. In contrast, in Jiazzi there must be a single component that integrates all augmentations, to create the final version of the class. Component linking in Jiazzi is performed statically, so it is not possible to dynamically add new methods to

existing classes. Dynamic augmentation is possible in RMJ, since it is integrated with Java's regular dynamic loading process.

The visitor design pattern [21] is a programming idiom that allows new operations to be added to existing classes without modifying existing code. However, the visitor pattern has several drawbacks that are not shared by external methods in RMJ, as discussed in Section 4.1. Most importantly, the ability to add new operations to an existing class comes at the cost of losing the ordinary object-oriented ability to add new subclasses, since each visitor must be modified in place to handle a new subclass. Several researchers have designed extended versions of the visitor pattern that resolve this and other problems [28, 43, 32, 37, 48, 49, 23]. However, these extensions require dynamic type casts or run-time reflection, and they often further complicate the already-complex visitor pattern.

6. CONCLUSIONS AND FUTURE WORK

RMJ represents a new point in the design space balancing extensibility against modular reasoning. It offers almost the full power of external methods and multimethods while retaining all of MultiJava's modular typechecking guarantees. Many of the extensibility idioms can be proven safe purely modularly, independently of how a file's classes and operations are used in enclosing programs, and the remainder are proven safe incrementally as each file is loaded. A preloader tool assists in discovering load-time errors before run time. Programmers can explicitly choose between expressiveness and early checking, based on their software development needs and goals.

Unlike some related systems that also offer greater extensibility, RMJ retains a modular approach to typechecking and compilation. RMJ can check files separately and either guarantee them safe or point out exactly those situations that programmers must be concerned about to avoid load-time errors. Current systems based on global or large-scale translation or weaving to combine separate concerns do not provide these kinds of early checking.

Much of the challenge in developing RMJ was in designing the interplay between compile-time and load-time checking and code generation, to keep load-time overhead small. RMJ's implementation strategy performs all code generation at compile time in a modular, per-file fashion. It also attempts to perform as much checking modularly as it can. For those checks that lead to warning messages, additional annotations are generated, directing RMJ class loader's efforts to those parts of the program that need load-time checking. The loader maintains several data structures that help it to perform the needed checks efficiently.

In future work, we plan to pursue several directions:

- We wish to gain experience with the strengths and limitations of RMJ by using the language, and convincing others to use it, in the implementation of several large systems. Others have already been using MultiJava in several domains, and this experience was one motivation for designing RMJ.
- We wish to explore supporting additional extensibility while retaining modular or load-time checking. It would be useful to declare that a class implements an interface outside of the class (for example, along with adding external operations to the class). The ability to add static methods and static fields to a class from the outside would be simple but useful extensions. The ability to add instance fields to a class from the outside would also be useful but is challenging to implement efficiently. We also wish to investigate how we might incorporate some of the additional extensibility of systems like AspectJ and Hyper/J, particularly the ability to extend individual methods with additional *before* and *after* behavior from the outside, while retaining modular checking.
- We wish to investigate including binary code generation or rewriting as part of custom class loading. For one, this would allow us to dynamically generate more efficient dispatching methods for external operations. When a new dispatcher class is loaded, any previous dispatcher method would be invalidated and dropped. The next time the external operation is invoked, a customized dispatcher method based on the list of currently loaded dispatchers would be dynamically generated, loaded, and invoked. Previous efficient multimethod dispatching algorithms can be used when generating the dispatcher based on the current snapshot of loaded methods [16]. Binary code generation could also be used to allow additional kinds of extensibility that are challenging to implement modularly, including the ability to write glue methods belonging to regular internal operations.

7. ACKNOWLEDGMENTS

Thanks to Curtis Clifton, Gary Leavens, and the anonymous reviewers for very helpful feedback on this work. We are grateful to Curtis Clifton, Gary Leavens, and the rest of the MultiJava team for their work on the MultiJava implementation, which formed the basis for our RMJ implementation. This work has been supported in part by NSF grants CCR-0204047 and CCR-9970986, NSF Young Investigator Award CCR-945776, and gifts from Sun Microsystems and IBM.

8. REFERENCES

[1] *Proceedings of the 1992 European Conference on Object-Oriented Programming*, LNCS 615, Utrecht, The Netherlands, June 1992. Springer-Verlag.

[2] J. Aldrich, C. Chambers, and D. Notkin. Architectural reasoning in archjava. In *Proceedings of the 2002 European Conference on Object-Oriented Programming*, LNCS 2374, Malaga, Spain, June 2002. Springer-Verlag.

[3] E. P. Andersen and T. Reenskaug. System design by composing structures of interacting objects. In *Proceedings of the 1992 European Conference on Object-Oriented Programming* [1], pages 133–152.

[4] K. Arnold, J. Gosling, and D. Holmes. *The Java Programming Language Third Edition*. Addison-Wesley, Reading, MA, third edition, 2000.

[5] D. F. Bacon and P. F. Sweeney. Fast static analysis of c++ virtual function calls. In OOPSLA96 [39].

[6] G. Baumgartner, M. Jansche, and K. Läufer. Half & half: Multiple dispatch and retroactive abstraction for java. Technical Report OSU-CISRC-5/01-TR08, Department of Computer and Information Science, The Ohio State University, Mar. 2002.

[7] D. G. Bobrow, K. Kahn, G. Kiczales, L. Masinter, M. Stefik, and F. Zdybel. Commonloops: Merging lisp and object-oriented programming. In OOPSLA86 [38], pages 17–29.

[8] B. Bokowski and A. Spiegel. Barat — a front-end for java. Technical Report Technical Report B-98-09, Freie Universitat Berlin, FB Mathematik und Informatik, Dec. 1998.

[9] J. Boyland and G. Castagna. Parasitic methods: An implementation of multi-methods for java. In *Proceedings of the 1997 ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, Atlanta, GA, Oct. 1997.

[10] G. Bracha. Personal communication. Jan. 2003.

[11] R. Cartwright and M. Fagan. Soft typing. *SIGPLAN Notices*, 26(6):278–292, June 1991. Conference onProgramming Language Design and Implementation.

[12] G. Castagna. Covariance and contravariance: Conflict without a cause. *ACM Transactions on Programming Languages and Systems*, 17(3):431–447, May 1995.

[13] C. Chambers. Object-oriented multi-methods in Cecil. In *Proceedings of the 1992 European Conference on Object-Oriented Programming* [1], pages 33–56.

[14] C. Chambers. The Cecil language: Specification and rationale. Technical Report UW-CSE-93-03-05, Department of Computer Science and Engineering. University of Washington, Mar. 1993. Revised, March 1997.

[15] C. Chambers. Towards Diesel, a next-generation OO language after Cecil. Invited talk, the *Fifth Workshop of Foundations of Object-Oriented Langauges*, San Diego, California, Jan. 1998.

[16] C. Chambers and W. Chen. Efficient multiple and predicate dispatching. In *Proceedings of the 1999 ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 238–255, Denver, CO, Nov. 1999.

[17] C. Clifton. MultiJava: Design, implementation, and evaluation of a Java-compatible language supporting modular open classes and symmetric multiple dispatch. Technical Report 01-10, Department of Computer Science, Iowa State University, Ames, Iowa, 50011, Nov. 2001. Available from multijava.sourceforge.net.

[18] C. Clifton, G. Leavens, C. Chambers, and T. Millstein. Multijava: Modular symmetric multiple dispatch and open classes for Java. In *Proceedings of the 2000 ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, Minneapolis, MN, Oct. 2000.

[19] J. Dean, G. DeFouw, D. Grove, V. Litvinov, and C. Chambers. Vortex: An optimizing compiler for object-oriented languages. In OOPSLA96 [39].

[20] *Proceedings of the 1998 European Conference on Object-Oriented Programming*, LNCS 1445, Brussels, Belgium, July 1998. Springer-Verlag.

[21] E. Gamma, R. Helm, R. E. Johnson, and J. Vlissides. *Design Patterns*. Addison-Wesley, Reading, MA, 1995.

[22] J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java Language Specification Second Edition*. The Java Series. Addison-Wesley, Boston, Mass., 2000.

[23] C. Grothoff. Walkabout revisited: The runabout. In *Proceedings of the 2003 European Conference on Object-Oriented Programming*, LNCS 2743, Darmstadt, Germany, July 2003. Springer-Verlag.

[24] W. Harrison and H. Ossher. Subject-oriented programming (A critique of pure objects). In *Proceedings of the 1993 ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 411–428, Washington, D.C., Oct. 1993.

[25] R. Keller and U. Hölzle. Binary component adaptation. In E. Jul, editor, *ECOOP '98—Object-Oriented Programming*, volume 1445 of *Lecture Notes in Computer Science*, pages 307–329. Springer, 1998.

[26] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An overview of aspectj. In *Proceedings of the 2001 European Conference on Object-Oriented Programming*, LNCS 2072, Budapest, Hungary, June 2001. Springer-Verlag.

[27] G. Kiczales, J. Lamping, A. Menhdhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In *Proceedings of the 1997 European Conference on Object-Oriented Programming*, LNCS 1241, Jyvaskyla, Finland, June 1997. Springer-Verlag.

[28] S. Krishnamurthi, M. Felleisen, and D. P. Friedman. Synthesizing object-oriented and functional design to promote re-use. In ECOOP98 [20], pages 91–113.

[29] G. T. Leavens and T. D. Millstein. Multiple dispatch as dispatch on tuples. In OOPSLA98 [40], pages 374–387.

[30] S. Liang and G. Bracha. Dynamic class loading in the Java virtual machine. In OOPSLA98 [40], pages 36–44.

[31] V. Litvinov. Constraint-based polymorphism in cecil: Towards a practical and static type system. In OOPSLA98 [40].

[32] R. Martin. Acyclic visitor. In *Pattern Languages of Program Design 3*, pages 93–104. Addison-Wesley, 1998.

[33] S. McDirmid, M. Flatt, and W. C. Hsieh. Jiazzi: New-age components for old-fashioned java. In *Proceedings of the 2001 ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, Tampa, FL, Oct. 2001.

[34] T. Millstein and C. Chambers. Modular statically typed multimethods. In *Proceedings of the 1999 European Conference on Object-Oriented Programming*, LNCS 1628, pages 279–303, Lisbon, Portugal, June 1999. Springer-Verlag.

[35] D. A. Moon. Object-oriented programming with flavors. In OOPSLA86 [38], pages 1–8.

[36] MultiJava home page. http://multijava.sourceforge.net.

[37] M. E. Nordberg. Default and extrinsic visitor. In *Pattern Languages of Program Design 3*, pages 105–123. Addison-Wesley, 1998.

[38] *Proceedings of the 1986 ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, Portland, OR, Nov. 1986.

[39] *Proceedings of the 1996 ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, San Jose, CA, Oct. 1996.

[40] *Proceedings of the 1998 ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, Vancouver, Canada, Oct. 1998.

[41] H. Ossher and P. Tarr. Hyper/J: multi-dimensional separation of concerns for Java. In *Proceedings of the 22nd International Conference on Software Engineering*, pages 734–737, June 2000.

[42] A. Paepcke. *Object-Oriented Programming: The CLOS Perspective*. MIT Press, 1993.

[43] J. Palsberg and C. B. Jay. The essence of the visitor pattern. In *Proc. 22nd IEEE Int. Computer Software and Applications Conf., COMPSAC*, pages 9–15, 19–21 Aug. 1998.

[44] A. Shalit. *The Dylan Reference Manual*. Addison-Wesley, Reading, MA, 1996.

[45] Y. Smaragdakis and D. Batory. Implementing layered designs with mixin layers. In ECOOP98 [20], pages 550–570.

[46] G. L. Steele Jr. *Common Lisp: The Language*. Digital Press, Bedford, MA, second edition, 1990.

[47] M. VanHilst and D. Notkin. Using C++ templates to implement role-based designs. In *Proceedings of 2nd International Symposium on Object Technologies for Advanced Software*, Mar. 1996.

[48] J. Vlissides. Visitor in frameworks. *C++ Report*, 11(10):40–46, November/December 1999.

[49] M. Zenger and M. Odersky. Extensible algebraic datatypes with defaults. In *Proceedings of the 2001 ACM SIGPLAN International Conference on Functional Programming*. ACM, September 3-5 2001.