Safe Upgrading without Restarting

Miles Barr and Susan Eisenbach
Department of Computing
Imperial College
London, Great Britain, SW7 2BZ
email: `sue@doc.ic.ac.uk, miles@milesbarr.com`

Abstract

The distributed development and maintenance paradigm for component delivery is fraught with problems. One wants a relationship between developers and clients that is autonomous and anonymous. Yet components written in languages such as C++ require the recompilation of all dependent subsystems when a new version of a component is released. The design of Java's binary format has side-stepped this constraint, removing the need for total recompilation with each change. But the potential is not fulfilled if programs have to be stopped to swap in each new component.

This paper describes a framework that allows Java programs to be dynamically upgraded. Its key purpose is to allow libraries that are safe to replace existing libraries without adversely affecting running programs. The framework provides developers with a mechanism to release their libraries and provides clients with the surety of only upgrading when it is safe to do so.

1 Introduction

The advent of the Internet has added its own problems for software maintenance.

> These problems cannot be prevented or solved directly because they arise out of inconsistencies between dynamically loaded packages that are not under the control of a single system administrator and so cannot be addressed by current configuration management techniques [23].

Software has moved away from the stand alone applications of pre-Internet days. Client/server applications are the norm and the stand alone applications that exist today are expected to undergo regular upgrades. Software development itself is done in distributed environments.

With the universal availability of the Internet we can now alter end-user software, ensuring that each client is using the latest version. Developers can work across the globe on the same project, or on different projects sharing the same resources. Unfortunately, the upgrading process may not go smoothly. Upgrades can be incompatible, or may introduce subtle errors that don't appear until later in the program's life. There is scope for a more rigorous upgrading process.

Even if the upgrade itself is fine, the upgrade process itself frequently causes problems. Usually when a component is replaced the program (typically a server) needs to be stopped and restarted. The maintainer downloads the new version, tests it, halts the server and installs it. To fully automate the process, especially if continuous execution is required, dynamic upgrading is necessary.

Problems arises when users are developers and the new component is a library. The Java Language Specification [11] defines changes as *binary compatible* if they can be made to programs without introducing link time errors. Making sure the component is binary compatible is only one part of the problem. Developers could be located any where in the world, almost certainly out of the control of the component author. We need a framework that will allow developers to *automagically* upgrade to the latest version. If the new component is binary compatible, it replaces the old one and the developer needn't do any work. We provide a framework to allow the safe dynamic upgrading of software across the Internet.

The problems of distributed software maintenance are encountered frequently. For example, a toolkit supplier may release software when it is barely out of alpha testing due to time constraints. So users of the libraries will get frequent upgrades, often resulting in systems breaking. Inevitably an upgrade to an incompatible version will be needed, but considerable time will be lost simply finding out if a new version will cause problems. With our framework upgrades to incompatible versions could be scheduled so that when the site goes live it could automatically switch over to newer versions of the toolkit to enhance performance.

If the demands of distributed software maintenance were not enough, designers of new software regularly test the limits of their technology. In an email correspondence with a software engineer from a major communications provider [21] a new project he is working on, is described in the following terms:

> The customers will deploy our application by scattering our classes on multiple HTTP servers. Hence, the end-user's application will gather the classes from multiple HTTP servers. Each having a potentially different release of the classes, and each having a potentially different subset of the classes (even per package). The client will assemble the application by itself from this multi-server download. For each class name, it will know which of the replicates is the most recent version. With this information it will build on-the-fly the most recent version of the application it possibly can.

The rest of this paper describes our aid to making this kind of software possible. In the next section we look at more detailed requirements for our framework, the *Distributed Java Version Control System* DJVCS. This is followed by a detailed examination of binary compatibility and the tool support required to deal with checking for binary compatibility. Next each of the other components that was built to meet the requirements is described in turn and the performance of the system as a whole. We then look at related work and conclude with an evaluation and future directions this work could take.

2 Requirements Specification

We have chosen to implement our ideas for Java programs for several reasons. Java was designed after the Internet with dynamic loading built in explicitly.

Older languages such as C++ suffer from the *fragile base class problem* [9]. When C++ compiles its object file it stores fields and virtual members as offsets. This causes problems when superclasses are altered since everything needs to be recompiled. Since sources aren't always available in a distributed environment this is a very serious problem with C++. Java gets around this by having typing information available at linking time. It also provides programmers with lists of changes that can be made that do not require recompilation of other code.

On the other hand C# [16, 26] also has dynamic linking explicitly built into the system (in this case .NET [18]) but hasn't yet hit the level of stability and maturity that the older Java language has. So building tools using it is likely to be harder. Extensions such as the Java Management Extensions [24], provide a standard framework for managing Java components don't yet exist for C#.

In order that our framework be genuinely useful there is nothing proprietary about it. It is written in pure Java (version 1.3) and does not require any modifications to the JVM. The binary class files it works on ,are produced by a standard Java compiler (javac or jikes) and are not altered by the framework.

These constraints mean that the system can be used wherever Java can be used and the resulting library is not special in any way. The system does not force any user to have a customized environment where existing Java tools and libraries might not function correctly.

The framework is designed to make software development with maintenance in mind, in distributed environments easier. To exist in a distributed environment it needed to adopt the client/server approach. The library developer runs the server and his clients are the applications written by application developers. The server manages the testing of binary compatibility and distribution of components. It also acts as a repository to the collection of binary compatibility rules. Finally, it needs to mediate the communication between the server and client applications.

To do this clients need to have a component that they can build on that handles the library. On the library author's end there has to be some sort of server. The server handles the requests from clients for upgrades. It takes in the request and returns the appropriate version. There may be cases where the clients want a newer version that is not binary compatible since it contains a new feature. The server needs to be able to cope with this as well.

To make use of these custom functions we needed our own 'client' applications to talk to the server, a *rule manager* and a textitlibrary manager. DJVCS needs to be able to add and remove rules from the server. It can be also be used to find out what rules are currently on the server.

The managing of the libraries is more complex. Besides being able to add and remove libraries, this tool also needed to cope with the different versions of the libraries, running the binary compatibility checks and retrieving the results from the tests.

While the technology to implement this is at the limits of Java's capabilities, the actual architecture is straightforward, as shown in figure 1.

3 Binary Compatibility

Java was the first language to define explicitly *binary compatibility* in its specification [11]. If two components are binary compatible and the first one links with other code then the second one will also link. This makes it possible to swap in the new compiled class file and the program will continue to run.

There is a list of thirty binary compatible changes (*r*ules) in the Java Language Specification (JLS) [11]. and a *safe*

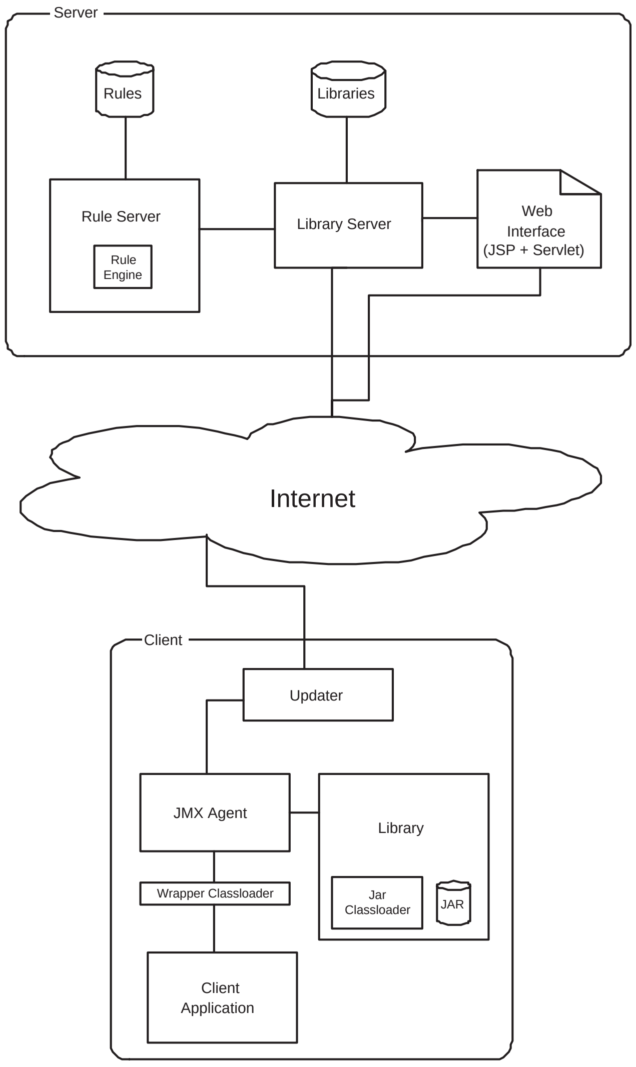


Figure 1. Overall Architecture

subset of these rules has been formalized in [19]. These include adding a new class or interface as long as the name of the new type does not introduce a namespace clash, altering a method body, changing the direct super-class of a class, as long as all direct or indirect super-classes continue to be direct or indirect super-classes and adding new methods and fields to classes.

Just because a change is listed as binary compatible, does not mean that the change will be trouble free. There are several changes listed as binary compatible that lead to unsatisfactory situations. They are listed in the appendix. We have chosen to flag these as dangerous. For example, from the formalisation [19] it can be seen that '13.5.3 The Interface Members' from the JLS [11] is not safe since all classes that implement the interface are expected to have the new method. This is contrary to the JLS [11] which states:

> Adding a method to an interface does not break compatibility with *pre-existing* binaries.

The exact wording of the rule gives a let out clause but it is still a dangerous change since new code based upon existing components can appear to 'randomly' break when they worked before.

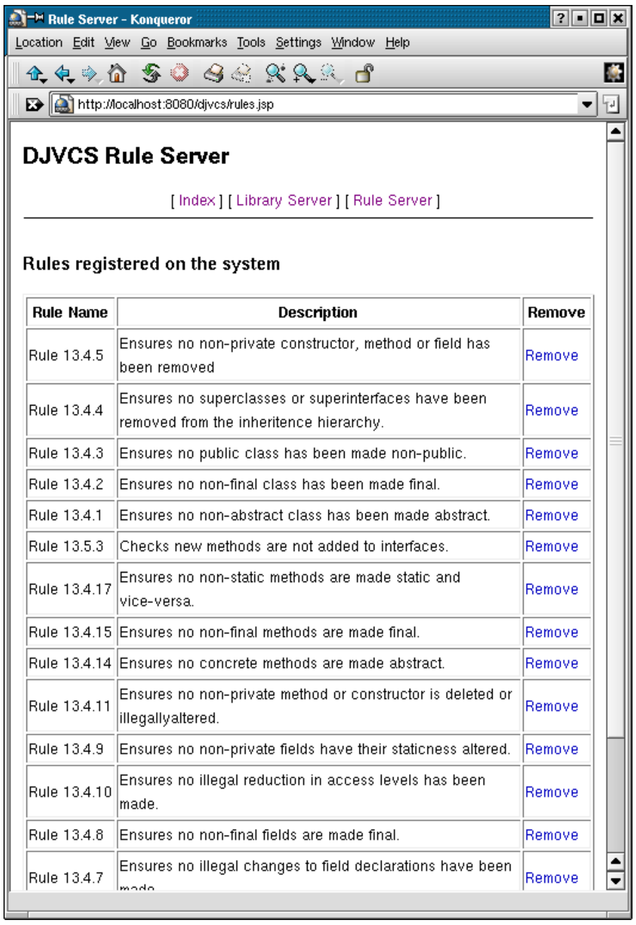
Other problems arise out of overloading. According to Rule 13.4.21 in the Java Language Specification [11], overloading of methods and constructors is always allowed. But overloading a method may introduce a dangerous change when the introduction of a new method causes a different method to be called on recompilation (see Appendix B. It is also possible to overload parameters so that there is ambiguity and the changed code will fail to compile.

Even though making a binary compatible change does not guarantee that the change is trouble free, knowing that a change is binary compatible (and in our terms also not dangerous) provides some assurance of success.

3.1 Tool Support

The core of DJVCS is the binary compatibility (BC) checker. This component consists of a *Rule Engine* and the *Rules*. The Rule Engine needs to support the dynamic loading of rules, comparison of two versions of a component (in jar files) against the rules it has loaded, and production of a report of the analysis.

It was implemented using Java's reflection mechanism. A generic `Rule` class does the majority of the work, with each rule implementing its logic in a subclass.


Rule Server - Konqueror
Location Edit View Go Bookmarks Tools Settings Window Help
http://localhost:8080/djvcs/rules.jsp

DJVCS Rule Server

[Index] [Library Server] [Rule Server]

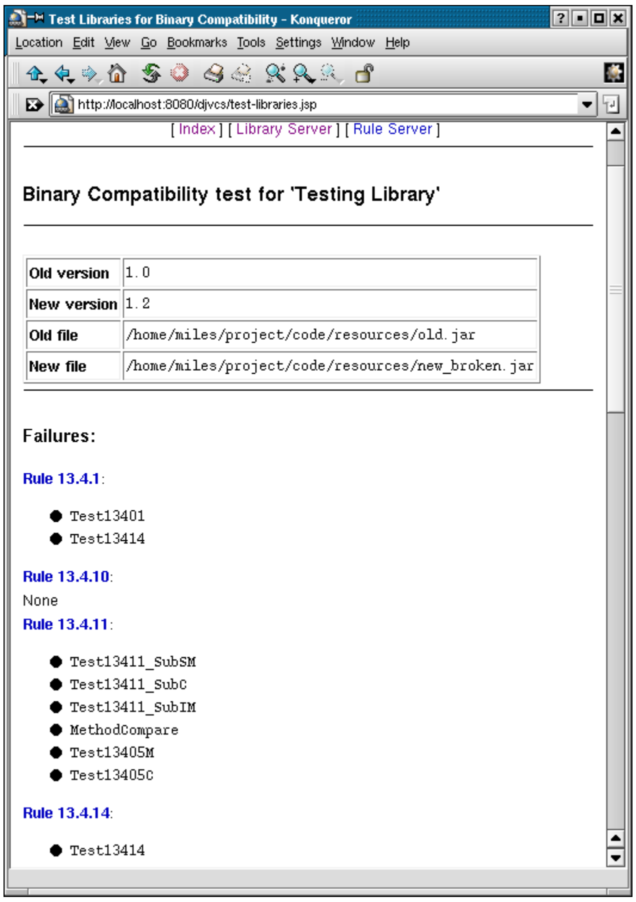
Rules registered on the system

Rule Name	Description	Remove
Rule 13.4.5	Ensures no non-private constructor, method or field has been removed	Remove
Rule 13.4.4	Ensures no superclasses or superinterfaces have been removed from the inheritence hierarchy.	Remove
Rule 13.4.3	Ensures no public class has been made non-public.	Remove
Rule 13.4.2	Ensures no non-final class has been made final.	Remove
Rule 13.4.1	Ensures no non-abstract class has been made abstract.	Remove
Rule 13.5.3	Checks new methods are not added to interfaces.	Remove
Rule 13.4.17	Ensures no non-static methods are made static and vice-versa.	Remove
Rule 13.4.15	Ensures no non-final methods are made final.	Remove
Rule 13.4.14	Ensures no concrete methods are made abstract.	Remove
Rule 13.4.11	Ensures no non-private method or constructor is deleted or illegallyaltered.	Remove
Rule 13.4.9	Ensures no non-private fields have their staticness altered.	Remove
Rule 13.4.10	Ensures no illegal reduction in access levels has been made.	Remove
Rule 13.4.8	Ensures no non-final fields are made final.	Remove
Rule 13.4.7	Ensures no illegal changes to field declarations have been made	Remove

Figure 2. Rule Server Index

Binary compatible changes that allow existing programs to still link, but where the existing programs may not recompile [7], are marked as dangerous changes, but not binary incompatible.

To test for binary compatibility between versions of a library, they need to have been loaded on the server. The


Test Libraries for Binary Compatibility - Konqueror
Location Edit View Go Bookmarks Tools Settings Window Help
http://localhost:8080/djvcs/test-libraries.jsp

[Index] [Library Server] [Rule Server]

Binary Compatibility test for 'Testing Library'

Old version	1.0
New version	1.2
Old file	/home/miles/project/code/resources/old.jar
New file	/home/miles/project/code/resources/new_broken.jar

Failures:

Rule 13.4.1:

- Test13401
- Test13414

Rule 13.4.10:
None
Rule 13.4.11:

- Test13411_SubSM
- Test13411_SubC
- Test13411_SubIM
- MethodCompare
- Test13405M
- Test13405C

Rule 13.4.14:

- Test13414

Figure 3. Test Results Page

operator then accesses the library server through a JSP interface and chooses the versions to be tested. The rule engine takes two versions of the library and applies the rules to them. Each rule class will take the two versions. It will then produce a return value based upon the comparison and possibly an error message.

To manage the rules on the system there is Rule Server Index (figure 2) where the rules already loaded can be viewed added to or removed.

A library developer tests a new library from a testing screen where the two versions to be tested against each other are entered. Once the test has been conducted the results will be shown (figure 3). In the screenshot you can see the library failing several binary compatibility tests.

4 The Server

The Distributed Java Version Control System takes care of library management duties as well as the binary compatibility checking, as described in the previous section. The server needs to maintain a record of what it has done so it knows which libraries it should distribute to each client. Initially communication between the server and client will be kept to a minimum, merely the transferring of the appropriate library. Client/server communication is implemented using RMI.

All libraries that are uploaded to the server are stored as are the previous versions (compatible or not) of each library. It contains a record of all binary compatibility tests that have been conducted and their results in addition to all the binary compatibility rules uploaded to it (until they are deleted). The server informs the client of the latest available binary compatible version of their library and sends the client the files they request.

The server side component consists of two separate servers, one for the libraries and one for the rules. It also includes a graphical interface to manage the libraries, the rules and testing.

The library server manages the libraries and the interaction with the rule server. The actual component consists of three smaller components, the actual server, a data structure to contain the libraries and finally a class to handle a single library and all of its versions. The `LibraryServer` acts as a wrapper to the data structure and also handles communications with the client. The system can handle several different libraries, the different versions of each library and all the dependencies between them.

For persistent information we used XML since it is now considered a standard for storage of information. Several XML parsers exist and we used the Apache Project's Xerces parser [27].

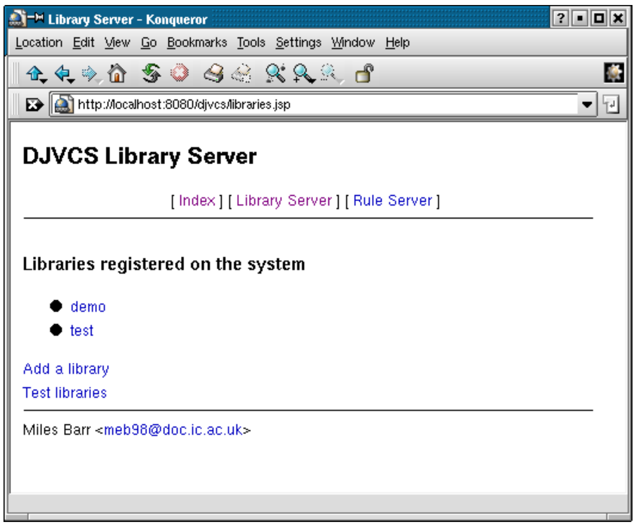
The `Library` class handles the information for a single library and its various versions. Calls are eventually passed down to this class to determine which versions are *safe*.

4.1 The Rule Server

The rule server is responsible for rule registration and persistence. It acts as the intermediary between the library server and the rule checking engine. The data file for the rules are just a list of the class files associated with the rules, since all other information is contained within them.

This component acts as the cornerstone for the entire system. It keeps track of all the libraries and rules and also performs the testing. It is important that it can handle multiple libraries and multiple rules and not to lose this information. It also needs to be able to talk to any client of the system and provide them with the correct information and library files.

Once the server is started it can be accessed through a web interface. There is a library manager which presents the main index (figure 4) as its first screen. From here libraries can be added, tested or examined.


Library Server - Konqueror
Location Edit View Go Bookmarks Tools Settings Window Help
http://localhost:8080/djvcs/libraries.jsp

DJVCS Library Server

[Index] [Library Server] [Rule Server]

Libraries registered on the system

- demo
- test

Add a library
Test libraries

Miles Barr <meb98@doc.ic.ac.uk>

Figure 4. Library Server Main Index

After an existing library is chosen the details of the versions available and the file that holds that version (figure 5) are given. Here versions can be added and removed (including the entire library) and the different versions tested against each other.

Client applications will have to incorporate a component if they wish to make use of the framework. The purpose of the component is to communicate to the server and upgrade any libraries that are supported (through JMX). This component is the only point of contact between the client and server.

Clients can inform the server what library and which version they are using. Clients are upgraded to the latest binary compatible version or can decide to upgrade to a later non-binary compatible version, because they wish to use the newly provided functionality and are aware that they have to recode in order to do so.

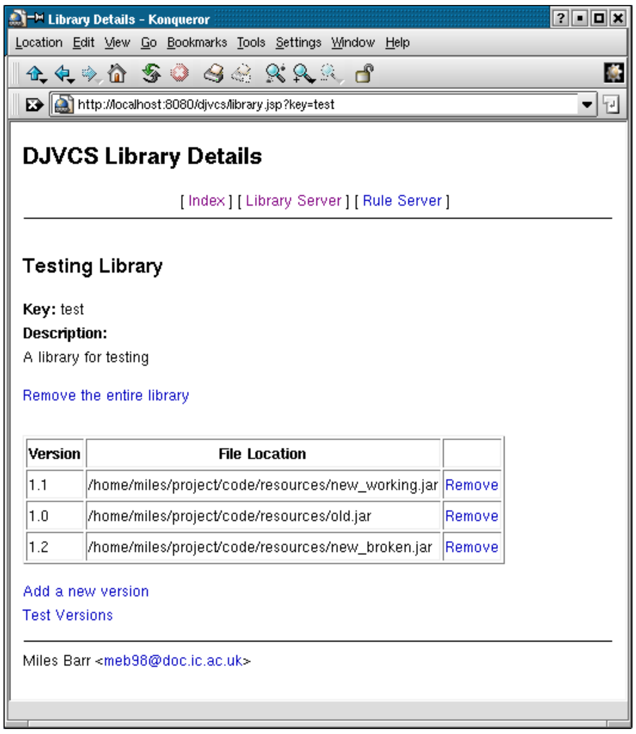
5 The Client

On the client side use is made of JMX [24], a standard Java framework for managing Java components. The library is the JMX component, it manages its jar file and has its own classloader. The client application must go through the JMX Agent (standard JMX procedure [8]) to instantiate classes from the library.

There is an updater component which when called performs the following steps:

1. Contact DJVCS server via RMI.

2. Retrieve latest binary compatible version information.


Library Details - Konqueror
Location Edit View Go Bookmarks Tools Settings Window Help
http://localhost:8080/djvcs/library.jsp?key=test

DJVCS Library Details

[Index] [Library Server] [Rule Server]

Testing Library

Key: test
Description:
A library for testing

Remove the entire library

Version	File Location	
1.1	/home/miles/project/code/resources/new_working.jar	Remove
1.0	/home/miles/project/code/resources/old.jar	Remove
1.2	/home/miles/project/code/resources/new_broken.jar	Remove

Add a new version
Test Versions

Miles Barr <meb98@doc.ic.ac.uk>

Figure 5. Library Detail Page

3. Open HTTP connection to server.

4. Download latest *safe* versions of main library and all dependent libraries.

5. Update local meta-data and mbeans (JMX modules).

There is a class loader component that exists to simplify object creation for the client application. Since we cannot override the system classloader all our objects have to be loaded via a custom classloader which can load objects from JMX library beans.

There is a reloading class loader component that exists within the `Library` mbean. Each time the jar file is updated it reloads itself with the latest class definitions so they are made immediately available to the running program. The program then can switch over to them. To accomplish this the mbean replaces its classloader when a new jar file has been made available.

The classloader has been specified to operate with minimal disruption, while still keeping the system dynamic. This is the component that allows the *dynamic* part of the upgrading process. It only loads classes that are in the same package as the library.

The functionality for updating libraries is provided by the framework but it is up to the client application to make the appropriate call. It is unlikely that the client needs updates on a regular basis, if they do, they can simply configure the framework to do so.

When the client makes the call to the `Updater` object it contacts the server and downloads the latest safe version then proceeds to update the client.

6 Performance

Earlier versions of this tool [6, 7] were concerned with the time taken to download files and the loading of classes (since it was done over a network). DJVCS makes use of a standard Internet protocol (HTTP) for file transfer hence performance issues of this aspect are beyond its control but it means that download speeds are as fast as any other you will experience on your connection.

Class loading performance has also be rectified because the library file now exists with the client so network performance does not affect it. There is the issue of the two levels of indirection (using a custom classloader, using JMX) that are necessary for retrieving a class object from the class loader but tests on the system have shown that changes in performance are negligible.

Times were recorded for binary compatibility checking on a two libraries consisting of 34 files (designed to either pass or break all the rules). Times ranged from 0.01 seconds up to 2.5 seconds for each rule, typically around 0.5 seconds. This level of performance is inline with previous engine implementations considering the library is larger and some rules cover multiple classes.

7 Related Work

The work described here arose directly out of theoretical work on binary compatibility done in collaboration with Drossopoulou and Wragg [4, 19]. As binary compatibility was conceived to assist with the development of distributed libraries [7, 6], we examined its effect on evolving libraries. We went on to model dynamic linking in [20] and we looked at the nature of dynamic linking in Java [5]. We have looked at the problems that arise with binary compatible code in [7] and built a less powerful tool, described in [6]. Other formal work on distributed versioning has been done by Sewell in [22], but this work does not consider the issue of binary compatibility.

Other groups have studied the problem of protecting clients from unfortunate library modifications. [17] identified four problems with ‘parent class exchange’. One of these concerned the introduction of a new (abstract) method into an interface. The other issues all concern library methods which are overridden by client methods in circumstances where, under evolution, the application behaviour is adversely affected. To solve these problems, *reuse contracts* are proposed in order to document the library developer's design commitments. As the library evolves, the terms of the library's contract change and the same is true of the corresponding client's contract. Comparison of these contracts can serve to identify potential problems. We hope to extend our system so that it can deal semantic compatibility.

Mezini [15] investigated the same problem (here termed horizontal evolution) and considered that conventional composition mechanisms were not sophisticated enough to propagate design properties to the client. She proposed a *smart* composition model wherein, amongst other things, information about the library calling structure is made available at the client's site. Successive versions of this information can be compared using reflection to determine how the client can be protected. These ideas have been implemented as an extension to Smalltalk.

[13, 10, 25] have done work on altering previously compiled code. Such systems enable library code to be mutated to behave in a manner to suit the client. Although work on altering binaries preceded Java [25] it came into its own with Java since Java bytecode is high level, containing type information. In both the Binary Component Adaptation System [13] and the Java Object Instrumentation Environment [10] class files are altered and the new ones are loaded and run. One of the main purposes of this kind of work is extension of classes for instrumentation purposes but these systems could be used for other changes. We have not taken the approach of altering library developers' code because it makes the application developer responsible for the used library code. Responsibility without source or documentation is not a desirable situation. There is also the problem of integrating new library releases, which the client may not benefit from.

Often configuration management is about configuration per se – technical issues about storage and distribution; or management per se - policy issues about what should be managed and how. Network-Unified Configuration Management (NUCM) [3] embraces both in an architecture, incorporating a generic model of a distributed repository. The interface to this repository is sufficiently flexible to allow different policies to be manifested.

World Wide Configuration Management (WWCM) [12] provide an API for a web based client-server system. It is built around the Configuration Management Engine (CME) to implement what is effectively a distributed project. CME allows elements of a project to be arranged in a hierarchy and working sets to be checked in and out, and the project as a whole can be versioned so several different versions may exist concurrently.

In this paper we have discussed changes that propagate without requiring explicit management. Where there are major modifications, which will need substantial rebuilding, Configuration Management systems such as those described will be necessary.

[28] have a framework with a two phase commit protocol for upgrading live software. There are also commercial tools where hot upgrading is supported. Macromedia's JRun 4 [14] has hot deployment as does Borland's Enterprise Server Appserver [1]. These tools leave it to the library developer though to do the update only when no harm will be done. They also don't have the capability of keeping generations of clients with the most up-to-date versions of components that won't break their code.

8 Conclusions

We have attempted to address the problems faced by a library developer who wishes to support remote clients, by taking advantage of Java's dynamic loading mechanism. Dynamic linking is supposed to allow clients to get the latest version of code — whether or not it will have disastrous effects. We have considered binary compatibility as a key criterion for safe evolution even though we are aware of its limitations and we have implemented all of the rules laid down in the Java Language Specification [11].

Several binary compatible changes are dangerous. Binary compatibility is clearly aimed at making sure pre-existing programs still run; if they still compile that's nice too but certainly not a major concern.

Examining source files as well as binaries may provide useful information. Reflection has been used to implement all the binary compatibility rules, but it has become clear it's nearing its limits while implementing some of the more complex rules and checking for *dangerous* changes.

There are situations where multiple dependencies can break the current system. Future versions need to address this problem by implementing version by version dependency information into the system.

One of the main reasons that this wasn't implemented in the current version was the choice of storage mechanism. DJVCS uses disk based XML files to store information and a custom data type to manages the libraries. This means that all logic in selecting information needs to be reflected in the XML format, parser, ADT and in Java code, i.e. it is non-trivial and *slow*. A more suitable format would be to use a database backend as it will give the developer a query language and off load processing onto a system that is optimized for it.

The rule engine is not used to its full potential. The next step is to add different types of rules. There are many different types of compatibility, e.g. ensuring a class only has a given memory footprint.

Up to now the only condition for declaring a library *safe* has been binary compatibility. It allows existing binaries to continue to link and run but does not consider the behaviour of the new library. Rule 13.4.20 ‘Method and Constructor Bodies’ allows any change to the body of a method or constructor. For critical systems you want to be sure that the only changes have been performance related and behaviour is exactly the same, i.e. logic compatible. There has been some work into this area [2] but not for Java. The nature of Java introduces levels of complexity not previously considered by logic checking frameworks.

The Java programming language and system has the hooks in it to enable programs to be dynamically upgraded. We have built a system that shows that the technology is there for safe dynamic upgrading. Our definition of safe is limited to ensuring that the upgrade will not cause errors at link time. It warns client developers of possible future problems with binary compatible changes. It also makes available non-binary compatible versions so that client developers can decide to build in upgrades to these in a controlled fashion.

Maintainers of software have many different properties, in addition to not throwing link errors, that they might wish to ensure. We hope to tackle some of these in the future.

Acknowledgements

Some of this work is based on more formal work done with Sophia Drossopoulou. We thank the DEJaVU implementor Shakil Shaikh for the predecessor of DJVCS. Finally we would like to thank Sophia and all the project students especially Andrew Shuttlewood and Darius Fidgett of our weekly meeting group for their helpful insights.

References

[1] Borland. Enterprise Server Appserver. info.borland.com/bes/appserver/ase1/feaben.html, 2002.

[2] J. Cook and J. Dage. Highly reliable upgrading of components. In *International Conference on Software Engineering*, pages 203–212, 1999.

[3] D. Hoek and M. Heimbigner, and A.L. Wolf. Versioned Software Architecture. In *Proc. of the 3rd International Software Architecture Workshop*, November 1998.

[4] S. Drossopoulou, S. Eisenbach, and D. Wragg. A Fragment Calculus: Towards a Model of Separate Compilation, Linking and Binary Compatibility. In *Logic in Computer Science*, pages 147–156, 1999.

[5] S. Eisenbach and S. Drossopoulou. Manifestations of the Dynamic Linking Process in Java. www-dse.doc.ic.ac.uk/projects/ slurp/dynamic-link/linking.htm, June 2001.

[6] S. Eisenbach and C. Sadler. Changing Java Programs. In *IEEE Conference in Software Maintenance*, November 2001.

[7] S. Eisenbach, C. Sadler, and S. Shaikh. Evolution of Distributed Java Programs. In *IEEE Working Conf on Component Deployment*, June 2002.

[8] M. Fleury and J. Lindfors. *JMX: Managing J2EE with Java Management Extensions*. Sams Publishing, 2002.

[9] I. Forman, M. Conner, S. Danforth, and L. Raper. Release-to-Release Binary Compatibility in SOM. In *Proc. of OOPLSA’95*, October 1995.

[10] J. C. G. Cohen and D. Kaminsky. Automatic Program Transformation with JOIE. In *USENIX Annual Technical Symposium*, 1998.

[11] J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java Language Specification*, pages 251–273. Addison Wesley, 2 edition, June 2000.

[12] J. R. J. J. Hunt, F. Lamers and W. F. Tichy. Distributed Configuration Management Via Java and the World Wide Web. In *In Proc 7th Intl. Workshop on Software Configuration Management*, 1997.

[13] R. Keller and U. Holzle. Binary Component Adaptation. In *Proc. of the European Conf. on Object-Oriented Programming*. Springer-Verlag, July 1998.

[14] Macromedia. Jrun 4. software/jrun/ productinfo/upgrade/, 2002.

[15] M. Mezini and K. J. Lieberherr. Adaptive Plug-and-Play Components for Evolutionary Software Development. In *Proc. of OOPSLA*, pages 97–116, 1998.

[16] Microsoft. C# Introduction and Overview. msdn.microsoft.com/vstudio/techinfo/articles/ upgrade/Csharpintro.asp, 2002.

[17] K. M. P. Steyaert, C. Lucas and T. D’Hondt. Reuse Contracts: Managing the Evolution of Reusable Assets. In *Proc. of OOPSLA*, 1996.

[18] S. Pratschner. Simplifying Deployment and Solving DLL Hell with the .NET Framework. msdn.microsoft.com/, November 2001.

[19] D. W. S. Drossopoulou and S. Eisenbach. What is Java binary compatibility? In *Proc. of OOPSLA*, volume 33, pages 341–358, 1998.

[20] S. Drossopoulou and G. Lagorio and Susan Eisenbach. Flexible Models for Dynamic Linking. In *Proc. Eurpopean Symposium of Programming*. LNCS, 2003.

[21] Y. Sauvageau. Private Email Correspondence, Cisco Systems, Inc., Feb 2003.

[22] P. Sewell. Modules, Abstract Types, and Distributed Versioning. In *Proc. of Principles of Programming Languages*. ACM Press, January 2001.

[23] Sun. Java product versioning specification. Technical report, Sun Microsystems, November 1998.

[24] Sun. Java Management Extensions Instrumentation and Agent Specification, v1.0. Self printed and bound, 901 San Antonio Road, Palo Alto, CA 94303, U.S.A., July 2001. Final Release.

[25] R. Wahbe, S. Lucco, and S. Graham. Adaptable Binary Programs. In *Technical Report CMU-CS-94-137*, 1994.

[26] S. Wiltamuth and A. Hejlsberg. C# Language Specification. msdn.microsoft.com/, January 2002.

[27] Xerces. Xerces-j. http://xml.apache.org/xerces-j/index.html.

[28] L. Yu, G. Shoja, H. Muller, and A. Srinivasan. A Framework for Live Software Upgrade. In *Proc. of ISSRE’02*, November 2002.

Appendix A: The Dangerous Binary Compatibility Rules

Language Spec. Section	Name	Description
13.4.6	Access to Members and Constructors	Reducing access levels to members and constructors may cause linking problems, but they are allowed.
13.4.7	Field Declarations	A field may not be made less accessible than before. A field may not change its staticness. A non-private field may not be deleted. A field can change its type and still link but may cause future compiling problems.
13.4.21	Method and Constructor Overloading	Overloading does not break binary compatibility but may have other side effects.(see figure 8)
13.5.3	The Interface Members	Added methods to an interface does not break binary compatibility but will cause recompilation problems.

Appendix B: Binding to Different Methods

This example comes from the Java Language Specification [11].

```
class Super {
  static void out(float f) { System.out.println("float"); }
}

class Test {
  public static void main(String[] args) {
    Super.out(2);
 }
}
```

is compiled and produces the output:

```
float
```

Suppose a new version of class `Super` is produced:

```
class Super {
  static void out(float f) { System.out.println("float"); }
  static void out(int i) { System.out.println("int"); }
}
```

If only `Super` is recompiled there is no change to the output, but if `Test` is now recompiled the output becomes:

```
int
```