Change and Adaptive Maintenance Detection in Java Software Systems

Derek Rayside
Systems Design Engineering
University of Waterloo
drayside@amorgos.uwaterloo.ca

Scott Kerr
Object Technology International Inc.
Ottawa, Canada
Scott_Kerr@oti.com

Kostas Kontogiannis
Electrical & Computer Engineering
University of Waterloo
kostas@amorgos.uwaterloo.ca

Abstract

Java is a relatively new programming language that is gaining popularity due to its network-centric features and platform independence ('Write Once, Run Anywhere'). This popularity has caused rapid evolution in the libraries that are available for Java applications. This evolution, in combination with Java's run-time linking, may cause incompatibilities between an application and the library it depends on: an application may execute with a different library version than the one it was compiled for. This paper presents techniques to automatically detect change in a library from its bytecode (binary) representation, and to apply the impact of those changes to any Java application. This paper also includes results of change detection experiments performed on the standard Java library (JDK).

1 Introduction

Most software applications depend on a set of library functions which are either generic (e.g. Unix C libraries) or specific to an application (e.g. matrix manipulation libraries). Applications written in Java are not an exception. Indeed, all Java applications necessarily depend on the Java Development Kit (JDK) library, since all objects in Java are descendents of `java.lang.Object`[1]

This paper discusses and proposes techniques for the identification of possible failure points due to incompatibility that may occur when an application written in Java is ported from one JDK version to another. The techniques presented here are general enough to be applied to any Java application and any Java library (not just the JDK).

Java is a network-centric environment, with applications and libraries being retrieved from disparate locations, often just before execution. Therefore, each execution of a Java software system could occur in a different configuration. This is also known as a dynamic configuration [Katz90].

The implication of Java systems representing dynamic configurations is that no party is in direct control of system configuration management. As an example, consider Sun's 'Java Plug-in' (formerly known as 'Activator') which automatically down-loads new versions of the JDK to a user's machine.

A Java software system is a self contained set of bytecode which is divided into application [2] and library subsystems. The important distinction is that no component in the library may depend on a component in the application (by definition). For the purposes of this analysis the application is considered invariant and the library may change. The application may need to be adapted to the library change after the analysis is complete.

Within this context, this paper discusses methods

*This work was funded by IBM Canada Ltd. Laboratory: High Performance Java Group and Centre for Advanced Studies. Acknowledgments to Erik Hedges, Mohammed Ladha and Proesenjit Lahiry of Systems Design Engineering at the University of Waterloo for their contributions.

[1] In other words, Java has a singly-rooted inheritance hierarchy [Eckel98].

[2] An application may include applets, beans, servlets, or code from other parties.

0-8186-8967-6/98 $10.00

for:

- identification and detection of interface-related failures that may arise in an application when the library interface changes. Interface failure is similar to a linkage error and precludes system execution. Interface failure can be computed deterministically by a static verification tool [Somm96].
- identification and detection of possible regression in the application due to an implementation change in the library. Regression is a loss of system functionality due to a change in some component [Myers79]. Regression is a subtle and complicated problem: it cannot be computed deterministically by a static verification tool [Somm96]. The application must be tested with the new library to determine if regression has occurred. Therefore, 'possible regression' is detected here.

In order to perform the above tasks a) Java source and bytecode have to be represented at a higher level of abstraction so that dependencies between different software components in a library can be modeled and revealed; ***b)*** a set, of Java features and dependencies that may affect portability has to be selected; *c)* a set of routines that automatically detect changes on the selected features and dependencies from one JDK version to another must be developed and finally; d) interface or implementation related library changes must be associated with the application's source code.

Our analysis is based on bytecode from which a set of program related facts are extracted. The *facts* are tuples represented in Rigi Standard Form (RSF) [Muller93] dare of the form ***[relation*** $entity_1$ $entity_2$]. The set of *relations* considered during the extraction of *facts* from bytecode conform to a domain model, the structure of which is discussed in more detail later in the paper. Moreover, these relations represent interface and implementation characteristics of a Java software component. Simple matching is applied to detect changes that may occur on the selected features from one library version to another. An entity-relation diagram that represents dependencies among the components is then used to propagate the detected changes to the application.

This paper is structured as follows: Section 2 discusses related work in the area of code similarity and JavaCheck from Sun Microsystems. The domain model for analysis of Java software systems is presented in Section 3. Section 4 identifies a technique to detect change between two versions of a Java library, and Section 5 discusses a technique to apply the knowledge of library change to detect required adaptive maintenance in an application. Section 6 briefly discusses how this tool would be used. Section 7 presents results of change detection experiments conducted on the JDK (versions 1.0.2 through 1.2beta3). Finally, conclusions are presented in Section 8.

2 Related Work

The work presented here is related to other work in industry and in research. The most similar industrial tool is JavaCheck from Sun Microsystems, which is discussed below.

This work is also related to past research in code similarity detection techniques. One major difference between this work and past research in pattern matching is that, here certain components are assumed to have similar patterns, and differences are detected. Past research in pattern matching often does not, start with this assumption, and instead looks for similarity amongst code that is assumed 'co be different. Nevertheless, the advanced pattern matching techniques for discovering code similarity discussed below would improve this work.

2.1 JavaCheck

JavaCheck from Sun Microsystems is the work closest to our area of application. The purpose of JavaCheck is to test an application for compatibility with a Sun defined subset of the JDK (such as PersonalJava). However, there are significant differences between JavaCheck and this work:

- JavaCheck does not, ***detect*** **change** between two versions of a library; the changes must be manually encoded in a platform *specification file.*
- The domain ***model*** used by JavaCheck is much less sophisticated than the one presented here. JavaCheck determines dependency relations by scanning the bytecode for strings. The domain model presented here involves an advanced analysis of the system's components and their interrelations (instances of which are also extracted from the bytecode).
- JavaCheck only investigates one form of adaptive maintenance that may be required when an application is executed with an alternate library version: interface failure. The tool presented here also looks for ***possible*** regression.

2.2 Syntactic based matching

Within the framework of syntactic matching, Baker, [Baker95] represents source code as a stream of strings. The approach uses parameterized pattern matching techniques based on a variation of a variation of the Boyer-Moore algorithm to identify duplication within a string. The algorithm allows for a mechanism to identify global substitutions between substrings so that strings can be matched by applying the substitutions.

Paul, [Paul94] proposes a system (SCRUPLE) in which regular-expressions are used to locate programming patterns in a large software system. Pattern matching is performed by testing if a code fragment is accepted by the automaton that is constructed by a regular-expression provided by the user, as a query.

Johnson, [Johnson94] uses a similar text based approach where fingerprints in source files are computed using a hashing mechanism. Fingerprints are compared to identify an overall similarity between two texts.

In [Jankowitz88], [McCabe90], [Konto97] statistical measurements and software metrics are applied in order to compute a fingerprint of a software component.

In [Wu92], [Muth] a Unix utility called *agrep* allows for approximate matching between a regular expression-like pattern and text in source code or plain text files.

2.3 Semantics based matching

Operational Semantics [Stoy77] is an alternative way of representing program behavior, in terms of *how* a computation is performed. Specifically, operational semantics focus on how the program ***states*** are modified during the execution of a statement. ***States*** are modified using a transition relation. Operational semantics are divided into two major approaches Natural Semantics and Structural Operational Semantics.

In Natural Semantics a transition relation specifies the relationship between the initial and the final state of the execution of each statement. Semantic equivalence of statements is defined in terms of the relation transition system. When *axioms* and rules are used to derive the final state after a statement is executed, a derivation tree can be obtained. The root of the tree is the statement and the initial state, and the leaves of the tree are instances of axioms. The internal nodes are conclusions of instantiated rules and their corresponding premises are their immediate sons.

Structural Operational Semantics place the emphasis on the individual steps of a statement's execution. The transition system is different than that of natural semantics since here we deal with sequences of transitions and not only a transition from an initial to a final state.

Denotational Semantics [Stoy77] offer a powerful method for representing programs as mathematical entities and then being able to reason about their dynamic behavior. The idea in this representation scheme is to define a semantic ***function*** which maps each syntactic entity to its meaning in terms of the *effects* it induces when is computed. In the denotational semantics approach there is a semantic clause for each of the basic syntactic constructs of the language [Letovsky88]. For composite syntactic constructs there is a semantic clause which is defined in terms of semantic clauses for the immediate constituents of the composite construct. The approach of denotational semantics is to use the static structure to organize the presentation of the dynamical or behavioral aspects of the program behavior.

2.4 Descriptive Matching

Descriptive matching offers a set of tools and techniques to identify differences that may occur between two code fragments by matching on a set of data and control flow features.

In [Mili97] a software representation, matching, and retrieval mechanism based on relational specifications is introduced. matching can be refined focusing on desired properties of the code segment sought. The approach is based on formal specifications and on the refinement ordering between specifications.

Canfora, and Cimitile [Canfora94], [Canfora92], [Cimitile96] describe efficient algorithms for analyzing the control and data flow in order to identify binding conditions on program variables. Versions of programs with altered binding conditions become candidates for portability failures.

In [Wills93] a program understanding system that uses attributed data flow sub-graphs to represent programs and programming plans is presented. Comparison is performed by matching sub-graphs and by checking constraints involving control dependencies and other program attributes.

In [Gra92] a program differencing tool for C is discussed. The tool, named CIAdiff, allows for comparing program features of corresponding versions by detecting changed, deleted, and added program entities.

In [Murphy96] a lightweight approach for generating flexible and tolerant source model extractors from lexical specifications is discussed. The source models provide means to represent source code features as

well as file dependences, event interactions and, call graphs. Program features can be used to identify modified code between different versions.

Other approaches in the category of descriptive matching, applied though to the area of syntax based editing, include CIA [Chen90], CSCOPE [Steffen85] and, the Pan system [Ballance92].

3 Domain Model

The domain model for Java software systems presented here consists of *entities, relations* and *attributes*. The purpose of this model is to represent components of Java software system and their interrelationships at, a higher level of abstraction. Instances of this model are extracted from Java bytecode and expressed in Rigi Standard Form (RSF).

3.1 Entities

This model contains two categories of entities: *components* and comparisons. Components have attributes, that describe their interface and implementation, and relations to other components. Comparisons occur between two versions of a component, and have attributes that describe the change in that component.

Most of the components in the model are instances of Java data types. The Java data types are organized in a hierarchical fashion. There are two main categories: Primitive and Reference. The Reference types are subdivided into Array and **ClassOrInterf** ace, which is made up of the Class and Interface types [LS97]. The Java notion of an Interface is a special type of abstract class; it is distinct from the general concept of interface. The difference between a Java Class and a Java Interface is so small in our comparisons that it makes sense to model both as ClassOrInterface. This is the approach adopted in [LS97] and [VM97].

This model does not represent Packages explicitly: a Package is a mechanism used in Java to partition the ClassOrInterface namespace in a hierarchical fashion. A Package correlates to a directory where the source code and bytecode are kept, and is included in the *fully* qualified name [VM97] of a **ClassOrInterface**.

The model has a comparison entity for each component that the developer can declare: **ClassOrInterfaceComparison**, **FieldComparison**, and **MethodComparison**.

The common notion of interface, as it applies to a ClassOrInterface, isthe aggregate interface of the declared Fields and Methods. This is contrary to the notion employed here: a **ClassOrInterf** ace is recognized as a component unto itself, not just an aggregation of other components (as a Package is). For example, a non-public **ClassOrInterface** cannot be used for casting by methods declared in other Packages. This example illustrates that a ClassOrInterface has an interface that is completely independent of its declared Fields and Methods. Similarly, the implementation of a ClassOrInterface is considered independently of its declared Fields and Methods.

The following simple application is used as an illustrative example of the domain model. The fully qualified **ClassOrInterface** names are not used here in the interest of clarity: the fully qualified name of Object is **java.lang.Object**.

```
/* A simple HelloWorld application */
public class App {
  public static void main(String[] args) {
    PrintStream p = System.out;
    p.println("Hello Hawaii!");
```

The entities extracted from the bytecode representation of this application (in RSF) are:

```
type App ClassOrInterface
type App() Method
type App.main(String[]) Method
type PrintStream ClassOrInterface
type PrintStream.println(String) Method
type Object ClassOrInterface
type Object0 Method
type String ClassOrInterface
type String[] Array
type System ClassOrInterface
type System.out Field
type void Primitive
```

Notice the presence of Object and its default constructor. By default all Java Reference types extend Object, even if this is not explicit in the source code. The bytecode presents a more detailed representation of the software.

3.2 Relations

The model presented here contains two categories of relations: definition and use. All relations represent a dependency between two components, and the direction of the relation is defined by the dependency [Muller93]. Therefore, these relations define a dependency graph of the entities.

Only use relations may cross the boundary from the application to the library. In other words, the application cannot define any components in the library and the library cannot define any components in the application.

The direct use of a ClassOrInterface component is represented by such relations as new, checkcast and instanceof.

Returning to the previous example, the definition relations are (in RSF):

```
fieldDeclaredBy System.out System
methodDeclaredBy App() App
methodDeclaredBy Object() Object
methodDeclaredBy App.main(String[]) App
methodDeclaredBy  PrintStream.println(String)
   PrintStream
```

It is important to note that the entity declared depends on the entity that declares it: all relations define a dependency. A subset of the use relations from the example is presented below:

```
getstatic App.main(String[]) System.out
ParameterType App.main(String[]) String []
ReturnType App.main(String[] ) void
Invoke App() Object ()
Invoke App.main(String[])
   PrintStream.println(String)
```

Note that the relations implicitly represented in a method signature, such as ParameterType, are explicitly represented in the model.

The complete model contains 7 distinct types of definition relations and 34 distinct types of use relations.

3.3 Attributes

Components have attributes that describe their interface and implementation. Comparisons have attributes that describe the change between two versions of a component. The attributes of components are used (with other information) to compute the attributes of the comparisons. All attributes take primitive values and are encoded in RSF.

The notion of 'attribute' used in this model is separate and distinct from the notion of 'attribute' used in [VM97].

3.3.1 Component Attributes

The most important interface attribute is the Signature. The Signature is the set of names which uniquely identifies each component.

The AccessModifiers are a set of booleans, some of which relate to interface and some of which relate to implementation.

The most interesting components are the ClassOrInterface, Field and Method, as they can be declared by the application developer. The attributes of these components are:

The Signature of a ClassOrInterface is its fully qualified name (i.e. including the Package name). Its interface defininition includes the public, abstract and interface AccessModifiers. Its implementation is defined by the final and super AccessModifiers.

The Signature of a Field is the signature of the class that declares it combined with its proper name and the signature of its type. Its interface definition includes the public, protected, private and static AccessModifiers. Its implementation is defined by the final, volatile, and transient AccessModifiers.

The Signature of a Method is the signature of the class that declares it combined with its proper name and the signatures of its parameters and return type. Its interface definition includes the public, protected, private and static AccessModifiers.

The implementation of a method is defined by its body, and the final, synchronized, native and abstract AccessModifiers. The Method body is not a primitive value; its change is summarized in the MethodComparison entity.

3.3.2 Comparison Attributes

The attributes of comparison entities describe the change between two versions of a component. All of the values for these attributes must be computed; this computation is discussed in detail in the next section.

The ClassOrInterfaceComparison and FieldComparison components have only two attributes: interf aceChanged and implementationChanged. These attributes take boolean values, and are computed by examining the change in attribute values of the component entities.

MethodComparison is similar to FieldComparison except that it has two extra attributes: codeChanged and exceptionsChanged. These attributes also hold boolean values, and are used in the computation of the implementationchanged attribute.

4 Change Detection

Change between the expected and alternate versions of a library is detected by computing the attribute values for comparison entities. Comparison entities

are only created (and computed) for those components which are in the *library interface.* The library interface is the aggregate of all components which are visible to the application.

The phrases 'expected' and 'alternate' are specifically chosen so as to not imply ordering. The expected version of the library is the one that the application was compiled with. The alternate version is the one that the application is required to execute with. The alternate version may be newer or older than the expected version [Katz90].

It is more important to detect change in a component's interface than in its implementation. If a library component that the application depends on has a changed interface it will certainly cause the application to have an interface failure; there is no point in computing whether the library component has changed implementation.

The Signature of a component is the most important interface attribute (as mentioned above), and components are first matched on Signature.

4.1 ClassOrInterface

The **ClassOrInterf aceComparison** interf **aceChanged** attribute is set to true if the Signature has changed or the **ClassOrInterface** has been deleted. Otherwise, it is computed by examining the attributes of each version of the **ClassOrInterface.** If the public flag has been turned off or the abstract or interface flags have changed state then the **interfaceChanged** flag is set to true.

The implementationchanged value is computed after the interf **aceChanged** value. The **implementationChanged** value is set to true if any of the implementation attributes have changed values.

4.2 Field

The **FieldComparison** interf **aceChanged** attribute value is set true if the **ClassOrInterface** that declares it has been deleted, changed Signature, or had its public flag turned off. Otherwise, the attribute values of the Field are examined, and any reduction in visibility or change in the static value causes interf **aceChanged** to be true.

If the type of a Field changes it will change the Field's Signature and cause the interf **aceChanged** value to be true.

The **implementationChanged** value is computed after the interf **aceChanged** value. The **implementationChanged** value is set to true if any of the implementation attributes have changed values.

Returning to the example: the static flag of the System. out field is true, as is commonly known. Suppose that a new version of the JDK was released where the static flag of this field was set to false. The **FieldComparison** entity for System. out would set the interf **aceChanged** bit to true.

4.3 Method

The **MethodComparison** interf **aceChanged** attribute value is computed following the same procedure for the **FieldComparison.** interf **aceChanged** attribute.

The **implementationChanged** attribute value is computed by first examining the values of the implementation attributes. If none of these have changed then the **codeChanged** and exception&hanged values are computed. The **implementationChanged** value is true if either **codeChanged** or **exceptionsChanged** are true.

The content of a method body is a list of opcodes. An opcode is a single byte instruction for the Java virtual machine, which may have zero or more arguments. The arguments for these opcodes are encoded as indices to the ConstantPool (symbol table) of the class file. To perform an exact match of two lists of opcodes it is necessary to de-reference their arguments from their respective ConstantPool's (the methods are in different class files).

The exceptions thrown by a method are represented by a set of indices to class Signatures in the ConstantPool. Performing an exact match of these sets is similar to performing an exact match of the code in that the ConstantPool references have to be de-referenced. However, the exceptions are a set and not a list, so a pseudo-join must be performed.

In the source code, the exceptions thrown by a method are considered part of the Signature, and the method cannot be invoked unless they are caught. The exceptions are not part of the bytecode Signature, and the restriction that they must be caught is only enforced by **javac** (the compiler) [VM97]. Therefore, exceptions are considered as part of a method's implementation here and not as part of its interface.

The comparison of the code and exceptions thrown could benefit from the advanced similarity techniques discussed in the Related Work section; in that case **implementationChanged** would be a real number to indicate how much the implementation had changed, instead of a boolean as it is here.

Suppose from the previous example that the **PrintStream.println(String)** method has been altered and now throws an **IOException** if it cannot write to standard out. This change will be

detected when the exceptions thrown are examined, and the `implementationChanged` attribute of the `MethodComparison` entity representing this method will be set to true.

5 Adaptive Maintenance Detection

Now that change has been detected in the library subsystem, the application must be analyzed to determine how to propagate that change (or if propagation is required at all). If the change in the library affects the application, the application will have to be adapted to run on the alternate version of the library.

The steps required to detect where the application requires maintenance to adapt are worked through below with the previous example:

The first step is to determine the dependencies that the application has on the library. This information can be extracted automatically from the dependency graph in the form of all of the use relations which cross the application-library boundary. In the example these relations are:

```
Extends App Object
Invoke App() Object()
ParameterType App.main(String[]) String[]
getstatic App.main(String[]) System.out
Invoke App.main(String[])
   PrintStream.println(String)
```

The next step is to examine the comparison entities for each of the library components that the application depends on. In the example Object, Object `()` and String have not changed, but `System.out` and `PrintStream.println(String)` have.

The final step is to report any use relation that terminates in a changed library component to the developer. There are two categories of change, and two corresponding categories of adaptive maintenance to report: interface failure and *possible regression*. From the example, the getstatic relation will be reported as interface failure, and the Invoke relation that terminates in `PrintStream.println(String)` will be reported as possible regression.

6 Tool Description

The domain model and analysis process presented here are encoded in a tool intended for use by Java software maintenance programmers. This tool accepts three inputs: the application, the expected version of the library, and the alternate version of the library.

The tool reports the RSF arcs that are affected by interface change, and those that are affected by implementation change. The arcs affected by interface change in the library indicate portability failure in the application. The arcs affected by implementation change in the library indicate possible regression in the application.

A number of metrics could be developed from the output of the tool to try to assess the effort needed to modify the application for use with the alternate library version. For example: the number of arcs correlates to the number of lines of code in the application requiring maintenance; the number of source nodes indicates how localized the maintenance in the application is; and the number of target nodes indicates how many changes in the library the developer must contend with.

In the case of the example presented here, the application needs a (proportionally) large amount of adaptive maintenance to work with the alternate version of the library. It may not be feasible to adapt an application requiring this degree of maintenance. However, this decision is left to the developer.

7 Change Detection Experiments

This section presents a summary of results from change detection experiments conducted on the JDK versions 1.0.2 through 1.2beta3. These experiments were conducted on the Microsoft Windows versions of the JDK: there are small differences between the Windows and Unix versions of the JDK.

The purpose of these experiments is twofold: first, to demonstrate the necessity of this tool by showing the degree of change in the JDK; and second, to provide experimental verification for the domain model presented here.

These experiments were conducted in two sets to show the complementary portability problems faced by Java application developers: running old applications on new libraries, and running new applications on old libraries. Table 1 presents a summary of the change in the JDK measuring forward from the initial release (1.0.2). Table 2 presents a summary of the change in the JDK measuring backwards from the latest release (1.2beta3).

7.1 Experiments

The JDK contains two major sets of packages: the Java packages and the Sun packages. The Java packages are considered to be the 'core JDK' and the Sun

JDK Change Measurement	1.0.2	1.1.1	1.2b3
Fields			
Interface Change	0	1	1
Implementation Change	0	3	3
Same (no change)	261	257	257
Methods			
Interface Change	0	13	14
Deprecated	0	128	145
Implementation Change	0	533	743
Same (no change)	1864	1318	1107

Table 1: Change detection results measuring forwards from the first JDK release (1.0.2).

JDK Change Measurement	1.0.2	1.1.1	1.2b3
Fields			
Interface Change	2675	2010	0
Implementation Change	3	2	0
Same (no change)	257	923	2935
Methods			
Interface Change	13,353	10,536	0
Deprecated	145	45	0
Implementation Change	741	935	0
Same (no change)	1107	3730	15,201

Table 2: Change detection results measuring backwards from the latest JDK release (1.2beta3).

packages the 'undocumented' portion of the library. To be certified as 1100% Pure Java an application must use only the core JDK (i.e. the Java packages). Therefore, only results :for the Java packages are discussed in detail here.

These experiments detect change in the library interface: the aggregate of all components which are visible to the application. In practical terms this is the public or protected fields and methods in public `ClassOrInterf` aces.

The low level data collected from these experiments can be used to detect the adaptive maintenance required in an application, as discussed previously. Most applications will not use every component in the library. Therefore this summary information represents the upper bound on the degree of adaptive maintenance required for the application. The adaptive maintenance detection will identify which specific components in the library cause the required adaptive maintenance.

Tables 1 and 2 present a summary of the data for two sets of experiments. The first set (Table 1) compares JDK version 1.0.2 to all subsequent versions (only 1.1.1 is shown from the 1.1.x series). The purpose of this set of experiments is to assess the *backwards* compatibility of the JDK: will future versions of the JDK be compatible with applications written for version 1.0.2?

The second set of experiments (Table 2) compares JDK version 1.2 **beta3** to all previous versions (only 1.1.1 is shown from the 1.1.x series). The purpose of this set of experiments is to assess the *backwards portability* of the JDK: will an application developed for the latest version of the JDK work on past versions of the JDK?

Results for the JDK 1.1.x series are not displayed as the changes are so small, as would be expected from the version numbers.

Method 'deprecation' is a feature that is unique to Java. The purpose is to allow the library developer (i.e. Sun) to specify which methods will be removed from the library interface in *future* versions. This is an advance warning to application developers that the library is going to change, so that they may adapt their applications before the change actually happens.

Recall that interface change indicates that an application may have interface failure with the alternate version of the library, and implementation change indicates that the application may regress if used with the alternate version of the library.

7.2 Results

The first set of experiments (Table 1) indicate, as expected, that the JDK shows good backwards compatibility. In other words, an application written for JDK 1.0.2 is likely to execute with any future version of the JDK, although the application should be tested for regression.

The second set of experiments (Table 2) show how much the JDK has grown since its initial release in late 1995. In just three years it has grown almost tenfold. Therefore, applications developed with later versions of the JDK are less likely to execute with earlier versions - unless the developer takes care to use only those features of the JDK that are available in earlier versions. Newer features of Java, such as Beans, reflection, RMI, and the new event model, are not available in older versions of the JDK.

These two sets of experiments display internal consistency. Notice that the third column of table 1 is similar to the first column of table 2. The third column of table 1 measures change between JDK 1.0.2 and JDK 1.2beta3; the first column of table 2 measures change between JDK 1.2beta3 and JDK 1.0.2.

The following numbers are identical: fields with implementation change, fields with no change, methods deprecated, and methods with no change. It is expected that the fields with interface change and the methods with interface change numbers be different, as these numbers will include fields and methods 'deleted'. There is a small anomaly here: methods with implementation change from 1.0.2 to 1.2beta3 is 743, and from 1.2beta3 to 1.0.2 is 741.

Also note that no change is detected between JDK 1.0.2 and JDK 1.0.2 (table 1, column 1) or between JDK 1.2beta3 and JDK 1.2beta3 (table 2, column 3). These columns indicate the size of the respective JDK versions.

7.3 Details of Results

Table 1 indicates that the JDK has a few small points where it may not be backwards compatible: these are discussed here.

The three fields that have displayed implementation change are in the System class: in, out and err (the standard IO streams). These three fields have had the final flag turned on. This is an implementation change, not an interface change, so applications which use the standard IO streams will still execute, but should be re-tested with later versions of the JDK.

The single field with interface change is the `pushBack` field from the `PushbackInputStream` class. This field was protected, and has been deleted.

Seven of the thirteen methods with interface change between JDK 1.0.2 and JDK 1.1.1 are accounted for by the fact that the `Win32Process` class is no longer public. This class is not to be used by application developers, and is not present on Unix implementations of the JDK (there is a corresponding `Unix32Process` class).

The following methods have been deleted from the Abstract Window Toolkit (AWT), the graphical user interface portion of the JDK: `FramePeer.setCursor ScrollbarPeer.setValue ComponentPeer.handleEvent ComponentPeer.nextFocus`

The AWT had the largest change (where change does not include addition) of any segment of the JDK, due to the introduction of a new event model.

The `SecurityManager.checkPropertyAccess` and `DatagramSocket.finalize` methods have also been removed from newer versions of the JDK.

The Vector.`toString` method has been moved to Vector's new parent class, as a part of the enhancement of the JDK collections classes in JDK 1.2beta3.

8 Conclusions

This paper discussed a domain model which allows for the representation of Java bytecode components at a higher level of abstraction. A bytecode parser extracts RSF facts (entity relationship tuples) that conform with the domain model. An entity relationship structure is then used to detect the change between two versions of a component. A simple illustration of how this change information could be applied to an application to detect adaptive maintenance was presented.

These techniques are general enough for any arbitrary application and library that can be represented as Java bytecode. Analyzing bytecode representations of software instead of source code has three advantages: often the source code is not available, other languages can be represented in Java bytecode, and the bytecode explicitly represents certain relations that are only implicit in source code.

A prototype static verification tool has been built at the IBM Toronto Laboratory that can detect change and compute interface failure deterministically from bytecode. This tool can also identify 'possible' regression.

The detection of interface failures and possible points of regression due to library changes is a very important issue for network centric applications such as those written in Java. The proposed system reduces the maintenance time, and allows for easier integration of large Java applications.

The domain model and change propagation techniques presented here could be combined with more sophisticated change detection techniques (see Related Work) to identify possible regression points more accurately.

Finally, we presented the results of change detection experiments conducted on the standard Java class library, the JDK. These experiments show that the JDK has grown almost tenfold in the three years since its inception, but still retains good backwards compatibility. Therefore, the prototype tool will be most useful to detect adaptive maintenance in Java applications developed on new versions of the JDK, but deployed on older versions. Other libraries may show different results.

References

[Baker95] Baker S. B, "On Finding Duplication and Near-Duplication in Large Software Systems" *In Pro-*

ceedings of the Working Conference on Reverse Engineering *1995*, Toronto ON. July 1995i, pp. 86-95.

[Ballance92] Ballance, R., Graham, S., Van De Vanter, M., "The Pan Language-Based Editing System", *ACM* Transactions on Software Engineering and *Methodology,* Jan. 1992, Vol. 1, No.1, pp.95-127

[Canfora94] Canfora, G., Cimitile, A., DeLucca, A., "Software Salvaging Based on Conditions", *IEEE Conf.* on Software *Maintenance*, 1994, pp. 424-433.

[Canfora92] Canfora, G., Cimitile, A., Carlini, U., "A Logic-Based Approach to Reverse Engineering Tools Production", Transactions *of* Software Engineering, Vol.18, No. 12, December 1992, pp. 1053-1063.

[Chen90] Chen, Y., Nishimoto, M., Ramamoorthy, C., "The C Information Abstraction System.", *IEEE* Transactions on **Software** Engineering, vol.16, No.3, 1990, pp.325-334.

[Cimitile96] Cimitile, A., Munro, M., "An Improved Algorithm for Identifying Objects in Code ", **Software** Practice and Experience, vol.26, No.1, 1996 pp.25-48.

[Eckel98] Eckel, B., "Thinking In Java", Prentice Hall, 1998

[Gra92] Grass, J., "Cdiff: A Syntax Directed Differencer for C++ Programs", *Usenix Ct t* Conference, Portland, Oregon, Aug.1992, pp.181-193.

[Jankowitz88] Jankowitz, H., T., "Detecting Plagiarism in student PASCAL programs", Computer Journal, vol.31, no.1, 1988, pp. l-8.

[Johnson94] Johnson, H., "Substring Matching for Clone Detection and Change Tracking", International Conference on Software Maintenance 1994, Victoria BC, 21-23 September, 1994, pp.120-126.

[Katz90] Katz, R.H. "Toward a Unified Framework for Version Modeling in Engineering Databases" *ACM* Computing Surveys, Vol. 22, No. 4, December 1990.

[Konto97] Kontogiannis K., "Evaluation Experiments on the Detection of Programming Patterns Using Software Metrics", In Proceedings *of WCRE'97,* Amsterdam, *The* Netherland *s,*

[Letovsky88] Letovsky, S. "Plan Analysis of Programs", *Ph.D thesis,* Yale University, *New Haven CT, Dept. of* Computer Science, *YALEU/CSD/RR662,* December 1988.

[LS97] "The Java Language Specification", Sun Microsystems Addison-Wesley, 1997

[McCabe90] McCabe T., J., "Reverse Engineering, reusability, redundancy: the connection", American Programmer, vol.3, no.10, Oct. 1990, pp.8-13.

[Mili97] Mili, R., Mili, A., Mittermeir, R., "Storing and Retrieving Software Components: A Refinement Based System", *IEEE* Transactions on **Software** Engineering, July 1997, vol.23, no.7, pp. 445-460.

[Myers89] Myers, E., Miller W. "Approximate Matching of Regular Expressions", Bulletin *of* Mathematical *Biology,* vol.51, no.1, 1989, pp.5-37.

[Myers79] Myers, G. J. "The Art of Software Testing", John Wiley & Sons, 1979

[Muller93] Müller, H.A., "Understanding Software Systems Using Reverse Engineering Technology Perspectives from the Rigi Project" *In* Proceedings *of CASCON'93*, Toronto, ON. 24-28 Oct. pp. 217-*226.*

[Murphy96] Murphy, G., Notkin, D., "Lightweight Lexical Source Model Extraction", *ACM* Transactions on Software Engineering and *Methodologyvo1.5,* No.3, July 1996, pp. 262-292.

[Muth] Muth, R.,Manber U.,"Approximate Multiple String Matching",
http://www.glimpse.cs.arizona.edu/udi.html

[Paul94] Paul, S., Prakash, A., "A Framework for Source Code Search Using Program Patterns", *IEEE* Transactions **on** Software Engineering, June 1994, Vol. 20, No.6, pp. 463-475.

[Somm96] Sommerville, I. "Software Engineering", 5th Edition Addison-Wesley, 1996

[Stoy77] Stoy, J.E., Denotational Semantics, MIT Press, 1977.

[Steffen85] Steffen, J., "Interactive examination of a C program with Csope", Proceedings *USENIX* ASSOC., Winter Conference, Jan. 1985

[VM97] "The Java Virtual Machine Specification", Sun Microsystems Addison-Wesley, 1997

[Wills93] Wills, L.M., "Automated Program Recognition by Graph Parsing", *MIT* Technical *Report 1358,* MIT, AI Laboratory, 1993

[Wu92] Wu, S., Manber, U., "Agrep - A fast approximate pattern matching tool", *Usenix* Winter 92 Technical **Conference,** San Fransisco, Ca., January 1992, pp. 153-162.