Automatic Analysis of Java™ Program Evolution and its Relevance to Regression Testing

K.Rangarajan
Man Machine Systems
Email: mms@giasmd01.vsnl.net.in

ABSTRACT

All software tend to evolve. Some reasons for evolution are bug fixes, code re-factoring, adding new functionality, and modifying existing functionality. When a Java program evolves from one version to another, typically new classes get added and some of the existing classes get modified or deleted. Understanding how a Java program evolves helps in program maintenance, and more importantly, can help in minimizing class regression testing. This paper presents a language-based approach for analyzing program evolution. It shows how the notion of ***atomic changes*** provides a semantically richer vocabulary to characterize a Java program that undergoes changes across versions. Some atomic changes with respect to Java are changing class member order, changing member access level, defining new methods, and method renaming. Examples are presented to demonstrate that classes that evolve by certain permutations of atomic changes might not require any regression testing. A tool called JEvolve™ has been developed to automatically analyze Java programs and to identify those modified classes that require regression testing. Data obtained by analyzing various versions of Java Developers Kit using this tool are presented.

1. INTRODUCTION

All software tend to evolve. In fact, evolution indicates that the software is being actively used! Some reasons for the evolution are

- new functionality is added
- existing functionality is modified
- bugs are fixed
- software is re-engineered by using ***newer*** technology
- code re-factoring is performed
- parts of the code are rewritten to address efficiency concerns

When a program written in an object-oriented language such as Java evolves from one version to another, new classes may be added and existing classes may get modified. Understanding how the code has changed across versions can be useful for at least four reasons:

- we may want to know what sort of changes have occurred in the source, for example, which classes have changed, what new classes have been added, etc. This will assist in maintaining code;
- if we are the developers of a set of classes that have evolved from the previous version, we would like to know whether regression testing needs to be performed on the classes in the new version, given that the previous version is correct;
- as system testers, we would like to minimize regression test effort when developers release new system builds;
- a version control software might apply an ***intelligent*** merge of the various branches, depending on what has changed along the branches.

This paper presents a new, language-based approach for characterizing the way a Java program evolves from one version to another. Based on the notion of ***atomic changes***, the approach can be automated.

Section 2 briefly touches upon related research. Section 3 outlines some ways of characterizing an evolving Java program. Section 4 defines the concept of atomic changes and enumerates some of the atomic changes applicable to Java programs. The connection between atomic changes and class retesting is discussed in Section

5. Section 6 briefly describes a tool that automatically analyzes two versions of a Java program to suggest portions that needed to be retested. Summary of results obtained by running the tool on various versions of the Java Developers Kit are presented in Section 7. Finally, Section 8 presents the conclusions.

2. RELATED RESEARCH

Kung et al. [3] identify the types of changes that can be made to an OO library and provide a system that captures these and makes inferences about their impact on software maintenance. Hsia et al. [4] address the issue of test case selection for revalidation purposes. Their approach consists in computing the class firewall for a changed class and using that to identify which classes need to be retested when a program evolves. Rothermel and Harrold [5] use a program dependence graph to represent control and data dependencies in an OO program and use this to identify the statements in the modified program that will produce different test results. None of these approaches is language specific and hence does not take advantage of certain properties that can further reduce regression test effort. Our approach brings into play peculiarities of specific OO languages (we have considered in our research C++ and Java) and uses that knowledge to minimize regression testing. Whitmire [6] describes a small number of ***atomic operations*** characterizing design changes. The changes we have proposed are more fine-grained than his and occur at the programming level. Palay [7] describes a C++ system that understands certain compatible changes to an evolving class in order to minimize recompilation.

3. CHARACTERIZING JAVA PROGRAM EVOLUTION

One common way to characterize an evolved Java program is to say something like ***10 files were modified***, ***15 classes were changed***, ***8000 source lines were altered***, and so on. This kind of description is too coarse and is not a useful indicator of ***change complexity***. Instead, we would benefit by coming up with a formalism that has the following desirable properties:

- it should form the basis for designing meaningful indicators of the effort involved in evolution
- it should facilitate program maintenance by suggesting ways to correlate different versions
- it should supply clues to identify portions of the modified code that require retesting
- it should support automated analysis

Programmers who maintain multiple versions of software usually run a ***file differencing*** utility to comprehend the kind of changes that have taken place across various versions. Such a tool provides assistance by enumerating the lexical differences with respect to the program text, without understanding the ***meaning*** of such differences. Let us look at a simple example.

```
class A {
   int i;
   void setVal(int v) {
      i = v;
   }
}
```

If the above class is changed to

```
class A {
   void setVal(int v) {
      i = v;
   }
   int i;
}
```

the text differencing engine will show that the two source files are different since the order of class elements have changed. Although these differences are relevant from a purely ***textual*** point of view, they are not significant from a program behaviour perspective. Consider another example:

```
class Base {
   protected int value;
   // other elements
}

class Derived extends Base {
   private int value; // --- (1)
   void ff() {
      System.out.println(value);
   }
}
```

If the derived class is later changed to

```
class Derived extends Base {
   private int i; // --- (1a)
   void ff() {
      System.out.println(value);
   }
}
```

a text differencing utility will highlight the differences with respect to lines marked (1) and (1a), but will show no change with respect to the body of Derived.ff(). However, it is clear that the behaviour of Derived.ff() has changed (because of change in symbol binding) and from a regression testing perspective, the method requires to be retested. Reasoning like this is possible if program differences can be represented at a higher level than purely lexical. The concept of ***atomic changes*** proves useful here.

4. ATOMIC CHANGES

An atomic change is a change applied to the source code such that

1. it is minimal, that is it cannot be decomposed into simpler changes, and
2. it can be defined purely in terms of one or more language features.

The following are a few of the possible atomic changes in Java:

- Reorder elements of a class
- Change the access level of a member
- Make a class public
- Change a data member from instance to static
- Change the body of a method
- Change method signature
- Rename method arguments

Every time a Java program undergoes changes, the change can be represented as a sequence of one or more of such atomic changes. As an example, consider the following Java class:

```
class Sample {
   protected int i;
   public Sample(int v) {
      i = v;
   }
   public int getVal() {
```

```
        return i;
    }
}
```

If this class evolves to

```
public final class Sample {
    public Sample(int v) {
        value = v;
    }
    public final int getVal() {
        return value;
    }
    private final int value;
}
```

the following atomic changes are part of the evolution:

- class has been made final
- class has been made public
- elements of the class have been reordered
- protected data member has been made private
- data member has been made final
- name of the data member has been changed
- getVal() method has been made final

Characterizing program evolution in terms of atomic changes facilitates a more meaningful analysis of the evolution process than is possible via lexical differences. As will be shown in Section 5, impact on regression testing can be automatically analyzed.

Atomic changes satisfy three interesting properties [2]:

1. They are language-dependent
2. They are finite in number
3. They may or may not preserve class equivalence

Due to space constraints, these properties are not elaborated in this paper. Further details may be found in [2].

5. REGRESSION TESTING

Regression testing is performed on modified software to assure that changes introduced have no unintended effect on old functionality. Typically, the regression test suite is rerun on the modified software after every modification and the outcome compared with expected behaviour. Such a regression testing can be applied at unit level (typically, a class), cluster level (collection of classes) or at system level.

An important research question is ***Should a class be retested every time it changes?*** Another related question is ***If a class is modified, what are the other classes that must be retested?*** In the most common scenario, every time a class is changed, the entire regression test suite is rerun. This eliminates the risk of not testing a class whose behaviour has changed in the process of program evolution, but can be quite costly! Assume that we have written a fairly large application in Java comprising around 500 classes that uses JDK 1.1.6. Should we retest our entire application if we link it to JDK1.1.7 when that eventually becomes available? Imagine the nightmare of having to retest all classes in our application! Since testing requires several resources such as time, humans, and hardware, we would like to expend no more than what is needed for retesting the application.

Unfortunately, the optimum effort needed to retest a set of classes that have changed cannot be easily computed. However, certain clues can be derived from a careful study of the source changes. If in doubt, of course, we can always resort to retesting. Going back to the ***Sample*** class discussed in Section 4, should we retest the modified version? Despite seven atomic changes, it is safe to conclude that retesting the modified class is not necessary since nothing ***significant*** has changed. But, how do we know nothing significant has changed *without* retesting the affected class?

As pointed out in Section 4, an atomic change might or might not induce retesting on a class. If we catalog all atomic changes possible in a Java program, associating with each one its impact on retesting, then by examining the actual atomic changes that are part of a particular evolution, it is possible to decide whether or not retesting is needed. The assumption we will make for our analysis is that the modified program builds without errors. In the ***Sample*** class above, none of the atomic changes induces retest, and hence the modified class need not be retested!

A complete description of possible atomic changes in a Java program is beyond the scope of this paper. However, it would be of interest to know some of the atomic changes that induce retest on the affected class. The following is a partial list:

- changing method body
- adding a new method
- deleting an existing method
- renaming a method
- adding a base class

To see, for example, why renaming a method might change the meaning of a class, study the following code:

```
class Base {
   public void f() { g(); }
   public void g() { /* ... */ }
}

class Derived extends Base {
   public void f() { /* ... */ }
}
```

If the Derived class is later on changed to

```
class Derived extends Base {
   public void g() { /* ... */ }
}
```

the modified code will build without errors and provide the same interface to the rest of the code, but the ***behaviour*** has changed significantly.

6. AUTOMATING REGRESSION ANALYSIS

Can a tool automatically identify code changes in two or more versions of a Java program, and possibly suggest which classes and methods might require retesting? Such a tool can be of immense help in program maintenance, and additionally help in minimizing regression test effort. At Man Machine Systems, as part of our research in the area of object-oriented software testing, we have developed a tool called JEvolve™ that performs regression analysis of Java programs.

JEvolve takes as input multiple Java projects, where a project is defined as a collection of source files along with one or more ***jar*** files. The latter are required to resolve references to classes whose source might not be

available. When performing regression analysis, any two projects (corresponding to two different versions of the program) can be selected at a time. The tool compares the two sets of sources and identifies the different atomic changes that are part of the evolution. JEvolve is aware of 63 atomic changes that are possible in Java programs, and knows about whether or not each such change induces retest on a modified class.

Useful reporting options are supported for the developer to understand program evolution from different perspectives. For instance, a report details the full list of atomic changes in the convenient form of a grid, and another generates a html report of the same information. Graphing options are also available. The tool is extensible in that developers can write add-ins that can interpret the parsed information in their own way for further analysis. JEvolve is available on Windows™ platforms.

7. EMPIRICAL DATA

JEvolve was run on Java Developers Kit source code of various versions (only classes and interfaces in the primary ***java*** package were compared). Table 2 below summarizes the key characteristics of various version changes. Versions of JDK compared are JDK 1.0 Beta Vs JDK 1.0.2 (shown in the table as **V1**), JDK 1.1.6 Vs JDK 1.1.8 (shown as **V2**), JDK 1.1.8 Vs JDK 1.2 (shown as **V3**).

Table 2. Summary of Version Evolution.

Description	**V1**	**V2**	**V3**
Total no. of atomic changes	246	420	3988
Classes modified	49	120	680
Classes added	1	22	546
Classes deleted	0	1	153
Classes that require retest	99	100	779
Methods that require retest	407	606	8149
Modified methods that do not require retest	8	30	61

The above table shows that 61 of all the methods modified in V3 require no retesting. Table 3 gives a partial list of the atomic changes across the evolved versions. Notice that a trivial change of reordering class members occurs 99 times in V3.

Table 3. Atomic Changes Across Versions.

Atomic change	**V1**	**V2**	**V3**
Accessibility level of field changed	0	2	19
Method made synchronized	1	8	30

Method made non-synchronized	1	33	18
Method argument renamed	2	2	44
Method made final	3	0	8
Method made non-final	0	0	31
Class made final	1	0	0
Class made non-final	0	0	1
Class made public	0	0	3
Accessibility level of method changed	1	2	26
Class members reordered	1	3	99
Field made final	0	1	16
Instance variable made transient	0	1	11
Instance variable made non-transient	0	0	3
Field direct initializer modified	1	10	34
Static initializer block added	0	2	47
Static initializer block removed	1	1	4
Static method added	0	7	210
Static method removed	0	3	37

8. CONCLUSION

It is possible to characterize an evolving Java program in terms of ***atomic changes***. An analysis of such a set of atomic changes provides interesting clues about which parts of a program might require additional or regression testing. Since ***unguided*** testing can consume considerable resources, deriving useful clues about what portions of a modified program need to be retested, greatly assists in achieving ***focussed*** testing. A tool called JEvolve has been developed on Windows platform to perform automated analysis of Java programs by applying the language-based approach described in this paper. Some of the results obtained by analyzing JDK versions have been presented.

REFERENCES

1. Rangarajan,K. and A.Balasubramaniam, "When are Two Classes Equivalent?", ACM SIGPLAN Notices, V.33(2) February 1998, pp. 59-64.

2. Rangarajan,K. and P.Eswar, "Understanding Class Evolution Through Atomic Changes", ACM SIGPLAN Notices, V.33(6) June 1998, pp. 48-52.
3. Kung, D., J.Gao, P.Hsia, F.Wen, Y.Toyoshima, and C.Chen, "Change Impact Identification in Object Oriented Software Maintenance", Proc. IEEE International Conference on Software Maintenance, 1994, pp. 202 – 211.
4. Pei Hsia, Xiaolin Li, D.Kung, C.Hsu, L.Li, Y.Toyoshima, and C.Chen, " A Technique for the Selective Revalidation of OO Software", Journal of Software Maintenance, Vol. 9, 1997, pp.217-233.
5. Gregg Rothermel and Mary Jean Harrold, "Selecting Regression Tests for Object-Oriented Software", Proc. IEEE International Conference on Software Maintenance, 1994, pp. 14-25.
6. Scott A. Whitmire, Object-Oriented Design Measurement, John Wiley & Sons, 1997.
7. Andrew J. Palay, C++ in a Changing Environment, Proceedings of the 1992 Usenix C++ Conference, 1992, pp.195 - 206.