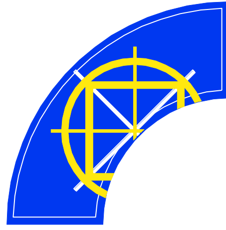


TEKNISKA HÖGSKOLAN

HÖGSKOLAN I JÖNKÖPING

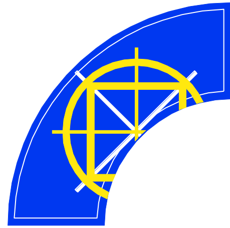
SEMANTIC DIFFERENCES AND GRAPHICAL VIEW OF FILES

Bandi Raghavender

Mohammed Rafiullah Khan

MASTER THESIS 2009

COMPUTER ENGINEERING



TEKNISKA HÖGSKOLAN

HÖGSKOLAN I JÖNKÖPING

SEMANTIC DIFFERENCES AND GRAPHICAL VIEW OF FILES

Bandi Raghavender

Mohammed Rafiullah Khan

This master thesis is carried out at the School of Engineering of Jönköping University within the subject area Computer Engineering. The work is part of the university's two-year master's degree. The authors are responsible for the given opinions, conclusions and results.

Supervisor: Thomas Gustafsson
Examiner: Vladimir Tarasov

Credit points: 20 points (D-level)

Date: 7/04/2009

Archive number:

Postal Address: Box 1026 551 11 Jönköping

Visiting Address: Gjuterigatan 5

Telephone: 036-10 10 00

ABSTRACT

This Master's thesis presents an algorithm that finds the semantic differences between two versions of files, an older version and a new modified version of the file. The algorithm is responsible for finding changes in the program's behavior and displaying them graphically. By this a lot of time can be saved because it is not necessary to go through the whole file to find the differences.

The program, Semantic Diff, developed in this master thesis uses the Javacc parser generator which is used to parse files and generate the abstract syntax tree for them. Using this tree it is possible to see all the methods, classes, constructors and parameters for both older version and modified version. By comparing all the methods, classes and interfaces of both the versions it is possible to find the differences that change the program behavior.

The algorithm for finding semantic differences has been evaluated by testing it on various test cases. By making changes in the original file and in the modified file. Like adding methods and deleting methods and adding classes to the files. The algorithm highlights those methods with green color which are added newly in the modified file and highlights the methods with red color in the original file which got deleted in modified file. This algorithm also finds the textual difference between two files and highlights those lines which are changed in modified file and which got deleted from original file.

Table of Contents

ABSTRACT.. 3

1 INTRODUCTION 6

2 LITERATURE REVIEW.. 7

2.1 SEMANTIC.. 7

2.2 CURRENT STATE OF FINDING DIFFERENCES BETWEEN FILES 8

2.2.1 LIMITATIONS OF SEMANTIC DIFFERENCE.. 8

2.3 USING SEMANTIC DIFFERENCE TO REDUCE COST OF REGRESSION TESTING 9

2.4 JDIFF: A DIFFERENCING TECHNIQUE AND TOOL FOR OBJECT - ORIENTED PROGRAMS.. 10

2.5 THE EFFECTS OF MODIFICATION 11

2.6 ABASTRACT SYNTAX TREE.. 12

2.7 UNIX DIFF... 13

3 PROBLEM FORMULATION 17

3.1 NEED FOR SEMANTIC DIFFERENCE AND GRAPHICAL VISUALIZATION................. 17

3.2 TOOLS AVAILABLE 18

4 PURPOSE AND GOAL OF THESIS... 19

5 PROGRAMMING LANGUAGES.. 19

5.1 JAVA... 19

5.2 JAVA PACKAGES 19

5.3 JAVACC 20

5.4 JAVA SWINGS 22

6 IMPLEMENTATION... 23

6.1 DIFFERENCING ALGORITHM.. 23

6.2 GENERATING TREE 25

6.3 TEXTUAL DIFFERENCE.. 27

7 FUNCTIONALITY TESTS 28

8 EVALUATION 33

8.1 VALIDATION... 35

9 RELATED WORK 36

10 CONCLUSION 38
10.1 FUTURE WORK 39
REFERENCES 40

1 INTRODUCTION

Maintaining software often involves analysis of two versions of a program meaning that an older version is compared to a newer version. It is not an easy task to understand the changes done in a new version of a program and one way to find the difference is running the new program in large number of test cases. Even a small change done in a new program may result in a big change in the program's output.

A number of tools exists (e.g., the UNIX diff utility [10]) to find and highlight the textual difference between two files. However these tools are just useful to find the textual difference between two files. These tools are not useful to understand the semantic differences between two versions of a program which are responsible for changing the program behavior.
Consider for example two versions of a file being part of a program, version - *P* which is the original and *P1* which is the modified version. Figure1 shows the two versions.

Program P	*program p1*
Public class A{	public class A{
Void m1( ){…..}	Void m1( ){…..}
}	}
Public class B extends A{	Public class B extends A{
	Void m1( ) {….}
Void m2( ) {….}	Void m2( ) {….}
}	}
Public class E1 extends Exception {}	Public class E1 extends Exception {}
Public class E2 extends E1 {}	Public class E2 extends E1 {}
Public class E3 extends E2 {}	Public class E3 extends E1 {}
Public class D {}	Public class D {}
Void m3(A a) {	Void m3(A a) {
a.m1( );	a.m1( );
try {	try {

```
    throw new E3( );                         throw new E3( );
}                                        }
Catch (E2 e)  {…..}                      Catch (E2 e)  {…..}
Catch (E1 e)  {…..}                      Catch (E1 e)  {…..}
        }                                        }
}                                        }
```

Figure 1: textual difference between two programs

In figure 1, textual differences between two version of program P and P1 are viewed graphically. However, as mentioned above, such differences cannot find out the statements which are responsible for changing the program's behavior.

To overcome such problems an algorithm is needed which compares two versions of a program and highlights the statements which are responsible for changing program behavior. By this it is possible to go to the statements directly which have been changed in the new version.

This Master thesis presents a program that can highlight both textual differences and semantic differences. The program, which is described in more detail in Chapter 5.1, takes two files as input and can find either textual or semantic differences. When semantic difference option is selected the program highlights the lines of code with appropriate colors which are responsible for changing the behavior of the program. The program does not concentrate on the syntactical or textual difference when a semantic difference is performed.

2 LITERATURE REVIEW

2.1 SEMANTIC

Semantic is defined as follows by [1].

"Relation between signs and the things they refer to, their denotata"

Semantics, it is the study of the structure of sign systems which focus on the form, not on the meaning. The term semantics can apply not only to the natural languages but also to technical languages, such as a computer programming language. In computer language it is considered as a part of application of mathematic language [1]. Semantics reflects the meaning of programs or functions.

For the following statements use different syntaxes (languages), but result in the same semantic [1]:

x += y; (C, Java, etc)

Let x = x + y;

x = x + y (various Basics)

The above three statements will perform an arithmetic operation addition y to x.

2.2 CURRENT STATE OF FINDING DIFFERENCES BETWEEN FILES

According to Michael L. Collard [2] there are two main kinds of differences one is textual difference and the other one is semantic difference. Two files are said to be textual difference if they have different character or different word at a given point in two files. But semantic difference is an approach that doesn’t concentrate on character or on syntax but on the meaning or behavior. Two files can be said semantically different if they have got different behavior and it takes AST (Abstract Syntax Tree [22]) view of the source code [2]. Two statements are said to be semantically different if they have different meaning [2].

2.2.1 LIMITATIONS OF SEMANTIC DIFFERENCE

Semantic difference is concerned only with changes in programming level that affect the behavior of the program. Semantic difference ignores lexical, syntactical and documentary changes and concentrates only on the changes that affect the behavior and execution of a program [2]. These semantic changes can be anywhere in program or all over the program a small change in one file of program causes semantic changes across entire program, e.g., the change of the type in a variable declaration changes the type of expressions that use the variable [2]. Because the changes are program wide the differencing must be applied to the entire program.

The limitation of semantic differencing is the abstraction level [2]. Programmers often use the abstractions of a programming language to hide type and other information, e.g., *typedef*, *macros* in the C/C++ languages. A textual change to program in a type or to an inheritance hierarchy is not a change that should be seen in entire program and these kinds of changes does not support analysis tasks. While semantic difference supports analysis tasks, especially for the calculation of metrics, it is often the wrong approach to differencing from a programmer's perspective [2].

2.3 USING SEMANTIC DIFFERENCE TO REDUCE COST OF REGRESSION TESTING

David Binkley says that **"**Software-maintenance tasks often involve analyses of two versions of a program an *original* version and a *modified* version [3]". David Binkley presented an algorithm for reducing cost of regression testing by reducing the size of the program that must be rerun. This algorithm partitions the components of the new program into two sets: "preserved points—components that have unchanged run-time behavior; and affected points—components that have changed run-time behavior [3]". Only a test case that affects the behavior of the program must be re-run. David Binkley's algorithm produces a program difference, which captures the behavior of the affected points. By this we can run our test cases on the affected points rather than running it on all test cases.

David Binkley's algorithm compares to programs modified and certified and produces a third program differences. There exist some programs for finding textual difference (e.g., the UNIX diff utility, Myers 1986 [10]) which are used to find textual difference. However, these textual differences are not sufficient to find the semantic difference or test cases that must be retested. For example, consider the following changes for which determining textual differences alone are insufficient for finding semantic difference [3]:

(1) "A textual change that does not produce a change in behavior".
(2) "A textual change in one location of a program that causes a change in behavior "down stream" in either the same procedure, a called procedure, or a calling procedure".

2.4 JDIFF: A DIFFERENCING TECHNIQUE AND TOOL FOR OBJECT - ORIENTED PROGRAMS

When developing software, information about the changes between two versions of a program older and modified is useful for number of software engineering tasks [4]. For example in regression testing, information about which parts of a program are changed can help us in identifying test cases that must be rerun [3]. Many of these tasks like regression test may not provide enough information for performing semantic difference effectively. We can find these kind of problem especially in the object oriented programs. Apiwattanapong presents a technique for solving the above problem and Apiwattanapong also present Jiff tool that implements the technique for java programs. For example, the two partial Java programs in fig. 1 the original program P and the modified program P1. If we run test cases on P and P1, and observe the output we will see that method S. m1 has been added to P1 and that the exception-type hierarchy has changed in P1. Without any additional information about these changes it would be difficult to find that in P1 the class A has one more additional method void m2 ( ). And has different exception type.

In order to overcome the problems like this and provide the differencing information (between two versions of program) required for tasks such as program-profile estimation, impact analysis, and regression testing [4] Apiwattanapong defined a new kind of graph representation and a differencing algorithm. This differencing algorithm uses the new graph representation to identify the changes at statement level between two versions of program. The representation of augments is a traditional control-flow graph (CFG *control-flow graph* is a directed graph in which nodes represent statements and edges represent flow of control between statements) to model behaviors caused by object-oriented features in the program [4]. By using this new kind of graph representation Apiwattanapong indentifies the changes done in the modified program which are responsible for changing the program behavior.

Apiwattanapong algorithm extends an existing differencing algorithm [5] and this algorithm consists of five steps. First, it matches classes, interfaces, and methods in the two versions of program. Second, it builds enhanced CFGs (Control Flow Graphs) for all matched methods in the original and modified versions of the program. Third, it reduces all graphs to a series of

nodes and single-entry, single-exit sub graphs called hammocks. Fourth, it compares, for each method in the original version and the corresponding method in the modified version, the reduced graphs, to identify corresponding hammocks. Finally, it recursively expands and compares the corresponding hammocks.

The main contributions of JDiff according to Apiwattanapong are:

- “A new graph representation that models the behavior of object-oriented programs [4]”.
- “A differencing algorithm that works on the graph representations and uses different heuristics to increase the precision of the results [4]”.
- “A tool, JDiff, which implements this differencing algorithm for Java programs [4]”.
- “A set of empirical studies, performed on two real, medium-sized programs that show the efficiency and precision of our algorithm [4]”.

2.5 THE EFFECTS OF MODIFICATION

Daniel Jackson describes a tool that takes two versions of a procedure and generates a report. This report summarizes the semantic difference between the two procedures. Daniel Jackson tool is not like any other tool which dependence on program dependence graph but it generates the report in terms of the observable input-output behavior of the procedure and not on the syntactical structure of the procedure. As this analysis is truly semantic this does not require any prior matching of syntactic components.

The SDG (System Dependence Graph) extends previous dependence graph representation which collects procedure calls rather than monolithic program [3]. “SDG model a flat language like ‘C’ with the following properties [3]”.

- “A complete system consists of single main procedure and a collection of auxiliary procedures”
- “Auxiliary procedure ends with return statement, the main procedure ends with end statement”
- “Parameters are passed by value - result.”

A SDG is a collection of procedural dependence graph (PDG) connected by inter procedural control and flow dependence edges [3]. PDG are similar to program dependence graph which are used to represent the program in vectorizing and parallelizing compilers [6] [7].

According to Daniel Jackson [15] a maintainer faces two principal problems when modifying some part of code: first, what does the existing code does and second one understanding the effects of a changes on the code. Most of the tools are build for first problem so Daniel Jackson has build a tool that help overcome the second problem. According to Daniel Jackson a programmer can be benefit from this tool. By running the tool after making the changes, he can correlate the tools summery against his intent. It is very difficult to understand a code with heavy changes than understanding a fresh code. This is because the maintainers often fail to change the document after the changes have made. Daniel Jackson tool can solve this problem to much extent because this tool can provide the structure of the document for a change in code.

2.6 ABASTRACT SYNTAX TREE

"In computer science an Abstract syntax tree or just syntax tree is a tree representation of some source code" that has been written in programming language [22]. An Abstract syntax tree (AST) is a finite, labeled, directed tree. Each construct that is present in the code is denoted by a node in Abstract Syntax tree. This tree is called abstract syntax tree since this tree may not show all the constructs present in the source code. *Example*: omission of grouping parentheses. Each interior node in AST represents a construct and children of programming language of that node. In AST, internal nodes are operators and the leaf node represents the operands of the operator, these leaf nodes are variables or constants [22].

An abstract syntax tree generates a tree structure by taking all the essential structure of the code and leaving the unnecessary syntactical details like grouping parenthesis, comas and comments in the code [8]. An AST can be distinguished from normal concrete tree because it omits tree nodes to represent punctuation marks such as semi-colons, commas and tree nodes that represent unary production in the grammar [8]. These Abstract syntax trees are generally created bottom up.

While generating the Abstract Syntax Tree nodes, "a common design choice is determining the granularity of representation of the AST [8]". That is weather all the constructs of the source code should be represent as different nodes of AST or only some constructs of the source code should be represented as AST nodes and differentiate using their values. one of the examples representing granularity is how to represent arithmetic binary operations. One way of representing is to have single binary operation tree node in which one of its attributes the operation e.g." +". And the other way is have a tree node for each binary operator. Then in object-oriented language this would result in creating classes like AddBinary, SubstractBinary, MultiplyBinary, etc with an abstract super class Binary [8]. There are some systems which will generate AST automatically from a small language such as Zephyr ASDL [9] or by integrated specification with a parser generator, Eli [11]. One of the useful futures of implementing AST is to have an instantiation of the builder pattern [17] with a function that will take strings to ASTs. Which are useful in generating test cases before grammar is completed or new language features are being experimented with.

2.7 UNIX DIFF

Diff is one of the oldest commands in UNIX and it was included in UNIX around 1976. This Diff was used to find textual difference between two textual files source file and targeted file. Diff compares these two files and generates delta "delta is lines that are changed or got deleted in either of files [18]". This was written by Hunt and McIlroy and based on algorithm for file comparison that they created (see J. W. Hunt and M. D. McIlroy, *An algorithm for differential **file comparison***, Bell Telephone Laboratories CSTR #41 (1976)). While it is the first but still it is the best.

For UNIX Diff both the files are textual files as the textual comparison is an illusive concept [18]. So a simplest approach is considered that is line indivisible and compute so called "longest common subsequence of lines" (lcs) [18]. And anything that is not in this lcs is considered as difference set. A difference set is the "minimal set of lines that needs to be changed for the transformation of source to be targeted" [18].

Using Diff it is easy to find the word based difference and symbol based difference but the problem is finding string based diff this is problem that is studied in all algorithms. Even word based difference can be found using Diff by changing small changes to the basic program. That is we first need to convert text in "one word per line" and then use line-based Diff.

Diff is always useful for us in some many cases it is useful for finding the textual difference between two textual files by this we can keep track of the document and we can easily find the differences between two files. the diff is not only useful for comparison of two files but even it can be used in program code for instance if there is a problem in the newer version and not in older version we can diff code browser for difference set and try to pinpoint the source of the problem.

Some of the diff comparison control option:

b — Causes blanks (spaces and tabs) to compare equally even if an unequal number of blanks exist.

-I — cause the case of letters to be ignored.

-w — causes all white spaces (blanks and tabs) to be ignored.

Directory comparison option:

-l — display long output listing.

-r — recursively descends through subdirectories.

-s — display files that are the same. Normally, identical files are not displayed.

Example:

Here we can see two files original and new file. There are some changes done to the new file by adding some more details and making changes in spelling. We can compare this two files using UNIX Diff through command line using command

Diff original new

The output of this command shows changes that need to be done to the original file to match the new file [19].

Original file	New
This part of the document has stayed the same from version to version. It shouldn't be shown if it doesn't change. Otherwise, that would not be helping to compress the size of the changes. This paragraph contains text that is outdated. It will be deleted in the near future. It is important to spell check this dokument. On the other hand, a misspelled word isn't the end of the world. Nothing in the rest of this paragraph needs to be changed. Things can be added after it.	This is an important notice! It should therefore be located at the beginning of this document! This part of the document has stayed the same from version to version. It shouldn't be shown if it doesn't change. Otherwise, that would not be helping to compress anything. It is important to spell check this document. On the other hand, a misspelled word isn't the end of the world. Nothing in the rest of this paragraph needs to be changed. Things can be added after it. This paragraph contains important new additions to this document.

The diff command (diff original new) produce the following out put

```
0a1,6
> This is an important
> notice! It should
> therefore be located at
> the beginning of this
> document!
>
8,14c14
< compress the size of the
< changes.
<
< This paragraph contains
< text that is outdated.
< It will be deleted in the
< near future.
---
> compress anything.
17c17
< check this dokument. On
---
> check this document. On
24a25,28
>
> This paragraph contains
> important new additions
> to this document.
```

- **a** stands for added, **d** for deleted and **c** for changed.
- Line numbers of the original file appear before a, b and c and after for new file.
- Angel brackets appear before line that are added, deleted or changed.
- By default lines which are common are not shown.

In the output we see the following lines

0a1, 6

> This is an important

> notice! It should

> therefore be located at

> the beginning of this

> document!

>

The first line 0a1, 6 mean that the lines from one to six are newly added to new document. And the angle which is towards right side shows that these lines belong to new file.

17c17

< check this dokument. On

> check this document. On

Here we can clearly see that the document spelling is changed from dokument to document.

'<' this angel mean that this line belong to original file and

'>' this angel mean that this line belongs to new file

3 PROBLEM FORMULATION

3.1 NEED FOR SEMANTIC DIFFERENCE AND GRAPHICAL VISUALIZATION

When developing software, information about the changes between two versions of a program older and modified is useful for number of software engineering tasks [4]. For example in

regression testing, information about which parts of a program are changed can help us in identifying test cases that must be rerun. Many of these tasks like regression test may not provide enough information for performing semantic difference effectively. We can find these kind of problem especially in the object oriented programs.

Finding semantic difference is not only useful in finding test cases that must be rerun. But it is also useful in soft ware maintenance. For developing software takes less effort than maintaining software. Maintenance often involves knowing the difference between two versions of program knowing the textual and syntactical difference are useful but knowledge about semantic difference between two versions of program is much more useful than syntactic or textual difference. By knowing semantic difference we can easily point out the lines of code in a program which are responsible for changing the program behavior. There are some tools present to find these differences.

3.2 TOOLS AVAILABLE

Concurrent Version Control: CVS is a version control system, an important component of Source Configuration Management (SCM). Using it, you can record the history of sources files, and documents.

JDiff: JDiff is a tool for finding differences between versions of object-oriented programs. The technique is based on a representation that handles object-oriented features and thus can capture the behavior of object-oriented programs. Apiwattanapong presents a technique for solving the above problem and Apiwattanapong also present Jiff tool that implements the technique for java programs.

UNIX Diff: Diff is one of the oldest commands in UNIX and it was included in UNIX around 1976. This Diff was used to find textual difference between two textual files source file and targeted file. Diff compares these two files and generates delta “is lines that are changed or got deleted in either of files [18]”. For UNIX Diff both the files are textual files.

4 PURPOSE AND GOAL OF THESIS

The purpose of this thesis is to find the semantic differences between two versions of a program and display the output graphically. It means that there are two parts in the thesis. The first part is devising a way to finding semantic differences between two versions of a program and the second part is to find a way to show the resulted differences between two programs. In order to perform above mentioned two tasks, a tool that will find the semantic difference and displays the result graphically has been constructed. Additionally, the goal is also to display the output i.e., the semantic difference in such a way that the user can easily and correctly understand the changes made to the original program.

5 PROGRAMMING LANGUAGES

In this section the programming languages that are used in this thesis for developing algorithm are discussed and how they are implemented in the program. The main goal of this thesis is to find out the semantic difference between two versions of a program and display them graphically.

5.1 JAVA

Java is a general purpose object oriented programming language developed by Sun Microsystems of USA. The most striking feature of the language is that it is platform neutral language. Programs developed in java language can be executed anywhere on any system.

5.2 JAVA PACKAGES

Packages in java are a variety of classes or interfaces grouped together. By organizing class together we get the following benefits.

- The classes in a package of other program can be easily reused.
- Classes in different package can have the same name.
- Packages provide us means to hide classes and thus preventing other programs from accessing classes which are meant for internal use only.

Java API (application programming interface) provides a large number of classes grouped into packages according to the functionality. Most of the time we use packages that are available in java API.

5.3 JAVACC

Javacc (java compiler compiler) Javacc is a parser/scanner generator for java [12]. Javacc is an open source parser generator for java language. A parser generator is a tool that reads a grammar specification and converts it into a java program that can read files matching the grammar. In addition to this parser Javacc provides other capability related to parser generator such as tree building through a tool called JJTree.

"Javacc is a parser generator and lexical analyzer generator [20]". These parser and lexical analyzer are software components which deal with input of character sequence. Lexical analyser breaks the input sequence of characters into a subsequence called tokens and it also classifies the tokens.

Consider this Small example in 'C' [20].

```
Int main ( )
{
Return 0;
}
```

The lexical analyser of C breaks this into sequence of tokens.

"int" , " " , "main" , " (",")" ,
" " , " {" , " \n " , " \t " , "return"
" " , " 0 " , " " , " ; " , " \n " ,
" } " , " \n " , ""   .

The lexical analyser even understands the token like '**;**' for the above program in C the token might be like this [20].

KWINT, SPACE, ID, OPAR, CPAR,

SPACE, OBRACE, SPACE, SPACE, KWRETURN,
SPACE, OCTALCONST, SPACE, SEMICOLON, SPACE,
CBRACE, SPACE, EOF

The EOF token represents end of the original file. These tokens are passed to the parser and parser analyses the sequence of tokens and determines the program structure. Often the parser outputs a tree structure representing the program [20]. This tree structure is then input to the components of compiler which are responsible for code generation.

This Javacc parser generator are used in programs to parse it and generate an abstract syntax tree for codes for which semantic differences are to be found via a package called JJTree. The JJTree package is used to generate abstract syntax trees for the two versions of the source code. JJTree is pre processor for JavaCC [21]. The output of JJTree is run through JavaCC to create the parser.

Because of the default behavior of JJTree it constructs a parse tree node for each non terminal language [21]. We can change this behavior if we want to so that some non terminals do not have nodes. JJTree defines some interfaces that all parse tree nodes must implement. This Interface provides some methods for operation like setting node for parents and setting node for children and retrieving them. JJTree operates in two modes simple and multi if you don’t provide any implementation for node classes then JJTree will generate simple implementation based on SimpleNode.

JavaCC is a top-down approach but JJTree construct parse tree from bottom up [21]. The JJTree pushes all the nodes into stack when they are created and when it finds the parent node it pops up all the child nodes from stack and adds them to parent and finally pushes the parent node into stack. The stack is always open that means you can push, pop or manipulate the contents present is stack.

5.4 JAVA SWINGS

To create a Java program with a graphical user interface (GUI), we need java Swing. The swing toolkit includes a set of components for building GUIs (Graphical User Interface) to Java application. Swing toolkit includes all the components that are useful for building GUI it consists of table controls, list controls, tree controls, buttons, and labels.

Swing is part of the Java Foundation Class (JFC). The following list shows some of the features of swings.

Swing GUI Components

The swing tool kit consists of array of components like buttons, check box, radio box, tables and labels which are all basic components swing even provides you drag and drop facility.

Java 2D API

If you want to make your application stand out, convey information visually, or to add images, figures or animation to your GUI you need Java 2D API.

Data Transfer

As said before swing provides us with facility drag and drop so that you can transfer data via cut, copy, paste and even drag and drop.

Undo Framework API

One of the best features of swing is that it allows us to undo and redo. Undo support is built in to Swing's text component. Swing supports us to perform these actions for unlimited number of times. That means you can easily add or remove elements from our table. This is very useful because if we want to do any changes we need not start from the beginning again and we can save lot of time.

6 IMPLEMENTATION

In this section, the implementation of an algorithm for finding textual and an algorithm for finding semantic differences are described and evaluated. The described program is divided into three modules. First, and the main thing, is to generate the Abstract Syntax Tree (AST) for the code. By generating the tree it is possible to find all program constructs. The second thing is to get various constructs from the AST and display them. The third task is to do the comparison between these constructs of two programs and display them graphically. The third step is a manual step performed by the user of the tool.

6.1 DIFFERENCING ALGORITHM

Overview:

The algorithm for finding semantic differences between two versions of a file is called SemDiff. It takes as input the original file and the modified file and produces as output a set of nodes with properties, e.g., modified or unchanged.

SemDiff performs its comparison first at class and interface level and then at method level, and finally at the node level. The algorithm first compares the classes of p (original program) with the classes of p1 (modified program) and Interfaces of p and p1. Then for each pair of class's c and c1, it compares the methods m and m1. Finally, method block comparison is done.

Algorithm: SemDiff

Input: original program *P*

modified program *P'*

Declare: nodeBase n and n1.

Begin: SemDiff.

1: Read all the classes in P and Store in C.

2: Read all the Classes in P' and Store in C'.

3: *compare* C and C'.

4: if classes in C and C' are equal

5: Read all the methods of P and P' and Store them in m and m'.

6*: compare* m and m'.

7: Read all the children of m and m', if equal

8: compare children of m and m'.

9: **end** SemDiff

Procedure: Boolean compare()

```
1.    For(I = 0; I < no. of nodes  in P; I++)
2.       Read node and store at c[i];
3.       For(j = 0; j < no. of nodes in P'; j++)
4.           Read node  and store at c[j];
5.           If(c[i].equalsc[j])
6.           Return true;
7.           End for
8.      End for
9.    End Match();
```

Class and Interface level comparison:

SemDiff begins its comparison at class and interface level (line 1-2). The algorithm reads all the classes declared in p and p1 and store them to c and c1, then it compare each class in c with c1. In order to compare, it take one class c and compares it with all the classes of c1. If the match is not found, it reads all the methods of those classes and if the match is not found it means that particular class in c is deleted in c'. Similarly, each class c1 of p1 is compared with all classes c of p, if match is not found which means that it is newly added class in c'.

In general, class declarations include the following:

1. Modifiers such as public, private.
2. The class name.
3. The name of class parent, i.e., super class if this class is extended.
4. A comma separated list of interfaces implemented, preceded by the keyword *implements.*
5. The class body

Method Level Comparison:

In this level, SemDiff compares, for each pair of classes c and c1 or interfaces I and I1, their methods. It reads all methods defined in class pair c and c1 or interface pair I or I1, and store them to m and m1. Like the approach used to compare classes and interfaces. Then it compares each method m with all methods m1, if the match is found it reads all the children of m and m1 and match not found means that particular method is deleted from m. Similarly, each method m1 is compared with all methods of m and if a match is not found, that a new method is added to m1.

In the method comparison, SemDiff compares the methods that are declared in a class. The method declaration includes:

1. Modifiers like public, private.
2. Method name
3. A comma separated list of parameters.
4. Method body.

Node Level Comparison:

After method level comparison, SemDiff performs a node level comparison. Here, a node refers to a building block of a method such as variables, loop statements and conditional statements. In this level of comparison, SemDiff reads all variables of a method pair m and m1 and store in v and v1. Similarly, conditional and loop statements are stored in some variables. The comparison is done same as method comparison or class comparison.

6.2 GENERATING TREE

We have used Javacc parser for parsing the input files. Javacc as mentioned above in 4.3 is a parser-generator written in Java, and JJTree is an add-on which allows the generated parsers to produce syntax trees. They are both freely downloadable, with documentation, from Javacc home page.

In order to pursue our goal, we needed to find a way to write parsers in Java, and we were lucky enough to come across Javacc. We also needed to build abstract syntax trees that we could walk to do type-checking and code-generation, hence we decided to use JJTree for this.

Diagram below shows the structure of JJTree.

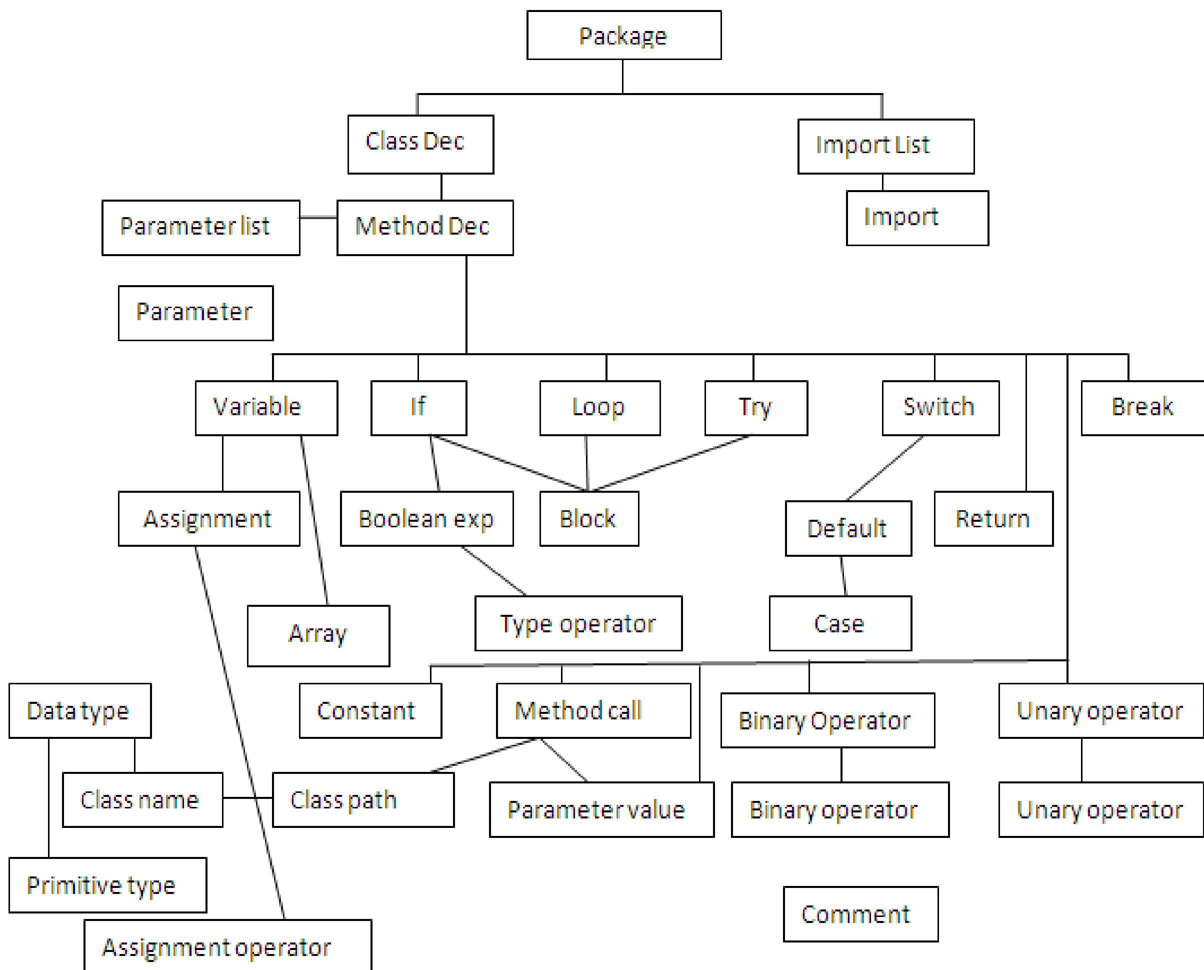


Fig2. Structure of tree

Tree implements simple Node interface. In java program information stored in packages are classes and interfaces defined in it and list of import statements. Hence, package declaration comes at root level and its children are class Declaration and import List. A class in java contains methods, constructors and data types, in order to perform some operation. Therefore, children of class declaration are method Declaration.

General syntax of method is return type, modifier string, method name, parameter list, variable declaration and method block. Method has three nodes parameter list contains list of parameters defined in a method. In variable node it stores information of all variables declared in method. Third child node of method declaration is a block.

The application logic of a method is defined in a block. To make the method functional we use various programming concepts such as conditional statements, looping etc in a method block. The child nodes of a block are shown in above figure.

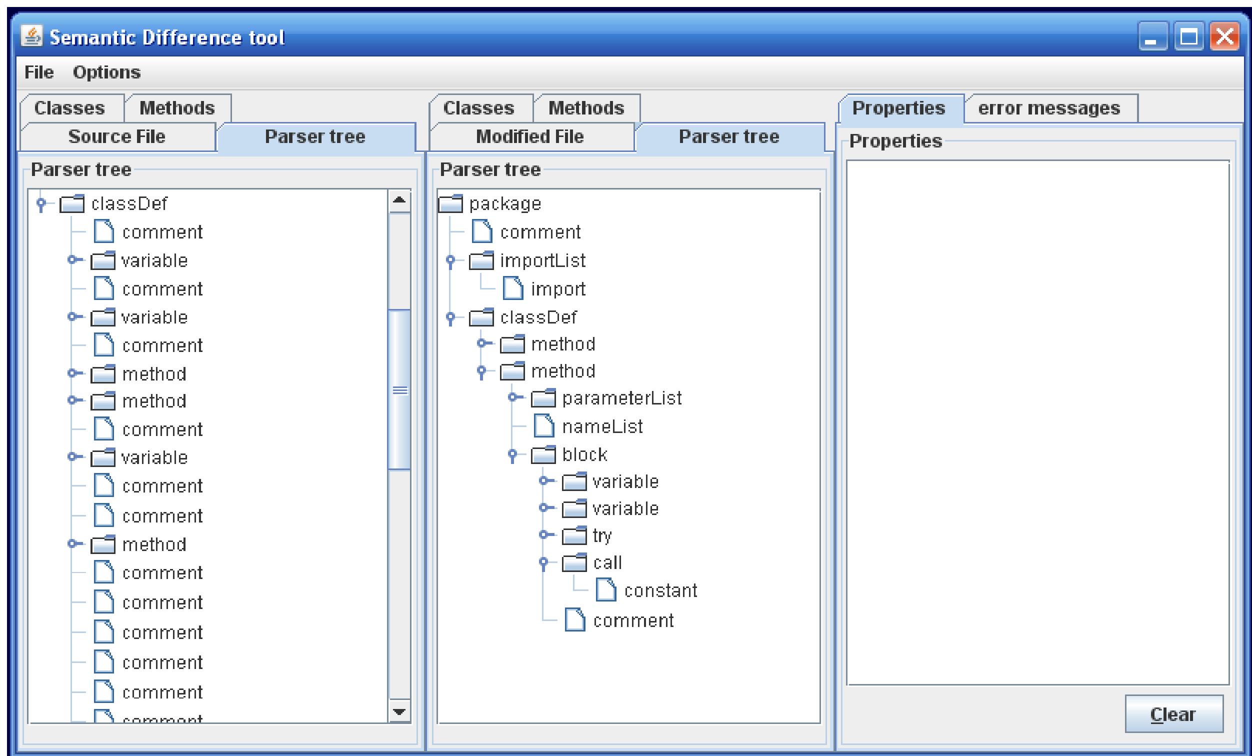


Fig3: generated tree structure for the input code

6.3 TEXTUAL DIFFERENCE

The algorithm used to find the textual difference is similar to the one which is used to find the differences between the methods.

One more feature of this tool is to find the textual difference between two codes although this is not the part of the semantic difference but we can find the extra lines of code which is added to the modified program. When you select the textual difference option it highlights the

line in green color which is added to the new code highlights the lines in red color in the original code which are deleted in modified code.

Figure below shows the results after textual difference.

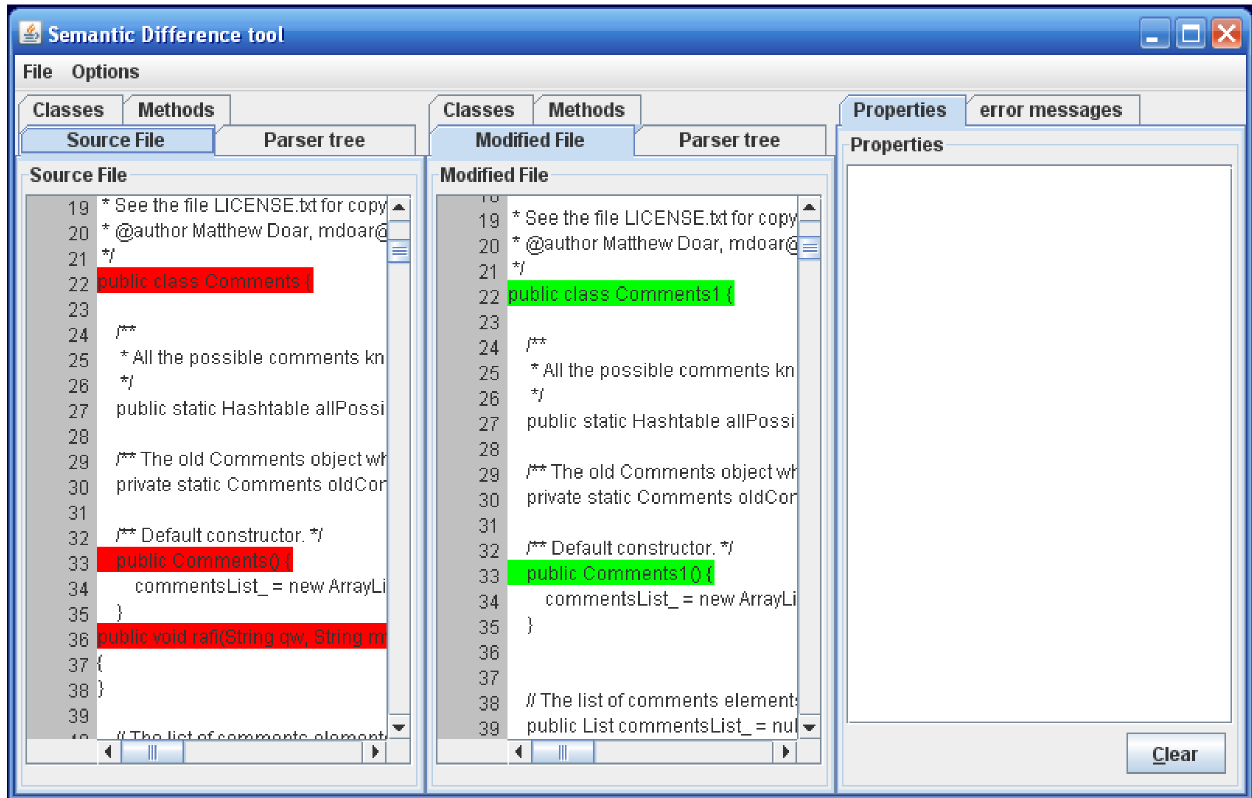


Fig: Highlighting unmatched lines in code.

7 FUNCTIONALITY TESTS

The tool consists of three main components (i) finding semantic difference, (ii) tree generator and (iii) finding textual difference. The Semantic differencing tool inputs the original and modified version of program. It generates AST for both the versions of a program and displays the trees for them. Matches all the nodes of P and P1 and highlights the particular line in a program referred to the node for which the match is not found. To test the functionality of the tool, it was run on three different test cases which cover all the functions and features of the tool and the result was as expected.

CASE 1:

In this case we made changes to the original program in such a way that it should not affect the object oriented features of the program. To achieve this, we made changes in comments. Since, the changes were not semantic therefore no difference were highlighted as shown in the figure 6.1. When we run the same file for textual difference to verify whether the changes made, it highlighted the changes made in comments as shown in figure 6.2. By this we understand that the algorithm shows only semantic differences if any.

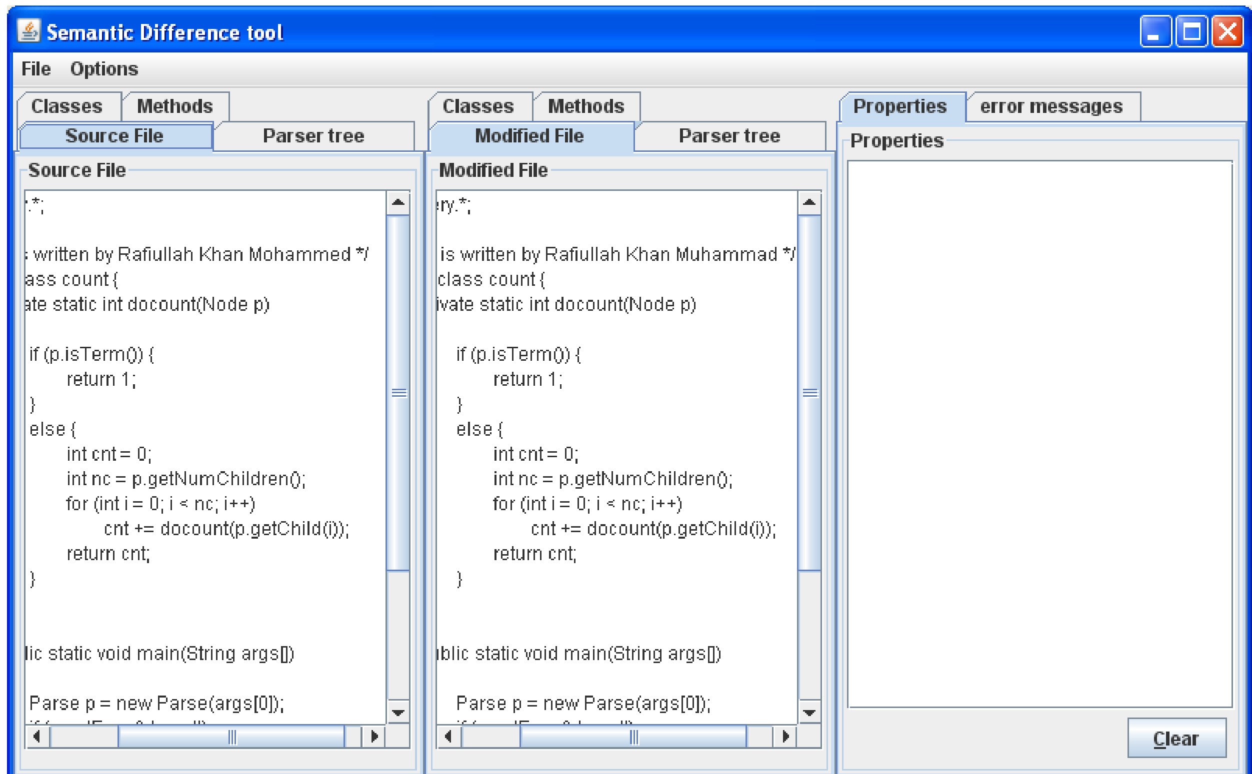


Fig 6.1: Semantic Difference

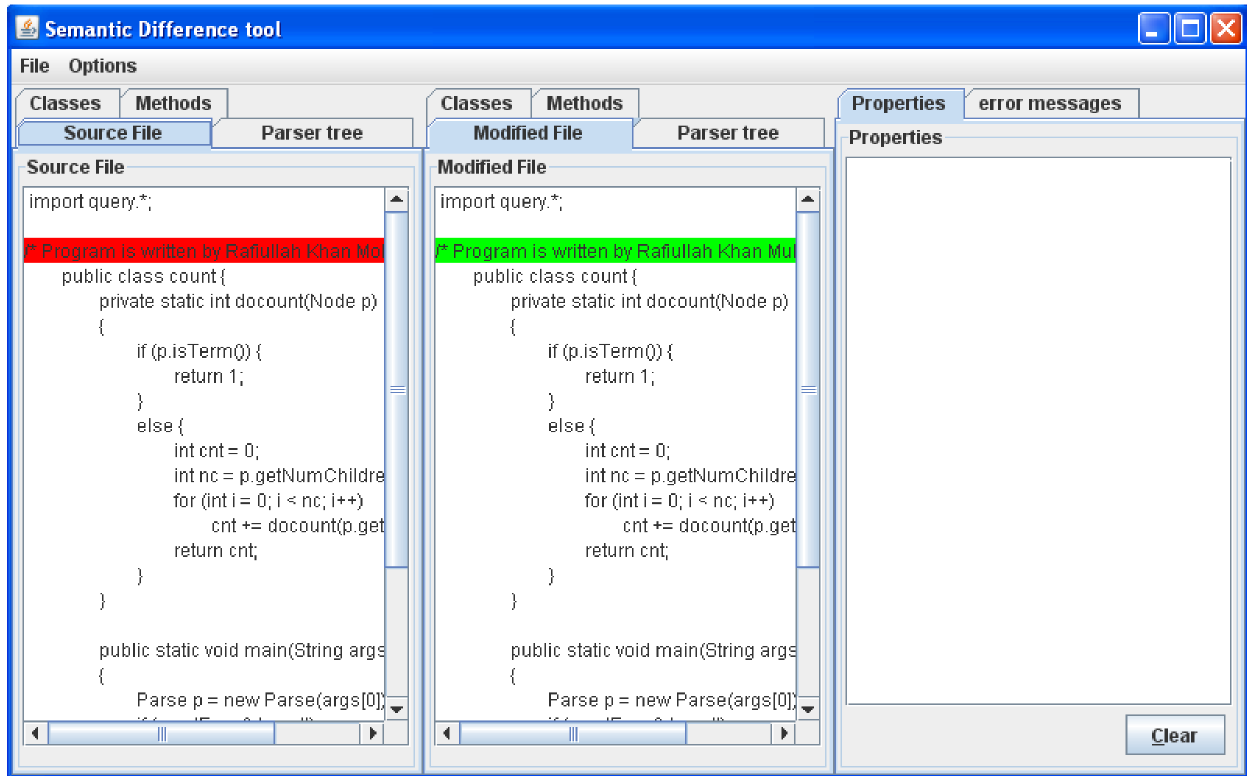


Fig 6.2: Textual Difference

CASE 2:

In this case, we made some semantic changes to the modified file by adding new methods and when we perform semantic difference it highlighted the newly added method. To differentiate we highlight the changes with green color which indicates that this part of code is added to the modified program as shown in the figure 6.3 below.

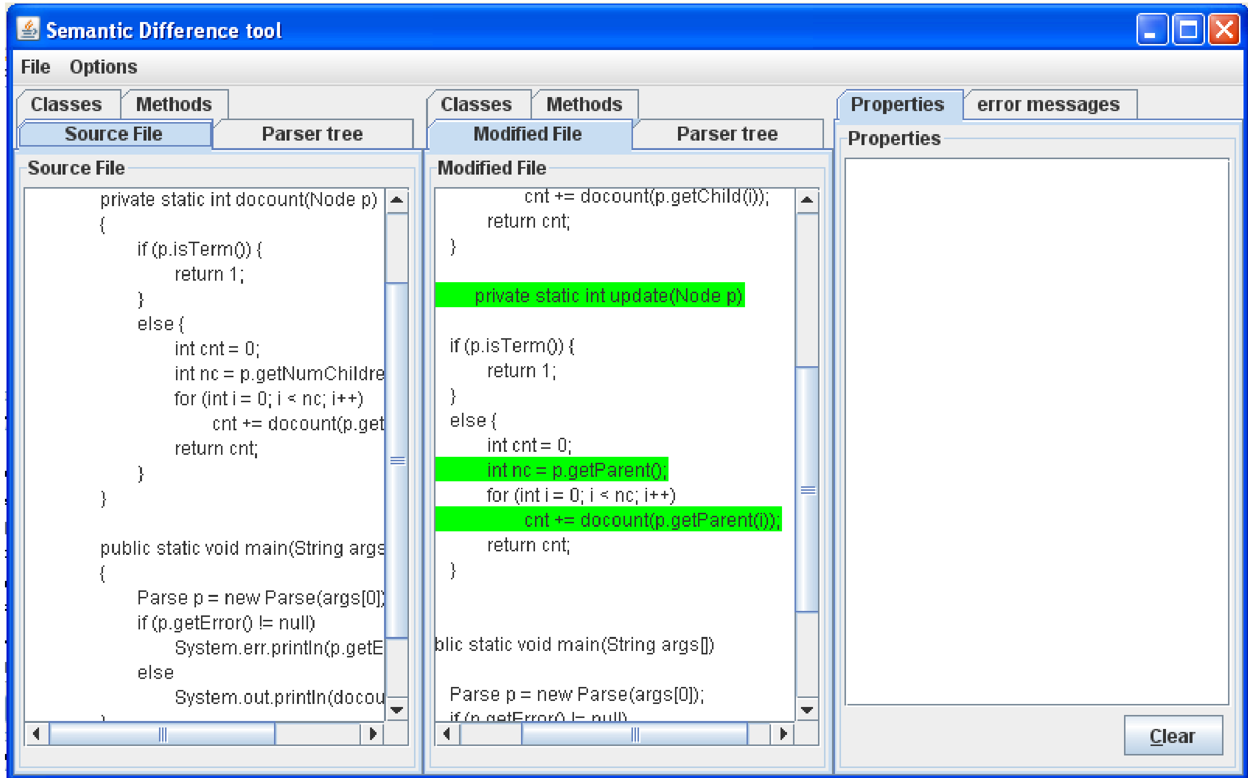


Fig 6.3: Semantic difference with added methods

Also, we have run our program on two files with some methods deleted in original file and some new methods added to modified file. When we run semantic difference methods which were deleted in original file were highlighted with red color and methods which were newly added were highlighted with green color as shown in the figure 6.4 below. Part of the code highlighted with red color indicates that this part has been deleted in the modified file. Using various colors to highlight the difference reduces the ambiguity to understand the changes done.

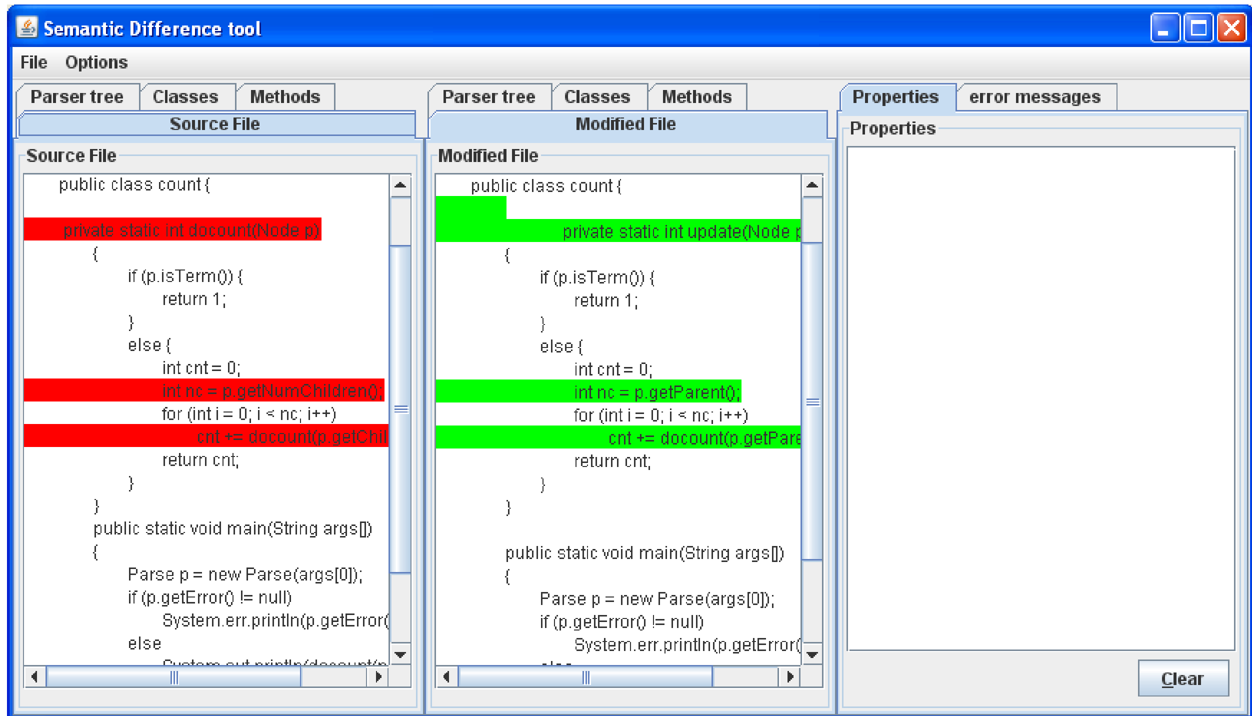


Fig 6.4: Semantic Difference after methods added and deleted

CASE 3:

While uploading the program in the tool it first parses the program to check for grammar. If there is any error in program code then it displays an error message in error messages tab as shown below.

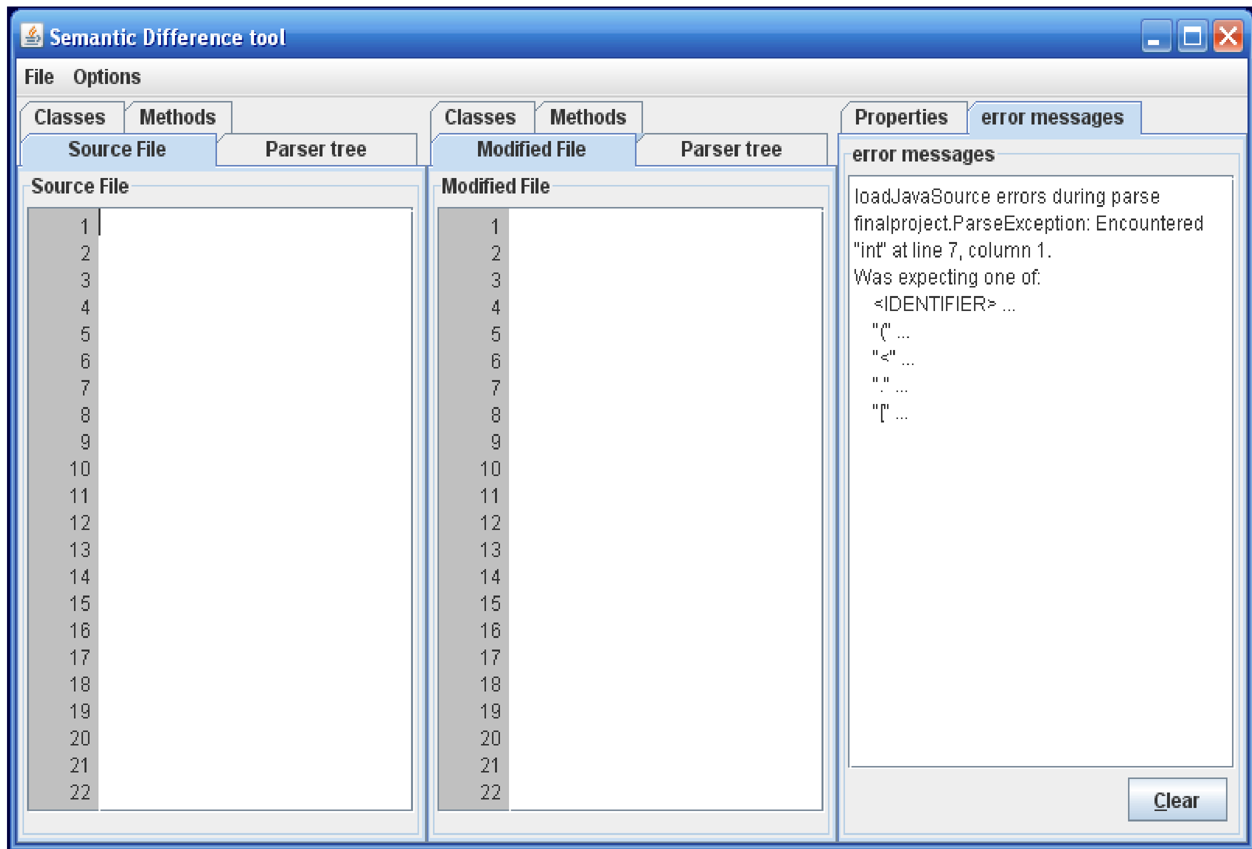


Fig5: Displaying error message.

8 EVALUATION

The aim of this study is that to evaluate the effectiveness of the algorithm compared to UNIX diff and JDiff.

As mentioned above in literature review UNIX diff is a traditional tool for comparing textual difference between two files and displaying the changes and the files are text files. JDiff is tool which is used to find the semantic difference between two versions of file.

We have taken two files one original and other modified file. In the modified file we made some spelling changes and added some new text in modified file. To find the difference between two text files using UNIX we need to run using DOS and the result will be shown like this

0a1,6

> This is an important

> notice! It should

> therefore be located at

```
> the beginning of this
> document!
17c17
< check this dokument. On
---
> check this document. On
```

The first line 0a1, 6 mean that the lines from one to six are newly added to new document. And the angle which is towards right side shows that these lines belong to new file. And the “dokument” spelling is changed to “document”.

When we run the same file in SemDiff the newly added lines got highlighted by green color and lines which are changed are highlighted by blue color.

When compared UNIX with SemDiff, SemDiff is a GUI application and UNIX diff is a DOS application. Our tool is easier than UNIX diff to use we can see the changes in two files there itself and we can edit even. But where as in UNIX diff it generates a report and we need to go to the files and make the appropriate changes. Our tool is time saving when compared to UNIX diff.

JDiff a differentiation tool for semantic difference is used as a reference for the study. One of the main advantages of using the above mentioned technique is that, it shows the changes that are affected due to object oriented features. When compared to JDiff technique and above technique both are used to find out the semantic difference caused in object oriented programming.

One of the main drawbacks of technique mentioned is that it can be performed only on single file. JDiff builds ECF’s graph (Enhanced Control Flow Graphs) while we build an AST (Abstract Syntax Tree) for program which shows the nodes and children of those nodes. JDiff can compare files faster than our tool because, it first compare classes, interface, methods or nodes of two versions and if a match found it adds them to a single stack. Where as in our algorithm it first reads all the classes, interfaces, methods or nodes of two versions and then stores them in two different text areas. Then it does the comparison and if any match is found it performs further actions.

This algorithm can run on small and medium object oriented programs where as JDiff can run on medium and large programs. To compare the algorithm with JDiff we have taken two pairs of small java programs *v1, v2* and *v2, v3* at the first run we made changes in method names and class names but no changes in variables. And watched the results the SemDiff has generated tree structure for both the files and displayed all the method names of both files and highlighted those method names which are changed by this we can say that those methods are different from original file. But whereas JDiff compares two versions and generate a table structure where it says changes in dynamic binding and changes in types

	Binding		type	
Pair	Ac	IA	AC	IA
v1, v2	13	43	17	34
v2, v3	__	__	19	45

JDIff shows two kind of object oriented changes one dynamic binding and other type changes. Form above table we can say that in version *v1* and *v2* there are 13 actual changes and 43 indirectly affected and 17actual local variable and field changes and 34 indirectly affected. And in second case we made changes in variables of methods and ran on SemDiff it displayed all the method names of both files but showed no differences in methods but still we can find the difference in variables manually by looking into tree structure this is one of the drawback of our tool that it can't show the changes done in local variables and fields and changes due to indirectly affected. Whereas JDiff says number of actual changes and indirect changes, but can't show the exact method and field in file where the change has done this is one point where our tool is better by showing the methods and classes where the changes have made leaving field and local variables.

8.1 VALIDATION

There are many threats to validity of our study. One of the external threats is that we have conducted or test using only two subject programs so we cannot claim generality of our result and the programs which were used o test were not real programs. But changes are real. One

more external threat is that we have conducted our test on only two pairs. Different test pairs may produce different results.

One of the threats for internal validation is there may be some error in our algorithm implementation and tool measurement so in order to overcome this problem we have done our test on some known examples and performed some tests.

9 RELATED WORK

There are many existing tools and technique for computing the difference between two versions of file. All these techniques vary in one or the other way some are used to find textual difference while other are used to find syntactic or semantic differences. These techniques vary in terms of operation, algorithm and application.

UNIX diff utility [10] is one of the most general tools for finding the textual difference as discussed in the literature review. Diff compares two files line by line and shows the difference in the files but these differences are purely textual so it does not identify the changes that are responsible for changing the program's behavior. This tool is not suitable for finding the semantic difference but SemDiff tool does not has such limitation because this technique is suitable for finding semantic difference of object oriented programming language.

Maletic and Collard's (Maletic and Collard's, 2004) approach is based on transforming C and C++ language into format called srcML [13]. (SrcML is an XML based format that represents the source code annotated with syntactic information). This approach generates srcML for both old and new version of the source code. After comparison it creates a new XML document with the additional XML tags that indicate the common, inserted and deleted XML elements. However it still has a limitation that it uses line based differencing information obtained from Diff.

Eclipse [14] (the Eclipse foundation, 2001) one of the modern IDE (integrated development environments) incorporates a parser for the programming language they support. Therefore

Eclipse can compare the differences between two versions of a program more effectively than other techniques which are based on textual comparison. However Eclipse has limitation that it can be used only to find the syntactical differences.

Semantic diff by Jackson and Ladd compares two version of program procedure by procedure [15]. Their algorithm Computes a set of input-output dependencies for each procedure and computes the difference between two sets for the same procedure in old and new version of program. That means it can apply to only sets that are common and misses the sets which are new in modified version that does not affect input-output dependencies.

Horwitz's approach finds both the syntactic and semantic difference between two programs using a Program Representation Graph (PRG) [16]. However this approach can be applied only to the languages written in C, C++, or Java because the PGR representation is defined only for programs written in languages with scalar variables, assignment statements, conditional statement, while loops and output statement.

Binkley's approach is used to find the semantic difference between two versions of a program [3]. This technique first generates a System Dependence Graph and unmatched nodes and nodes with different incoming data or control flow of two version of a program are taken into consideration. Then it performs forward slicing on all such affected nodes in the graph. The main aim of this approach is to reduce cost regression testing.

JDiff is build to find the semantic difference for two versions of program written in Java [4]. This technique uses calcDiff approach where it first gets all the set of classes first then set of interfaces, set of methods, and set of nodes of two version of program. First it matches classes, methods and interfaces in the two versions then it builds enhanced CFGs (control flow graph) for all matched methods in the original and modified versions of the program then it reduces all the graphs into single entry and single exit sub graph called hammocks. Finally it recursively compares the corresponding hammocks.

10 CONCLUSION

In this paper we are submitting an algorithm for comparing two java programs. To implement we have generated a tool called “Semantic Difference Tool”. Given two programs this tool parses, builds abstract syntax tree then finds the semantic as well as textual difference and displays them graphically. This algorithm can be used for various maintenance tasks. The main advantage of using this algorithm is that we can save lot of time when compared to other tools which are used to find textual difference and syntactical difference. Our technique is defined for java programming language.

This tool was tested on three test cases and we got positive results. Case one says that if there are no semantic changes in the programs then no differences are shown. In case two we made some semantic changes and the changes were shown visually. As mentioned above, the tool parses the program before uploading it and if there is any error it will show the error message. Therefore, to test this feature we made some grammar mistakes in the programs and upon loading the program to the tool it shown the error message as expected.

One of the oldest tools which are used to find the difference between two files is UNIX Diff which was included in UNIX 1976. This Diff is used to find the textual difference between two files but only finding textual difference is not useful for many soft ware maintenance tasks. The tool implemented in thesis can be useful to reach this maintenance tasks. This tool not only finds out the textual difference between two versions of program but also finds out the semantic difference between them which are responsible for changing the program behavior. This tool doesn’t generate any report in the output but it graphically shows the difference between two files where the changes have made. By this a programmer can directly go to the lines of code which are changed textually or lines of code which are responsible for changing the program behavior.

Based on the experience with the tool, this technique is 100% successful in finding the object oriented-oriented semantic difference and textual differences between two versions of a program and displaying them visually. We think this tool is very useful in providing the

information about semantic changes in program. Even though this program provides useful information but it cannot be directly used for software maintenance tasks.

10.1 FUTURE WORK

This tool finds the semantic differences between two versions of a program. However, it would be more beneficial if we find the semantic differences between more than two versions of a program. During software evolution a single program may evolve into many versions, if we find the semantic differences between all the versions it will be more beneficial. Therefore, in future this tool can be extended to find the semantic differences between many versions at a single run. By doing this we can save time, cost and moreover, the maintenance of the software during evolution becomes easier.

It is also useful if we find to what extent changes has been done to the modified file. One way to do this, we can display the extent of changes in percentage. For example, extent is 50%, it indicates that half of the program is changed. This can be done by showing the number of changes in the program. If a graph of the differences is generated, we can capture the information from various angels that means more information can be gathered and it reduces the ambiguity.

REFERENCES

[1] Semantics - Wikipedia, the free encyclopedia.mht

[2] Michael L. Collard August, 2004 meta-differencing: an infrastructure for source code difference analysis.

[3] David Binkley Loyola College in Maryland Using Semantic Differencing to Reduce the Cost of Regression Testing

[4] Taweesup Apiwattanapong · Alessandro Orso · Mary Jean Harrold, JDiff: A differencing technique and tool for object-oriented programs

[5] Laski, J., Szermer,W.: Identification of program modifications and its applications in software maintenance. In: Proceedings of IEEE Conference on Software Maintenance, pp. 282–290 (1992)

[6] Kuck, D.J.Kuhn, R.H.Leasure and wolfe, “Dependence graph and complier optimization,” pp. 207-218 in *conference record of the Eighth ACM symposium on principles of programming Languages.*

[7] Ferrante, J., Ottenstein, K., and Warren, j., “the program dependence graph and its use in optimization,” *ACM transactions on programming languages and systems (July 1987)*

[8] Joel Jones *Abstract Syntax Tree Implementation Idioms,* Department of Computer Science, University of Alabama, jones@cs.ua.edu

[9] Daniel C. Wang, Andrew W. Appel, Jeff L. Korn, and Christopher S. Serra. The Zephyr abstract syntax description language. In USENIX, editor, *Proceedings of the Conference on*

Domain-Specific Languages, October 15–17, 1997, Santa Barbara, California, pages ??–??, Berkeley, CA, USA, 1997. USENIX.

[10] Myers, E.W.: An O (ND) Difference algorithm and its variations. Algorithmica **1**(2), 251–266 (1986)

[11] Robert W. Gray, Vincent P. Heuring, Steven P. Levi, Anthony M. Sloane, and William M.Waite. Eli: A complete, flexible compiler construction system. *Communicationsof the ACM*, 35(2):121–130, February 1992.

[12] Java.net the source for java technology collaboration, https://javacc.dev.java.net/

[13] Maletic, J.I., Collard, M.L. Supporting source code difference analysis. In: Proceedings of the 20th IEEE International Conference on Software Maintenance (ICSM 2004), pp. 210–219 (2004)

[14] The Eclipse Foundation. Http: //www.eclipse.org (2001)

[15] Jackson, D., Ladd, D.A.: Semantic diff:Atool for summarizing the effects of modifications. In: Proceedings of the Internation Conference on Software Maintenance, pp. 243–252 (1994)

[16] Horwitz, S.: Identifying the semantic and textual differences between two versions of a program. In: Proceedings of the ACM SIGPLAN'90 Conference on Programming Language Design and Implementation, pp. 234–246 (1990)

[17] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns*. Addison-Wesley Publishing Company, Inc., Reading, Massachusetts, 1994. ISBN 0-201-63361-2.

[18] diff- softpanorama open source software education society

[19] diff- Wikipedia, the free encyclopedia.mht

[20] 2003 sun Microsystems, javacc tutorial 4.0, introduction to Javacc

[21] JavaCC [tm]: *JJTree Reference Documentation*

[22] Abstract Syntax Tree - Wikipedia, the free encyclopedia.mht