INF9290 Project Report: Testing Eclipse Compare's Algorithm for Three-way Merging

Martin Fagereng Johansen

June 4, 2010

1 Introduction

Scope of project In my project for INF9290, I have tested a part of an Eclipse plugin. The plug-in is the Eclipse Compare Framework[1] which is a part of the Eclipse Platform[2]. The concrete parts I tested are the classes for supporting a three-way merge. This functionality is described in section 3. This is the system under test, and I refer to it as "the SUT" where a short term is appropriate.

Goals of project The goal of the project is to identify and implement a test suite that has a good combination of size, execution time and fault-detection capabilities. Good fault detection capabilities is achieved by finding a good testing strategy with a coverage criteria that is met by the test suite and which, optimally, allows for the specification of a good oracle.

Structure of the report This report is structured as suggested in the template for INF4290 (INF9290) projects: First, I describe some background information about the theory behind the project. Then I describe the design of the case study, followed by an analysis of its results. The document ends with a discussion of lessons learned, and the appendix includes information that might be interesting as an additional source of detailed information about some of the topics discussed.

2 Background

For testing the SUT, I have tried both white-box and black-box testing techniques. For the black-box technique, I constructed an oracle. For reducing the running-time of the black box test suite, I applied a random testing technique.

The source code of the SUT is distributed as several Eclipse projects. I tested version 3.5.2 of the SUT on Eclipse of the same version. The tests are implemented as plug-in tests using the JUnit framework version 3 to invoke the test execution.

2.1 Description of methodologies, techniques and tools

2.1.1 White-box testing - Unit testing

I used a white-box testing technique in the beginning to get to know the system. There were a number of tests already implemented for the SUT. These tests had low statement-coverage. Being a minimal coverage criteria, that is quite bad. I ran the original tests using a coverage analyzer[3], and then I implemented a handful of tests to increase the statement-coverage. I had to use EclEmma because CodeCover[4] did not support software spanning multiple Eclipse projects.

The unit tests uses JUnit 3 for execution. The code metrics were found using the metrics tool for Eclipse[5].

[1] `http://eclipse.org/eclipse/platform-team/`
[2] `http://www.eclipse.org/platform/`
[3] EclEmma: `http://www.eclemma.org/`
[4] `http://codecover.org/`
[5] `http://metrics.sourceforge.net/`

2.1.2 Black-box testing - Equivalence class testing

The testing strategy that I ended up choosing is a black-box testing technique. The reason this technique is proper for the SUT is that the SUT has a clearly defined task that is accessed through a simple interface. There are basically three inputs, each is a list of atoms from some alphabet. The output is the merge result as chunks and chunk-types (explained later) as well as the merge if there were no conflicts. The technique I used was equivalent class testing, which is a well known technique. I identified equivalence classes over the input after having studied the SUT in depth. Hence, it is the functional and well-defined nature of the SUT which makes equivalence class testing proper for the SUT. This is explained in detail later.

2.1.3 Oracles

I managed to create and oracle for the equivalence class testing. An oracle is a system that can provide information that is used in the evaluation of the results from the SUT.

2.1.4 Random testing

The execution of the black-box test suite took a long time. I employed random testing to reduce the time spent testing the complete black-box test suite. Random testing is well known in testing theory.

2.2 Relevant published works

In preparation for the project, and in addition to the lectures, I read "Software Testing - A Craftman's approach" [1]. This book provided me with many techniques that I could consider when deciding which testing strategy to use for testing the SUT.

For a formal understanding of the algorithm in the SUT, I read "A formal investigation of Diff3" [2]. Diff3 is another implementation of three-way merging.

For information about the Eclipse platform plug-in architecture, I read "Eclipse Platform Technical Overview" [5].

I consider these to be the most relevant works given the level and scope of this report.

2.3 Possibly relevant testing techniques not used

White-box testing and mutation test-suite testing Major faults in the SUT was found to be related to errors in algorithm design and in the specification rather than in the actual implementation. This makes white-box testing and mutation testing infeasible, as both are based on source code. This is further elaborated in later sections where applicable.

Class-testing Some class testing uses white-box techniques, and hence they were not applicable as discussed above. Using class testing as black box testing, such things as inheritance is not too important for the SUT. It uses inheritance mainly as a way of implementation and not of reuse. Hence, when testing the SUT as a black-box, one covers the classes as well as their super-classes without needing any special theory to ensure full coverage. Also, the future retesting of new sub-classes was not relevant. Hence, I did not find class testing relevant for the SUT.

Robustness testing Robustness testing would be interesting for the SUT, but due to time constraints, and the presence of interesting faults found with the other testing, I decided not to do robustness testing.

Decision tables, Logic functions and Category-Partition After reading the paper "A Formal Investigation of Diff3" [2], I saw that there was a possibility to test the SUT using equivalent class testing. One might get good results using these three techniques from the header, but the reason I did not study their applicability was the applicability of equivalence class testing and its fruitful results.

```
H: [ CA-,     S,   DB-,     S]
A: [1004, 1001, 1002, 1003]
O: [1000, 1001, 1002, 1003]
B: [1000, 1001, null, 1003]
O':[1004, 1001, null, 1003]
```

Figure 1: An example comparison and merge diagram

3 Design of the Case Study

3.1 Background on the SUT

3.1.1 Three-way merge in general

An ubiquitous problem in software development is parallel editing of the same file. Two or more people make changes to a single file. The common way to organize this is to have, for example, source code in a repository. Those who work on the source make a copy to their local workstations. Each person edits the file and then submits the changes back to the repository. A problem occurs when someone tries to submit their changes to a file that has been changed by someone else in the meantime. It is then the responsibility of the repository control system to merge the two changes and to request user-input for resolving conflicting changes. If there are no conflicting changes, the merge is automatic.

3.1.2 A detailed look at three-way merge

The following formal look at the three-way merge algorithm is inspired by and based on the formal investigation of Diff3 by Khanna et al. 2007 [2].

A three-way merge is a functional task. It takes three files as input, and produces a comparison and, if there are no conflicts, a merged file. We denote the file F. The original version of this file is denoted O, then the two changed versions are denoted A and B.

Eclipse Compare implements a generic merge. Hence, we regard a file as a list of atoms. For example, a text file can be seen as a list of text lines. In that case each text line is the atom. The atoms must support comparison with each other, an evaluation of whether two atoms are equal or not. An atom is denoted $Atoms = \{a_1, a_2, a_3, ...\}$. In this document a list of atoms is denoted $[1000, 1001, 1002]$ (instead of $[a_{1000}, a_{1001}, a_{1002}]$) where each entry in the list is the index of an atom a, where the a-symbol is omitted for readability. The empty atom is denoted *null*. The reason for starting at 1000 is that each atom is denoted by four characters, including the empty atom (null).

I would like now to introduce a comparison diagram. The contents of the three files, A, O and B are listed under each other. The original file contained $[1000, 1001, 1002, 1003]$. A decided to change the first line to 1004 making $[1004, 1001, 1002, 1003]$, while B decided to delete 1002 making $[1000, 1001, 1003]$. The tree way comparator arranged the contents of the files as in figure 1. Here we can see that the comparator has aligned the atoms that are the same in each file, 1001 and 1003. The atoms that were not all the same are then arranged in between with nulls if necessary. Each column in this diagram is called a *chunk*.

The chunks can be of different types according to what the contents of the three files are at that column. In figure 1 the first column is classified as CA- which means non-conflicting change of a. The second and fourth column is classified as S, meaning "the same". The third column is classified as DB-, which means non-conflicting delete in B. The dash means no conflict. Since there are no conflicts we can merge the tree files into one, denoted O', according to rules for each chunk-type. The result is O':[1004, 1001, 1003].

Obviously, there is a limited number of possible chunk-types. Each chunk can, for A, O and B, be one of four atoms denoted a, b, c or null. Hence, the number of chunk-types is ${}_4P_3 = \frac{4!}{(4-3)!} = 4! = 24$. Many of these classifications are essentially the same (for example, the columns "aaa" and "bbb" are both the same type of chunk, namely a matching one). There are 14 unique chunk types, and these are listed in table 1. The type represented by S is called a *stable chunk*. All the other chunk-types are called *unstable chuks* or *hunks*. The term hunks is found in the source code for the SUT. The chunks that includes conflicts are called *conflicting chunks*.

We are now in a position to define the function of a three-way merge. A three-way compare *compare* is a function of three files, O, A and B, resulting in a list of chunks. A three-way merge *merge* is a

Type-code	Description	How to resolve
S	the same	O or A or B
IA-	insert into A	A
IB-	insert into B	B
IG-	the same insert into both	A or B
CA-	change of A	A
CB-	change of B	B
CG-	the same change of both	A or B
DA-	delete A	null
DB-	delete B	null
DG-	same deletion of both	null
IGX	conflicting inserts	user-prompt
CAXDBX	change in A, delete in B	user-prompt
CBXDAX	change in B, delete in A	user-prompt
CGX	conflicting changes in both	user-prompt

Table 1: Chunk types and how to resolve them

function on three files, O, A and B, resulting in a list of atoms, O'. All possible resolutions of the chunks H is denoted O' where $|O'| = 1$ if there are no conflicts, and in general $|O'| = 3^n$ if there are n conflicts.

$$compare(O : Atom[], A : Atom[], B : Atom[]) : H[]$$

$$merge(O : Atom[], A : Atom[], B : Atom[]) : Atom[] = O'$$

3.2 A concrete look at the software

"Eclipse is an open source community, whose projects are focused on building an open development platform comprised of extensible frameworks, tools and run-times for building, deploying and managing software across the lifecycle."[6]

Eclipse has a plug-in architecture [5], and even the internals of Eclipse is build using plug-ins. On eclipse.org ten main packages are offered, and even more packages are available elsewhere on the web. One of the plug-ins that is common among almost all distributions is the Eclipse Compare Framework.[7] This plug-in contains functionality for comparing files. Of interest to me in this project is the functionality for performing a three-way merge.

The primary class is `org.eclipse.compare.rangedifferencer.RangeDifferencer` which reports the result of the three-way comparison as chunks of class `org.eclipse.compare.rangedifferencer.RangeDifference` (this is thoroughly described in section 3.1). This comparison is then easily used to perform a merge. But, the concrete merge is dependent on the type of file being compared. An implementation for text-files is found in `org.eclipse.compare.internal.merge.TextStreamMerger` at under 100 loc.

The following method performs a three way compare. O is called ancestor, A is called left and B is called right. This method is located in `org.eclipse.compare.rangedifferencer.RangeDifferencer`

```
RangeDifference[] findRanges(
  IProgressMonitor pm,
  IRangeComparator ancestor,
  IRangeComparator left,
  IRangeComparator right
)
```

The following method performs a three-way marge of text-files. O is called ancestor, A is called target and B is called other. O' is stored in output. The encoding of each text-file is given after each text-file. This method is located in `org.eclipse.compare.internal.merge.TextStreamMerger`

[6] `http://www.eclipse.org/org/`
[7] `http://eclipse.org/eclipse/platform-team/`

```
IStatus merge(
  OutputStream output, String outputEncoding,
  InputStream ancestor,String ancestorEncoding,
  InputStream target, String targetEncoding,
  InputStream other, String otherEncoding,
  IProgressMonitor monitor
)
```

The following is the distribtution of responsibilities for performing a three-way marge. These classes are the main parts of the SUT in this projects.

The following classes are in the package `org.eclipse.compare.internal.merge`.

- TextStreamMerger - 61 loc - Performs a three-way merge on three text-files
- LineComparator - 31 loc - Contains atoms for lines of text - see IRangeComparator.
- (..).DocLineComparator - 110 loc - Contains atoms for lines of text in a document.

The following classes are in the package `org.eclipse.compare.rangedifferencer`.

- RangeDifferencer - 200 loc - The three-way compare implementation independent on file-type - algorithm basically the same as Fig. 2 in [2]
- RangeDifference - 113 loc - The results of a three-way compare, chunks, is reported as an array of instances of this class, a list of chunks
- RangeComparatorLCS - 137 loc - This class extends the LCS class to compact and shift the results of LCS to the front of the file
- LCS - 255 loc - An implementation of Myers LCS algorithm [3]
- IRangeComparator - 6 loc - Interface for atoms in the three-way merge

All the code is a total of 907 loc.

3.3 Source code

3.3.1 Source Code of the SUT

The source is found at the following CVS source repository.

```
:pserver:anonymous@dev.eclipse.org:/cvsroot/eclipse
```

The packages listed below contain the plugin, the tests and some required dependencies, respectively, for the compare framework 3.5.2. It compiles on Eclipse for RCP/Plug-in Developers Galileo-SR2. Version 3.5.2 is related to the R3_5_2 version tag.

```
org.eclipse.compare/plugins/org.eclipse.compare
org.eclipse.compare/plugins/org.eclipse.compare.core

org.eclipse.compare.tests

org.eclipse.core.tests.harness
org.eclipse.core.tests.resources
org.eclipse.test.performance
org.eclipse.test.performance.(your OS-identifier)
```

3.3.2 Source Code of test suites

The source code for various implementations are found in the following files belonging to the package `org.eclipse.compare.test`.

The code is in Java v.1.4 because this is the language both the SUT and the existing test-cases are written in.

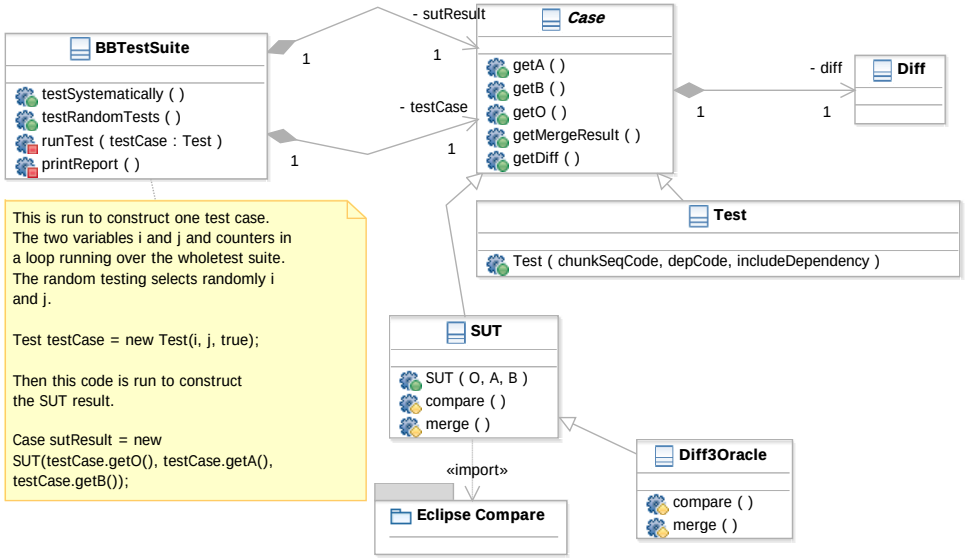

Figure 2: Class diagram with the test suite design

- BBTestSuite.java - This file contain the implementation of the black-box test suite. It also contains an implementation of random testing that selects random test cases from the black box test suite. Figure 2 shows the class diagram of the test suite. The BBTestSuite class has two subclasses of Case. One instance is of the Test - which generates a test case given a number identifying which test case to generate from the test suite identified in section 5. The SUT is initialized with the three input files from the test case. After this, the actual merge is compared to the expected merge, and a verdict is given. This test suite is executed using JUnit 3, but it is not a unit test. (Note: In order to run this test suite, the method RangeDifference(int kind, int rightStart, int rightLength, int leftStart, int leftLength, int ancestorStart, int ancestorLength)" on the class `org.eclipse.compare.rangedifferencer.RangeDifference` must be turned from protected to public.)

- Oracle.java - This file contains the implementation of an Oracle for a three-way merge. It is a slow algorithm - it is equal to or slower than $O(x^n)$, but it is sufficient as an Oracle for the black box test suite implemented in BBTestSuite.java.

- DocumentMergerTest.java - This file contains the implementation of the white-box test suite. It is only partially implemented, and achieves increased statement coverage of a larger SUT than the one I ended up with for the black-box test suite. This testing strategy is discussed in section 4. This test suite is executed using JUnit 3, and is in fact several unit tests.

- report.txt.zip - This file contains all the test cases that were valid but that failed or were inconclusive when run on the SUT.

4 Idea 1: White-box testing: Unit testing

My initial idea was to test the system using white-box testing to achieve some sort of code-coverage.

The test-suite that comes with the SUT is a set of unit tests and some performance tests. The performance tests are considered outside of this documents scope. The unit tests achieve a statement-coverage of 42% of the two main packages of Eclipse Compare. The tests execute in 8.5 seconds, and totals 215 tests over 2.800 loc. For the class DocumentMerger, the statement coverage is 46%, for the main classes for the compare algorithm, it is 77%.

Before giving up, I improved the statement coverage by adding 25 tests of a total of 477 loc. 6 of the tests are for bugs I found in the bug-database of the Eclipse compare framework.

5 Idea 2: Black-box testing: Equivalence class testing

5.1 Design of a black-box test suite

Based on the formal understanding of the three-way merge functionality, I was able to create a black box test suite based on equivalence class testing.

The first stage was to consider all possible inputs to a three-way merge $(A, O, B)*$. These inputs maps *onto* chunks $H*$. The mapping from $H*$ to $(A, O, B)*$ is surjective, making each member in $H*$ an equivalence class over the inputs. The number of chunks is still infinite, so I decided to split each list of chunks into a list of five or less chunks. This reduction looses input-cases, but I think it is a reasonable cut. This will test all possible chunk-lists where a divide and conquer into lists of five chunks is applicable. Hence, if these tests passes, there might still be faults resulting from unique chunk-lists of six or more chunks.

The first stage of my testing is to generate a list of expected chunks, H_e, of five or less chunks. When I have these chunks, I generate four files: O, A, B and the expected merge O'_e. After performing a three-way merge, the result should be the same or better than the original list of chunks, and give the same, or a better, merge result. I pass O, A an B to the SUT and read of the resulting chunks of the execution, H, and the merge, O'.

Hence, I end up with the following data: H, H_e, A, B, O, O' and O'_e. There is a lot of analysis on these results that one can do. I did some of these and report on them in the results section.

5.1.1 Example

Step 1: Generate a list of chunk-types.

```
He: [S, IG-, S, DG-]
```

Step 2: Generate one atom for each chunk-type. The column must be an instance of the chunk-type and include the expected resolution.

```
He:  [   S,  IG-,    S,  DG-]
A:   [1000, 1001, 1002, null]
O:   [1000, null, 1002, 1003]
B:   [1000, 1001, 1002, null]
O'e: [1000, 1001, 1002, null]
```

Step 3: Generate text files with one atom per line making a list of line atoms, and then pass these files to the SUT.

```
A:   "1000\n1001\n1002\n"
O:   "1000\n1002\n1003\n"
B:   "1000\n1001\n1002\n"
```

Step 4: Read the comparison and the merge results:

```
H:   [S, IG-, S, DG-]
O':  [1000, 1001, 1002]
```

Step 5: Analyze the results based on all the acquired information. in this case the comparison and the merge is the same as the expected results.

```
He:  [   S,  IG-,    S,  DG-]
A:   [1000, 1001, 1002, null]
O:   [1000, null, 1002, 1003]
B:   [1000, 1001, 1002, null]
O'e: [1000, 1001, 1002, null]
H:   [   S,  IG-,    S,  DG-]
O':  [1000, 1001, 1002, null]
```

5.1.2 Detailed analysis of the test suite

The number of chunk-lists in H_5 is $|H_5| = 14^5 = 537.824$. There are several aspects that are important for this test suite. I discovered these while designing the test-system.

Separation problem Each atom in a list must be separated somehow. In a text file, each line is an atom. Each line must be separated by a new-line symbol. The new-line symbol must be seen as an atom because it is optional to have it as the last character of a file. Hence, the last chunk of a comparison in Eclipse Compare deals with if the new-line character is there or not.

Similarity problem If two atoms of a file are the same instance, then the comparison might yield several correct comparisons, but there probably are comparisons that are better or worse. Hence, it is necessary to test that the three-way merge does not result in a less than optimal comparison where there are atoms that are equal. I decided to have two different similarities: close similarity and far similarity. These can occur in four combinations, where the close and far are present or not for each version of the file, respectively. This yields 12 additional test cases for each test case identified before: $12 * 14^5 = 12 * 14^5 = 6.453.888$.

Invalid test cases A test case is invalid if two chunks are not separable after having initiated the three files. An example is this sequence: IA-, IB-, which denotes an insertion in a and b respectively. This will be interpreted as a change in both files, where they might or might not be conflicting. Hence, when two chunks are inseparable, the test case must be considered invalid. I implemented a method that checked all the test cases and it determined that 1.893.456 of these were invalid. Hence, the number of test cases are now $6.453.888 - 1.893.456 = 4.560.432$

Consecutive unstable chunks The diff3 algorithm, as presented in [2] and as implemented in Eclipse Compare, does not recognize consecutive unstable chunks. They are put together into one unstable chunk. Hence, all the cases with consecutive chunks which are unstable are put together and reclassified as the combination of all the chunks. I checked all my tests and found that there are 4.501.296 of these in my suite. Hence there are $4.560.432 - 4.501.296 = 59.136$ test cases left.

5.1.3 Summary of the test strategy and test suite design

The test strategy consist of separating merges into equivalence classes according to expected comparison results. When taking into account important details I found 6.453.888 equivalence classes in the extended test suite and 59.136 in the normal test suite. From each of these classes I generate one test case and one expected result. These are then compared to the results given by the SUT and classified with some verdict. Statistics about the verdicts are then written in a report.

5.2 Random testing

I have implemented a random testing suite based on the original black box test suite. The idea is to select random test cases from the complete set with the idea that we can reduce the execution time for the test, and still get the same expected number of faults predicted from the random sampling. I chose uniform random selection among the 6.453.888 tests, and I expect, for some number significantly less than 6.453.888 to find a less reduced amount of fault-cases.

5.3 An alternative idea

I did consider another idea than the one presented here which is a variation on the same theme. I later discarded this idea, but I have included a short discussion of it.

The idea was to, instead of looking at the chunks, try to look at the parallel editing of the two files and try to apply knowledge from transaction-handling in databases. The problem is, however, that consecutive changes to one file does not give any meaning in isolation. A change following another change is only meaningful given knowledge of what the previous edit was. Hence, I could not separate out edits and try to merge the interleave the edits given their time stamps. The following should illustrate the idea. Here, we have two edits to two files simultaneously with a time stamp in front. If we interleave the changes, given the change, we get different results. Hence, after having implemented it and seen that it did not work, I discarded the idea and when back to the previous idea.

```
A: (1) delete character 1
   (3) change character 1

B: (2) delete character 2
   (4) change character 4

O: (1) delete character 1
   (2) delete character 2
   (3) change character 1
   (4) change character 4
```

5.4 Test suite evaluation

Both mutation testing and fault seeding is unlikely to be uninteresting because errors probably occur during specification and not during implementation.

5.4.1 Evaluation of test suite based on bugs

One way to evaluate the test suite is by finding all relevant bugs in the bug database of Eclipse and see if the test suite can find them. I found seven bugs that I considered to be relevant to the SUT out of 380 on in the bug-database. The bugs and their analysis is listed in Appendix B.

5.4.2 Evaluation using an oracle

Another way to evaluate the test suite is to run it against another implementations of three-way merge. One famous and widely used implementation is Diff3. I implemented a version of Diff3 which calculates the best three-way merge using the Diff3 algorithm and a search through alternative two-way comparisons to find the best three-way merge given knowledge of all equally good two-way merges.

The benefit of this oracle is that it was quick to implement and is easy to verify by inspection. Also, since I know what the expected results are, I can use them to test the oracle, and then use the tested tests to test the SUT.

5.5 Summary of the design of the case study

I have delimited the SUT to the algorithmic parts of the three-way merge system used by Eclipse. I have build a test suite which has the following goals stated in section 1 "... to identify and implement a test suite that has a good combination of size, execution time and fault-detection capabilities." In order to give it fault-detection capabilities I have used equivalent class testing, I have found equivalent classes and covered all the classes with one test case. In order to evaluate the fault-detection capabilities I have created an oracle that knows how to produce the right answer, but which is drastically slower than the SUT, which makes it unusable as anything but an oracle.

Here is an example run-through of how the system finds a fault and validates that the fault is valid. First we select a list of five or less chunks.

```
Test generation:
He:  [   S,  IG-,    S,  IA-]
```

The three versions of the file F: O, A and B and the expected merged version O' given the correct three-way comparison is computed from the chunks.

```
A:   [1000, 1001, 1002, 1002]
O:   [1000, null, 1002, null]
B:   [1000, 1001, 1002, null]
O'e: [1000, 1001, 1002, 1002]
```

The three files are generated.

```
Input:
A: 1000, 1001, 1002, 1002
O: 1000, 1002
B: 1000, 1001, 1002
```

Class	Initial Statement coverage	Final Statement Coverage	Result
DocumentMerger	45.6%	85.3%	No bugs uncovered

Table 2: Summary of whitebox testing

They are then run through the SUT which produces a list of chunks, a comparison and a merged version.

```
SUT output:
H:   [   S,         CGX,    S]
A:   [1000, 1001, 1002, 1002]
O:   [1000, null, null, 1002]
B:   [1000, 1001, null, 1002]
O':  [1000, ??????????, 1002]
```

The inputs are then fed to the oracle which produces the following.

```
Oracle output:
He:  [   S,  IG-,    S,  IA-]
A:   [1000, 1001, 1002, 1002]
O:   [1000, null, 1002, null]
B:   [1000, 1001, 1002, null]
O'e: [1000, 1001, 1002, 1002]
```

We can now see that the oracle managed to produce the correct comparison and merge given just the input. We know that this is the right answer since we generated the input from the output to begin with. Since we know that the oracle could do it (since we know the right answer in advance), we know that the SUT failed.

6 Analysis of the results

6.1 How are the results analyzed?

The results are analyzed with reports containing results from the test suite classified into groups. From each group of results I have given some examples with explanation. I timed the execution and these times are compared with the numbers in the reports of the execution results.

I also analyzed the coverage of the white-box test suite and evaluate the test suite using bugs found in the bug database of the SUT.

6.2 Detailed presentation of analysis results

I will first briefly discuss the attempts at a white-box test suite before moving on to the black-box test suite. The results of the white-box test suite is shown in table 2. The initial statement coverage of the test suite bundles with the SUT was 45.6%. (Remember thet "the SUT" for white box testing is different from "The SUT" in the context of my black-box test suite.) I implemented 25 tests in 477 loc that increased the statement coverage to 85.3% without finding any faults. At this point I concluded that white-box testing was fruitless. One of the reasons is that it is very hard to increase the statement coverage, let alone the other more sophisticated forms of code and data coverage. Also, faults in the SUT were probably not coding faults but rather faults in the specification. When working with code and data coverage, I get little sense of what the application is actually doing in the large picture. Thus, I moved on to constructing and running the black-box test suite.

Executing the improved test suite increased the test suite execution time by 0.3 seconds.

Table 3 shows the main results of running the normal black-box test suite on the SUT. (In the context of the black-box test suite, the SUT is the algorithmic parts of the compare framework implementing three-way compare and merge.) By covering all the equivalence classes with one test case from each, I found that the system fails in 10,33% of the cases. 3,65% are inconclusive cases, cases where the test suite

Verdict	Number	Percentage
Fail	6110	10,33%
Inconclusive	2160	3,65%
Success	50866	86,02%
Total	59136	100,00%
Invalid	6394752	
Grand total	6453888	

Table 3: Summary of SUT results - normal test suite

Bug id	found?	reason
171678	not found	not a bug in the SUT
271427 and 289697	not found	not bugs in the SUT
260532	not found	due consecutive unstable chunks: outside the scope of the merge algorithm
185099	not found	not a bug in the SUT
306314	not found	due consecutive unstable chunks: outside the scope of the merge algorithm
88830	not found	not a bug in the SUT

Table 4: Evaluation of relevant bugs

cannot determine if the run failed or not. 86,02% of the test cases succeeded. This result is surprisingly unstable for a system that has been so much used. And, indeed, I could not find any bugs in the bug database for the SUT reporting any of these faults. Relevant bugs are shown in table 4 and they are described in detail in Appendix B.

Table 5 shows the detailed report after running the black-box test suite on the SUT. The different verdicts have been grouped into a bit finer division. Notice that most of the failures and the inconclusive are for test cases with 1 or more similarities in them (except for 120 of the test cases which happen without any similarities). Hence, the SUT fails mostly because of the two-way compare done in the first part of the algorithm. The two-way compare of O and A, and O and B without caring about which two-way compares is optimal for the three-way merge of O, A and B. See [2] for an explanation of the three-way merge algorithm implemented in the SUT.

It is realistic that a text-file contains two files that are similar. For example, source code in the C-family commonly has a single line with a '}' in it, ending a class, a method, an interface or other construct. Also, these symbols are often on consecutive lines and even accumulated at the end of files. Usually these symbols are indented, but usually lines of source code are compare without taking into account new-lines. Thus, I can conclude that the failures of the SUT are indeed realistic and that they should be fixed.

One of the detailed description is "merge failed without there being conflicts". This means that the comparison resulted in a comparison without conflicts. Why then does the merge fail? This should not happen and is properly reported as an error. The following are some examples from some of the groups.

Fail: introduced conflicts If the expected chunks did not contain any conflicting chunks and the output of the SUT did, then it is a clear failure. This is a test case from the black box test suite that failed. In this case, both A and B agree on inserting 1001, and A introduced another 1002 at its end, resulting in a similarity in the file A. This should be a simple case, but the SUT reports a conflict.

A similar failing condition is when the SUT increased the amount of conflicts. This is called "increased conflicts".

Verdict	Detailed verdict	Similarities	Number
Fail	increased conflicts	2	1343
Fail	increased conflicts	1	1939
Fail	introduced conflicts	2	870
Fail	introduced conflicts	1	1838
Fail	merge failed without there being conflicts		120
Inconclusive		2	412
Inconclusive		1	1032
Inconclusive	different merge	2	12
Inconclusive	different merge	1	52
Inconclusive	same differences, but different sequence	2	16
Inconclusive	same differences, but different sequence	1	216
Inconclusive	Test case has conflicts, but result does not	2	132
Inconclusive	Test case has conflicts, but result does not	1	288
Invalid	changes are impossible to separate	0	1281012
Invalid	changes are impossible to separate	1	590844
Invalid	changes are impossible to separate	2	21600
Invalid	Two consecutive unstable chunks		4501296
Success	conflicts, equal resolutions		28798
Success	same merge		22068

Table 5: Full SUT results - normal test suite

```
He:  [   S,  IG-,    S,  IA-]
A:   [1000, 1001, 1002, 1002]
O:   [1000, null, 1002, null]
B:   [1000, 1001, 1002, null]
O'e: [1000, 1001, 1002, 1002]

H:   [S,          CGX,    S]
A:   [1000, 1001 1002, 1002]
O:   [1000, null null, 1002]
B:   [1000, 1001 null, 1002]
```

Fail: merge failed without there being conflicts: one file is empty This occurs when one or more of the three input files are empty. In this case the SUT reports a failure to merge, even through this case should be handled. Notice that the chunks the SUT reports are actually correct; thus, it is only the merge that fails.

```
He:  [ IA-,  IA-,  IA-]

A:   [1000, 1001, 1002]
O:   [null, null, null]
B:   [null, null, null]
O'e: [1000, 1001, 1002]

H:   [ IA-,  IA-,  IA-]
```

Inconclusive This is an example of a test case resulting in an inconclusive verdict. The input is a file where A has deleted the first two lines and edited the last line by changing it to the same as the third line. B has only changed the last line to something other than A. The SUT decides to match the first occurrence in A with O. This causes the other chunks to change. This comparison is not any worse than the original one, but the test suite does not have the power to determine this and hence classifies it as inconclusive.

Verdict	Number	percentage
Fail	0	0,00%
Inconclusive	1458	2,47%
Success	57678	97,53%
Total	59136	100,00%
Invalid	6394752	
Grand total	6453888	

Table 6: Summary of oracle results - normal test suite

Verdict	Number	Projected percentage	Difference
Fail	96	10,55%	2,10%
Inconclusive	30	3,30%	9,74%
Success	784	86,15%	0,16%
Total	910	100,00%	
Invalid	99090		
Grand total	100000		

Table 7: Summary of SUT results - normal test suite - Random selection of 100,000 test cases

```
He:  [ DA-,  DA-,    S,    CGX]
O:   [1000, 1001, 1002,   1003]
A:   [null, null, 1002,   1002]
B:   [1000, 1001, 1002,   1004]
O'e: [null, null, 1002,   ????]

H:   [       CA-,    S, CBXDAX]
O:   [1000  1001, 1002,   1003]
A:   [null  1002, 1002,   null]
B:   [1000  1001, 1002,   1004]
```

I verified the test suite using an oracle implementation of the SUT and verified it against the expected result of the SUT. The result, shown in table 6, was that the test suite found no faults in the oracle and that the oracle produced more successes than the SUT, meaning that it managed to produce the right answer in 97,53% of the cases as verified by the expected results. This shows that the test suite does indeed find faults in the SUT.

The full normal black-box test suite may have its execution time improved while retaining its fault detection capabilities. Instead of running all 6 million tests, I ran a selection of 2% of the cases. The result of this execution is shown in table 7. The projected amounts of failures are close to the actual number of the SUT, as seen in the difference column. Hence, this test suite is better to run when a quick check is needed.

Table 8 shows the execution time of the different versions of the black box test suite. When testing the oracle, the time spent is 100 seconds. The oracle is a slower implementation of the SUT. The SUT takes 38 seconds which is acceptable in development environments. A limit mentioned in the testing community is 10 minutes, which is a lot longer. The random test selection has similar fault detection capabilities as the full test suite. The random test suite takes less than a second to run, making it comfortable to use in a development environment.

System	Test case selection	Execution time in seconds
Oracle	Normal	100
SUT	Normal	38
SUT	Random selection	0.8
SUT	Extended	1020 (17 minutes)

Table 8: Execution time for different systems and selection techniques

Verdict	Number	Percentage	Improvement over 3.5.2
Fail	4146	7,01%	33,54%
Inconclusive	3484	5,89%	78,71%
Success	51506	87,10%	1,10%
Total	59136	100,00%	
Invalid	6394752		
Grand total	6453888		

Table 9: Summary of SUT results - version 2010-05-31 - normal test suite

Verdict	Number	Percentage	Improvement over 3.5.2
Fail	3805588	83,45%	11,41%
Inconclusive	514284	11,28%	268,99%
Success	240560	5,27%	91,96%
Total	4560432	100,00%	
Invalid	1893456		
Grand total	6453888		

Table 10: Summary of SUT results and its improvement - version 2010-05-31 - extended test suite

The SUT is version 3.5.2 of the system. This system is some months old as of writing this report. I ran the test suite on the newest version of the SUT found in the source code repository. These results are shown in table 9. For the normal test suite, the result is an improvement of 1.1% of successful test cases. There are 33.54% less failures, reducing the number to 4146.

The extended version of the test suite contains tests that treats consecutive unstable chunks of different kinds. The newest version of the SUT succeeds in 91.96% more of the cases than version 3.5.2. The improvement is shown in table 10. Notice that the SUT still fails in 83,45% of the cases. This is because the diff3 algorithm used in most implementation actually does not support consecutive unstable chunks. One particular bug was, however, fixed. (See "Bug 306314 - Three-way merge wrongly detects conflict to adjoining (not overlapping) changes" in Appendix B) This resulted in an 11.41% decrease in the number of failures of the SUT.

6.3 Summary of important results

- I tried and dismissed one white-box and one black box testing technique.
- I decided to use equivalence class testing for the SUT.
- The equivalence classes are inputs which yield any of five combinations of chunk-types with a far, close or no dependency on O, A or B.
- The coverage criteria is generating one test case for each equivalence class.
- Random testing greatly reduces the execution time, and the fault-detection capabilities remained descent compared to the reduction in time spent.
- The test suite validated that the SUT functions correctly in at least 86,02% of the cases it covers.
- The test suite found the SUT to fail in at least 10,33% of the equivalence classes. None of the faults are known in the bug-database of the SUT. The test cases that failed were established as realistic.
- The test suite was validated by an oracle implementation that managed to produce the same or better results than the ones expected. The oracle implementation is only useful for validating the test suite, as it is an exponential time algorithm.
- The newest version of the SUT improves with at least 1.1% by the normal test suite, and 91,96% by the extended test suite.

7 Lessons learned and open issues

7.1 Practical and technical difficulties

Understanding the SUT and choosing a testing strategy It takes some time to understand the subtleties of a system to test, hence the judgment about which testing strategy might fluctuate as one increases ones understanding of the SUT, to finally crystallize.

Unusable tools CodeCover (`http://codecover.org/`) did not support software spanning multiple Eclipse projects, something my SUT does. I spent some time trying to debug this which was fruitless. Once I found a code-coverage tool that worked, EclEmma (`http://www.eclemma.org/`), it only supported statement coverage, a coverage technique that is weak. My wish was for an open source tool for analyzing various code or data coverages in Eclipse, a wish once promised by CodeCover which I did not manage to find in another tool.

White-box testing is difficult in practice I learned by experience that white-box testing is difficult to use in practice on complex examples. This might be easier given tools, as a discuss in the next subsection. Achieving code and data coverage is, however, inherently difficult, as is commonly known in the testing field. The reason is that evaluating how to reach a certain point in the code, given certain conditions in this point, is a task of arbitrary and often high complexity. This manifested itself when testing the SUT in detailed analysis of code-fragments within a larger system, trying to deduce how to reach it without considering the whole application, which is many-many thousand lines of code. The reason for some code structures is to be found in complex aspects of the SUT and the architecture of the application around it.

Complete coverage and black-box testing It is simpler to find a partial coverage than a complete coverage. One might argue that even a complete coverage might avoid some detailed case. For example, if one has a function which subtracts two numbers, one may use equivalence classes with numbers that are negative-negative, negative-positive, positive-negative and positive-positive. This covers all the input, but, if there is a problem with adding certain numbers in the range 1 million to 2 million, these equivalence classes will probably not find it. Hence, when doing equivalence class testing, one have to decide on which aspects to avoid considering and consider the aspects of the system one finds relevant to test. In the case of this case study, I evaluated that sequences of five chunks with two types of similarities was sufficient. There are clearly cases outside this which this test suite will not evaluate.

Analyzing faults Even after a black-box test suite with an oracle is in place, there is a need for a lot of time-consuming analysis to categorize the faults. Faults can, for example, be far-fetched and unrealistic in practice or faults can be due to optimizations which causes failures in very unlikely cases, but drastically improves speed. Another problem is the sheer number of faults. If the number of faults is high (for this case study it was 6110), one cannot go through every test case. Hence, there is additional and time consuming work to be done after a test suite is implemented and validates for actually classifying the faults.

Evaluation of the test suite Even if the test suite report faults, there might be faults in the test suite. Validating the test-suite increases the confidence in it, and hence in its fault-detection capabilities. It is important to validate the test suite using a technique that allows validation in another way than testing. For this case study, I evaluated the test suite using code inspection and an inverse function. Using the inverse function, I knew in advance what the answer of the true function, implemented by the SUT, had to be. After having successfully evaluated the SUT, I was a lot more confident in its error detection capabilities.

Workload It was a lot of work to design, implement and validate the test suite. For the type of algorithm implemented in the SUT, a formal analysis might actually be simpler. This is just a thought, but it should be considered when the algorithm is as well specified as the one in this case study. The test suite is valid for all implementations of a three-way merge, but a formal analysis and evaluation would result in a good algorithm which could be distributed as a library.

7.2 What would have been done differently given access to the right tools

White-box testing tools White-box testing is a field in which tools are of good help. If there were sophisticated tools available for doing coverage analysis of, for example, MCDC coverage and du-path coverage, I would have spend a lot more time implementing white-box testing. When I, in the case study, achieved statement coverage, and when this was the only coverage analysis available in simple open source tools for Eclipse, I decided to drop white-box testing and go with a black-box testing strategy. This proved, however, to be a good choice, and thus I am not convinced that there is a real need for white-box testing tools when white-box testing is so poor in practice.

7.3 Open issues that should be addressed by research

The specification might be unclear It is important to test that the implementation is correct according to the specification, but an equally important thing is testing that the specification solves the problem it was meant to solve. The SUT is meant to perform a three-way compare and merge in an as good way as possible. But, the specification is only one way, maybe a way which trades accuracy for speed. Hence, I learned that it is important to see the difference between the specification of a SUT and the problem one is trying to solve, if there is any. The open issue here is how to properly handle this difference when testing.

Trade-offs Algorithms may be implemented such that they are quicker but less accurate. This is acceptable is the accuracy is lost for unlikely cases. For the SUT, it may be the case that speed is preferred over accuracy to a certain extent. In the source code of the SUT, right where they have made a kind off trade-off decision they state that "This tends to produce good results most of the time." The open issue here is how trade-off decisions makes it harder to test a system, and whether it is a good idea to do such trade-offs considering a systems testability.

Realistic test cases Establishing the fact that a test case which failures is realistic is important. If one makes a test suite with highly improbable test cases and notices that 90% fails, then this is not as interesting. The open issue here is how to make test cases that are realistic.

References

[1] Paul C. Jorgensen. *Software Testing: A Craftsman's Approach, Third Edition.* AUERBACH, 3 edition, 2008.

[2] Sanjeev Khanna, Keshav Kunal, and Benjamin C. Pierce. A formal investigation of diff3. In Arvind and Prasad, editors, *Foundations of Software Technology and Theoretical Computer Science (FSTTCS)*, December 2007.

[3] Eugene W. Myers. An o(nd) difference algorithm and its variations. *Algorithmica*, 1:251–266, 1986.

[4] Bill Ritcher. Guiffy suremerge - a trustworthy 3-way merge. published online, 10 2009. `http://www.guiffy.com/SureMergeWP.html`.

[5] J. Rivieres and W. Beaton. Eclipse Platform Technical Overview, 2006.

A Index of relevant information found in the document

Detailed test models and requirements All relevant information about the test models and the requirements are spread through the document.

Test driver code (must be commented), test stubs Description of were to find the test code is listed in section 3.3.2.

Detailed test results See the attached file report.txt.zip for a full listing of all test cases which either failed or which were inconclusive.

If possible, source code being tested Where to get the source of the SUT and how to install it is described in section 3.3.1.

UML models A UML model of the test suite is shown in figure 2.

B Known Bugs In the SUT

Out of 380 bugs listed on the Eclipse bug tracking system for the Compare Framework, seven bugs were selected. These bugs are all related to the three-way compare and merge algorithm. Many of the other bugs are related to the GUI-code. The selected bugs are described in turn.

Bug 171678: Token diff treats space as significant

URL	`https://bugs.eclipse.org/bugs/show_bug.cgi?id=171678`
Reproduced as test	DocumentMergerTest.testBug171678()
Bug description	"The token diff used by the text merge viewer treats spaces as significant. What this means is that a line like "a = b" would result is three token diffs when compared with "c + d" even though it is really a complete change."
Bug analysis	This bug is not really a programming error, but rather a problem of specification. How fine should the difference selection be? If there is one change on a line, it better be shown as a single character change. But, if there is a single similarity on a line, it should be shown as a complete change of the whole line.

Bug 271427 - Text Compare mishandles comparison after Roman numerals

URL	`https://bugs.eclipse.org/bugs/show_bug.cgi?id=271427`
Reproduced as test	DocumentMergerTest.testbug271427()
Bug description	Roman numerals are in certain cases found to be different, even if it is the same in all the files in the comparison.
Bug analysis	This bug is related to 289697. See it for the analysis and the fix.

Bug 289697 - Wrong highlighting of individual changes in text compare

URL	`https://bugs.eclipse.org/bugs/show_bug.cgi?id=289697`
Reproduced as test	DocumentMergerTest.testbug289697()

Bug description	"I changed a line in a *.java file from:
	
	`return zip.matches("^[a-zA-Z0-9- \\/]+$");`
	
	to:
	
	`return zip.matches("^[a-zA-Z0-9- \\/]{8}$");`
	
	The higlighted part is old version is "\\/]8$" and in new version it is "\\/]+$". That is obviously wrong, since the "\\/]" part is unchanged , also the "$" is unchanged."

Bug analysis	This bug is related to bug 271427. The problem here is `org.eclipse.compare.contentmergeviewer.TokenComparator`. It groups tokens into three groups: unspecified, whitespaces, digits, letters and quotes. This bug is caused by the unspecified tokens being one group. Treat each unspecified character as a token. Change line 62 in TokenComparator.java.

Bug 260532 - Diff problem in compare view

URL	`https://bugs.eclipse.org/bugs/show_bug.cgi?id=260532`
Reproduced as test	DocumentMergerTest.testbug260532()
Bug description	After having compared two files, one person decided to apply one particular change out of many changes from one file to the other. He then recompared the three files. The applied difference was then again marked as a difference, through a part of another difference this time.
Bug analysis	The same behavior is seen in the two applications diff3 and kdiff3. The behavior is consistent with the algorithm presented in "A Formal Investigation of Diff3" [2]. Ritcher describes a similar problem in [4], and he has implemented his improved three-way merge in his own application. There might be speed-issues related to how the three-way diff is implemented in Eclipse. Maybe the Eclipse implementation is a trade-off between accuracy and speed. The essential point, and the reason the application behaves as the bug described, is that Longest Common Subsequence for the original O and each of the two changed versions, A and B, has multiple equally good answers. The Eclipse compare framework choses non-optimal outcomes of two-way diffs: diff(O, A) and diff(O, B).

Bug 185099 - Ignore whitespace causes wrong line between differences if difference is at bottom of a file

URL	`https://bugs.eclipse.org/bugs/show_bug.cgi?id=185099`
Reproduced as test	DocumentMergerTest.testbug185099()

Bug description	Comparing these strings shows the problem:
	
	`String left  = "A\nB\nC\nD\nE";`
	`String right = "A\nB\nC\nD";`
	
	The differences are shown to be
	
	`left: offset: 8, length: 1`
	`right: offset: 7, length: 0`
	
	which is not correct.

Bug analysis	The difference is really "\nE", but when ignoring whitespaces it becomes "E". E is positioned outside the range of string right. The right-side offset should be 8, because this is really where the difference E is located. Of course, for the actual merging, the white-spaces cannot be ignored.

Bug 306314 - Three-way merge wrongly detects conflict to adjoining (not overlapping) changes

URL	`https://bugs.eclipse.org/bugs/show_bug.cgi?id=306314`
Reproduced as test	class RangeDifferencer3WayDiffTest

Bug description	The following three files are merged as shown in column 5 "merge (cur)". The expected and right behavior are as in column 6 "merge (expected)".
	
	`# | O | A  | B  | merge (cur) | merge (expected)`
	`--+---+----+----+-------------+-----------------`
	`1 | A | A  | A  |             |`
	`2 | B | B1 | B  | <-conflict  | <-left changed`
	`3 | C | C  | C1 | <-conflict  | <-right changed`
	`4 | D | D  | D  |             |`

Bug analysis	This is not a bug if Eclipse Compare is to behave like diff3 as explained in [2]. This bug has recently had a suggested bug-fix including test cases. It does not fix any of the other bugs I have checked. As of 4/5-2010, the bug-fix has been applied to head, but I expect that I will find a case where this does not work, simply because it is not supported by the diff3 algorithm as described in [2].

Bug 88830 - Compare treats new line as non-white space change

URL	`https://bugs.eclipse.org/bugs/show_bug.cgi?id=88830`
Reproduced as test	DocumentMergerTest.testbug88830()

Bug description	When setting the SUT to ignore whitespaces, these two files are still found to be different.
	
	`String left  = "A\nB\nC\nD\nE";`
	`String right = "ABCDE";`

Bug analysis	The reason for this bug is that new-lines have a special meaning when looking at a text-file. A text-file is a list of lines, each line ends with a new-line. Hence, ignoring new-lines makes it ambiguous if this includes new-lines. New-lines are both whitespaces and line-separations. If one ignores all whiespaces, then all text-files will consist of just one line, obviously.