didiffff: A Viewer for Comparing Changes in both Code and Execution Traces

Tetsuya Kanda
Osaka University
Suita, Osaka, Japan
t-kanda@ist.osaka-u.ac.jp

Kazumasa Shimari
Osaka University
Suita, Osaka, Japan
k-simari@ist.osaka-u.ac.jp

Katsuro Inoue
Osaka University
Suita, Osaka, Japan
inoue@ist.osaka-u.ac.jp

ABSTRACT

One of the important purposes of code review is to find potential defects caused by other developers' code changes. When reviewing bug fixes, it is important to check the program behavior is properly changed to remove the bug. On the other hand, it is also important to check the program behavior that is not related to the bug is not changed. To investigate the program behavior, omniscient debugging which records all the runtime events is proposed. With omniscient debugging techniques, existing tools visualize multiple execution paths and the states of local variables of a method, but they are not focusing on code changes. In this paper, we implemented a prototype tool that compares and visualizes the difference between two execution traces caused by code changes. Each variable has a maximum of two lists of values, before and after the code changes, so we proposed their categorization based on their difference of length and contents. We also developed a viewer to show both code changes and the difference of execution traces at a glance by extending our previous viewer for omniscient debugging.

CCS CONCEPTS

• **Software and its engineering → Maintaining software**; **Software evolution**.

KEYWORDS

Software Visualization, Dynamic Analysis, Diff

ACM Reference Format:
Tetsuya Kanda, Kazumasa Shimari, and Katsuro Inoue. 2018. didiffff: A Viewer for Comparing Changes in both Code and Execution Traces. In *Woodstock '18: ACM Symposium on Neural Gaze Detection, June 03–05, 2018, Woodstock, NY*. ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/1122445.1122456

1 INTRODUCTION

One of the important purposes of code review is to find potential defects caused by other developers' code changes [3]. Since code reviewing requires reviewers to understand the code and changes [2], various methods and tools are proposed to assist their task. When reviewing bug fixes, it is important to check the program behavior is properly changed to remove the bug. On the other hand, it is also important to check the program behavior that is not related to the bug is not changed.

Change impact analysis is a collection of techniques for determining the effects of modifications on source code [1]. Such kind of techniques are performed based on static and dynamic analysis, and identify the program code that is potentially affected by changes [11]. On checking the potentially affected code and selecting corresponding test cases, they are very useful.

To investigate the actual changes in the dynamic behavior of a program, omniscient debugging which records all the runtime events is proposed. NOD4J [10] is a tool that records local variables and fields used in a Java program execution and annotates the source code in a web browser.

In this paper, we implemented a prototype tool called didiffff [7] that compares and visualizes the difference between two execution traces caused by code changes. Our tool records whole program execution by running a test case with an omniscient debugging technique before and after code changes. Each variable has a maximum of two value lists, before and after the code changes, so we propose to categorize variable tokens based on their difference of length and contents. We also developed a viewer to show both code changes and the difference of execution traces at a glance by extending the NOD4J viewer component. The viewer is implemented as a web application so that users can investigate the changes easily.

2 RELATED WORKS

Some existing tools aim to visualize multiple executions of the program. Jones et al. presented the visualization technique to assist fault localization tasks [5]. Their tool colors program statement based on the outcome of the test suite, and as a result, faulty statements will be highlighted. REMViewer [8] visualizes multiple execution paths and the states of local variables of a method. Based on the execution trace, REMViewer acts like an interactive debugger but can reproduce multiple executions of a target method simultaneously. Celine et al. visualized a large-scale refactoring process with log-based behavioral differencing [4], comparing two executions before and after the refactoring by highlighting the graph.

However, the approaches described above are proposed mainly for finding defects and visualizing only the difference of the executions. Our tool is designed to visualize both code changes and the difference of execution traces at a glance.

3 COMPARING EXECUTION TRACES

Our proposed tool takes two execution traces recorded with NOD4J as inputs and processes them. This section describes how to compare these two execution traces.

Woodstock '18, June 03–05, 2018, Woodstock, NY

ACM ISBN 978-1-4503-XXXX-X/18/06…$15.00
https://doi.org/10.1145/1122445.1122456

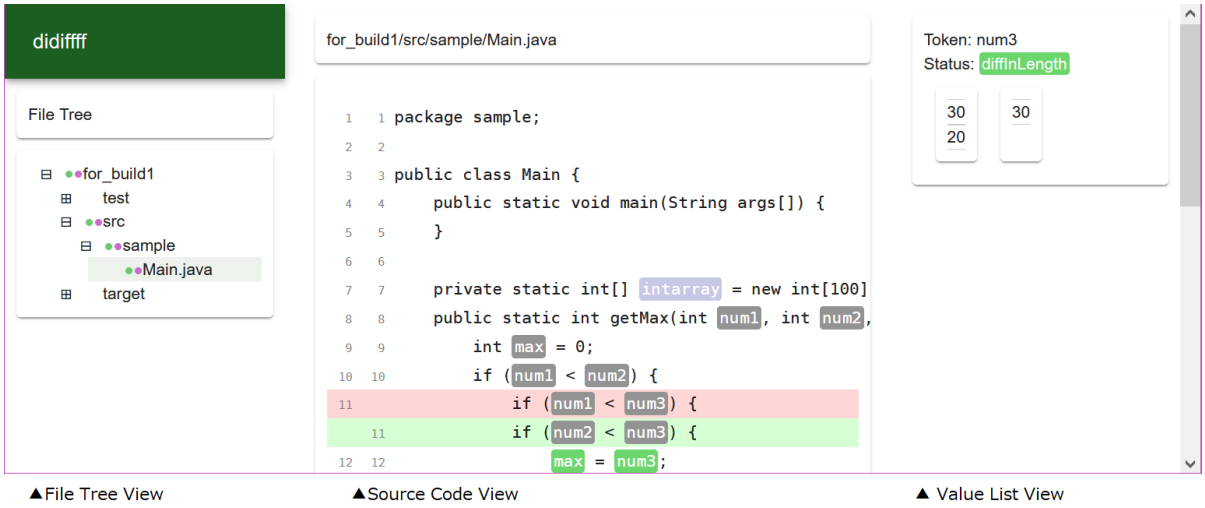


Figure 1: An overview of the Viewer

An execution trace recorded by NOD4J contains the list of values in the variable and source code that contain them. If the variable is a primitive data type, `String`, or `Exception` object, NOD4J records its value or textual contents, otherwise NOD4J records its object ID.

To compare two execution traces, our tool firstly compares the source code to detect changes and then matches the lines. Secondly, our tool compares the execution traces for each variable.

3.1 Comparing source code

Firstly, we extract the correspondence of lines in two execution traces. For each source file that has the same file path in two execution traces, we mark each line as either "add", "delete", or "unchanged" with UNIX diff tool [9]. If there are no files with the same file path on one side, our tool stops processing that file.

Our current implementation does not track renamed files. In addition, it does not track operations that UNIX diff cannot handle such as move method refactoring.

3.2 Comparing value lists

After comparing source code, each variable token has a maximum of two value lists. The tool compares these traces and categorizes them as the following criteria.

Case 0: If the variable token has no value lists, do nothing.

Case 1: If the variable token has its execution trace but is included in the "add" or "delete" line, we cannot find an exact matching for variables that appeared in the changes. The tool mark it as **"Trace 1 only"** if it is included in the deleted lines and **"Trace 2 only"** if it is included in the added lines. These categories indicate that the tool could not find the value list to compare.

Case 2: If the length of two value lists is different, the tool reports that **"Diff in length"**. This includes the case that one of the lists is empty and the other is not. Unlike the previous case, the overall execution path might be changed in this case so that the number of times the variable has been called has also changed.

Case 3: If the lengths of two value lists are the same and they are not considered as a primitive data type nor `String`, the tool reports that **"Same trace length Object"** since the contents of the object is not available in the trace.

Case 4: Here, the type of two value lists are a primitive data type or `String` and the tool compares the values in these two value lists. If they have exactly the same contents, the tool reports **"Diff in trace"**, otherwise **"No diff in trace"**.

It should be noted that it is not a simple task to take diff of the value list. Let us consider the case where the value list `[1, 1, 2, 1]` turns into `[1, 2, 1, 1]` after fixing the bug. In this case, there are several possible scenarios; the execution before the value 2 appeared becomes shorter and after it becomes longer, the intermediate values `[2, 1]` have changed to `[1, 2]`, and so on. In another situation like the value list `[1]` turns into `[1, 1, 1]`, it is almost impossible to determine which `[1]` in the revised version corresponds to the one before the revision, and it is even possible that none of them correspond in the first place. For these reasons, we decided not to show the detailed diff status like text (add, delete, ...) for trace and leave the further investigation up to users.

4 VIEWER: WEB UI

To show both code changes and the difference of execution traces in one place, we developed an extended version of the NOD4J viewer component. The viewer is implemented as a web application running on the Node.js environment. Figure 1 is a screenshot of a tool, running NOD4J example test code. The viewer consists of three components: the file tree view, the source code view, and the value list view.

4.1 File Tree View

The file tree view provides the list of directories and files included in the analysis target. Directories and files which are contained in both two execution traces are shown in the view. Each file node in the tree has two indicators to show the existence of differences. The green one on the left shows the differences in the text and

the purple one on the right shows the differences in the execution traces. Directory nodes also have similar indicators which show whether some of the files in the directory have differences. With these indicators, users can easily reach files to investigate changes.

In Figure 1, we can see that some directories (test, target) have their child node but they have no difference in either the source code or the value lists.

4.2 Source Code View

The source code view displays the contents of the source code. The first two columns show the line numbers of the two files to be compared. As with most text diff tools, deleted and added lines are highlighted.

Variable tokens which have at least one value list are also highlighted. By clicking one of the highlighted variable tokens, the value list(s) of it is displayed in the value list view. Highlights are based on the status described in Section 3.2. If the variable is categorized as "No diff in trace", "Trace 1 only", or "Trace 2 only", it is highlighted in gray, and "Same trace length Object" is highlighted in light purple as well. In these two cases, traces might be either unaffected by the code changes or they cannot be checked in detail by our tool. "Diff in trace" is highlighted in blue and "Diff in length" is in green. The behavior of the target program is changed in these two cases so that users can investigate the effects of code changes by observing such variable tokens.

In Figure 1, we can see that arguments of the method `getMax()` on line 8 contain the same value lists in two execution traces. The view also indicates that variables on line 12, right after the code changes in line 11, have different length of value lists.

4.3 Value list View

The value list view shows the recorded value list(s) of the selected variable token with its status. If the variable is a primitive data type or `String`, concrete values are displayed; otherwise, lists of object ids with their class name are displayed. As mentioned in Section 3.2, the tool just shows two value lists and does not provide detailed information like text diff in the current implementation.

The value list view in Figure 1 shows the status and value lists of variable `num3` on line 12. Since the previous conditional expression was modified, the execution path was changed. We can check how each variable is affected by code changes using this view.

5 EXAMPLE

We explain the usage of our tool with the example scenario; reviewing the debug result of a bug **Math 57** of the Defects4J dataset [6]. The bug is found in the clustering component of the library. One of the variables to contain the distance is declared as `int` so that the distances between points are truncated to integers. This bug is removed by changing the type of that variable from `int` to `double`. In this case, only one line (more specifically, only one token) has been changed to fix the bug. After fixing the bug, the new test case to ensure the bug is removed is added to the repository. We invoke this test case before and after fixing the bug and visualize with our tool to show how those execution traces are changed.

Figure 2 shows the part of the file tree view of comparing two execution traces. Indicators in the file tree view show that there are

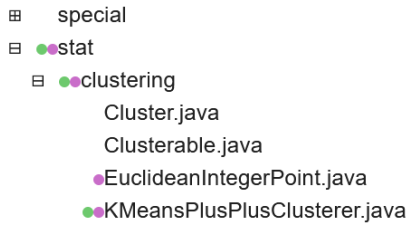


Figure 2: Part of the File Tree View in the example scenario. Some files and directories have indicators turned on.

differences in both text and traces is in the `clustering` directory. Some files have differences only in execution traces but one file named `KMeansPlusPlusClusterer.java` also has differences in text. Now we select this file to check detailed information.

Figure 3 displays the part of the source code view of a file `KMeansPlusPlusClusterer.java`, around a method including code changes. The first thing we can see is that line 175 is changed and it is because the type of variable `sum` is modified from `int` to `double`. No other code changes are found in this file, so next, we investigate the difference of execution trace.

Focusing on method arguments in line 162, they have the same length of value lists before and after the code changes. In addition, the second argument `k` has the completely same values. They suggest that this method was called the same number of times before and after the modification. After the method declaration, there are some Object variables until code changes on line 175, Our tool cannot investigate the detailed comparison of value lists of them, but we can still see that the length of the value lists is the same.

Some lines after line 175, the point of code change, have no difference in trace or object variables which have the same length of the value lists. On line 180, the view shows that there are some differences in the traces of variable `sum`. Figure 4a shows the value list view selecting variable `sum` on line 180. Looking into the traces in the trace view, we can find that the result of the latest execution of this line has changed. For a better understanding of the situation, we also show the value list view of variable `d` on line 180 in Figure 4b. We can see that the same small value `0.00001` is assigned to the `sum` before and after the code changes, but it is no longer rounded after changing the code.

A loop variable `i` and an Array object `dx2` in line 187 are highlighted in green, which indicates they have different length of value lists. The value list of the variable `i` in line 187 is displayed in Figure 4c. We can see that changing the type of `sum` also leads to a different comparison result in the condition statement. Please note that this loop and its following condition statement have been repeated over 64 times, which is the default value for the length limit of the NOD4J recorder so that values on the first part are missing and the list on the right-hand side starts with a value `9936`.

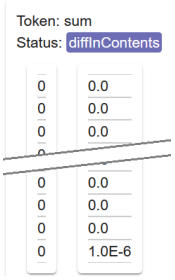
The if statement on line 188 is executed at most once since it has a break statement inside. The output of this method changed because different values of `i` are specified as arguments to the `remove()` method call on line 189.

As mentioned in this section, our tool makes it possible to track changes in behavior caused by code changes in a single application, without setting breakpoints or opening debuggers in parallel.

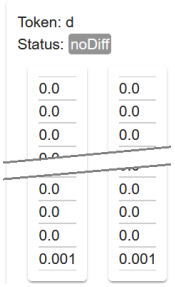
```
161 161   private static <T extends Clusterable<T>> List<Cluster<T>>
162 162       chooseInitialCenters(final Collection<T> points, final int k, final Random random) {
163 163
164 164       final List<T> pointSet = new ArrayList<T>(points);
165 165       final List<Cluster<T>> resultSet = new ArrayList<Cluster<T>>();
166 166
167 167       // Choose one center uniformly at random from among the data points.
168 168       final T firstPoint = pointSet.remove(random.nextInt(pointSet.size()));
169 169       resultSet.add(new Cluster<T>(firstPoint));
170 170
171 171       final double[] dx2 = new double[pointSet.size()];
172 172       while (resultSet.size() < k) {
173 173           // For each data point x, compute D(x), the distance between x and
174 174           // the nearest center that has already been chosen.
175           int sum = 0;
    175           double sum = 0;
176 176           for (int i = 0; i < pointSet.size(); i++) {
177 177               final T p = pointSet.get(i);
178 178               final Cluster<T> nearest = getNearestCluster(resultSet, p);
179 179               final double d = p.distanceFrom(nearest.getCenter());
180 180               sum += d * d;
181 181               dx2[i] = sum;
182 182           }
183 183
184 184           // Add one new data point as a center. Each point x is chosen with
185 185           // probability proportional to D(x)2
186 186           final double r = random.nextDouble() * sum;
187 187           for (int i = 0 ; i < dx2.length; i++) {
188 188               if (dx2[i] >= r) {
189 189                   final T p = pointSet.remove(i);
190 190                   resultSet.add(new Cluster<T>(p));
191 191                   break;
192 192               }
193 193           }
194 194       }
195 195
196 196       return resultSet;
197 197
198 198   }
```

ⓐ ⓑ ⓒ

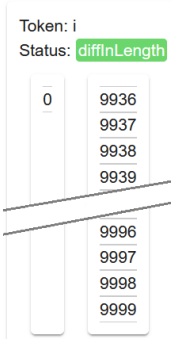
Figure 3: Source Code View in the example scenario. Marks a, b, and c are given for explanatory purposes only and do not appear in the actual view.



(a) Selecting variable `sum` on line 180.



(b) Selecting variable `d` on line 180.



(c) Selecting the second variable `i` on line 187.

Figure 4: Value List View in the example scenario.

6 CONCLUSION AND FUTURE WORK

It is important to understand how changes in the code affected the behavior of the program on code reviewing. In this paper, we proposed a tool that visualizes both code changes and the difference of execution traces at a glance. We categorize variable tokens based on their value lists recorded in the execution traces. The viewer is based on the existing tool NOD4J, which records and visualizes an execution trace, and we extended it for handling two execution traces. Our tool allows developers to track not only the test results but also the changes in program behavior before and after code changes in a single window of the web browser.

We are planning to extend the tool by assuming usage scenarios. For example, some test cases give multiple argument sets as input to a method for boundary value tests. If the test fails in the middle of an iteration, the remaining inputs will not be executed. As a result, many variables will have a shorter length of value list comparing with when the test succeeds and indicated as "Diff in length" in the viewer. If we assume that the trace is fully recorded, we can split the execution traces into each method execution and highlight the iteration in which the difference first occurred. We would also like to investigate more user-friendly interface for comparing value lists to assist users' tasks.

ACKNOWLEDGMENT

This work was supported by JSPS KAKENHI Grant Numbers JP18H04094 and JP19K20239.

REFERENCES

[1] Robert Arnold and Shawn Bohner. 1996. *Software Change Impact Analysis*. IEEE Computer Society Press, Washington, DC, USA.

[2] Alberto Bacchelli and Christian Bird. 2013. Expectations, outcomes, and challenges of modern code review. In *Proceedings of the 35th International Conference on Software Engineering (ICSE 2013)*. 712–721. https://doi.org/10.1109/ICSE.2013.6606617

[3] Amiangshu Bosu, Jeffrey C. Carver, Christian Bird, Jonathan Orbeck, and Christopher Chockley. 2017. Process Aspects and Social Dynamics of Contemporary Code Review: Insights from Open Source Development and Industrial Practice at Microsoft. *IEEE Transactions on Software Engineering* 43, 1 (2017), 56–75. https://doi.org/10.1109/TSE.2016.2576451

[4] Celine Deknop, Kim Mens, Alexandre Bergel, Johan Fabry, and Vadim Zaytsev. 2021. A Scalable Log Differencing Visualisation Applied to COBOL Refactoring. In *Proceedings of the 2021 Working Conference on Software Visualization (VISSOFT 2021)*. 1–11. https://doi.org/10.1109/VISSOFT52517.2021.00010

[5] James A. Jones, Mary Jean Harrold, and John Stasko. 2002. Visualization of test information to assist fault localization. In *Proceedings of the 24th international conference on Software engineering (ICSE 2002)*. 467–477. https://doi.org/10.1145/581339.581397

[6] René Just, Darioush Jalali, and Michael D. Ernst. 2014. Defects4J: a database of existing faults to enable controlled testing studies for Java programs. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis (ISSTA 2014)*. 437–440. https://doi.org/10.1145/2610384.2628055

[7] Tetsuya Kanda. 2022. https://github.com/tetsuyakanda/didiffff/

[8] Toshinori Matsumura, Takashi Ishio, Yu Kashima, and Katsuro Inoue. 2014. Repeatedly-executed-method viewer for efficient visualization of execution paths and states in Java. In *Proceedings of the 22nd International Conference on Program Comprehension (ICPC 2014)*. 253–257. https://doi.org/10.1145/2597008.2597803

[9] Eugene W Myers. 1986. An O(ND) difference algorithm and its variations. *Algorithmica* 1, 1-4 (1986), 251–266. https://doi.org/10.1007/BF01840446

[10] Kazumasa Shimari, Takashi Ishio, Tetsuya Kanda, Naoto Ishida, and Katsuro Inoue. 2021. NOD4J: Near-Omniscient Debugging Tool for Java Using Size-Limited Execution Trace. *Science of Computer Programming* 206 (2021), 102630. https://doi.org/10.1016/j.scico.2021.102630

[11] Xiaoxia Ren, B.G. Ryder, M. Stoerzer, and F. Tip. 2005. Chianti: a change impact analysis tool for Java programs. In *Proceedings of the 27th International Conference on Software Engineering (ICSE 2005)*. 664–665. https://doi.org/10.1109/ICSE.2005.1553643