Journal of Visual Language and Computing

journal homepage: www.ksiresearch.org/jvlc/

ViBERT: Visual Behavior Regression Testing

Chunying Zhao[a,*], Cong Chen[b], Kang Zhang[c] and Jun Kong[d]

[a]School of Computer Sciences, Western Illinois University, USA
[b]Independent Scholar
[c]Department of Computer Science, The University of Texas at Dallas, USA
[d]Department of Computer Science, North Dakota State University, USA

ARTICLE INFO

Article History:
Submitted 12.1.2020
Revised 12.7.2020
Second Revision 12.9.2020
Accepted 12.18.2020

Keywords:
Software Visualization
Program Comprehension
Behavior Regression Testing

ABSTRACT

Regression testing is a type of software testing that aims at identifying faults caused by code changes. Regression testing is important especially during software evolution and maintenance. As developers integrate programs or make updates to a software system, they need to make sure the changes do not adversely affect other parts of the system. Using dynamic analysis, behavioral regression testing (BERT) is one of the techniques proposed to solve the problem by re-executing test cases that target the affected area. It compares the behavior of a program before and after the changes upon certain test cases. This paper proposes *Visual BEhavioral Regression Testing (ViBERT)*, a visualization approach to comparing the behavioral differences between the new and old versions of a program in regression testing. We build a prototype called *SoftLink*, a visual environment that shows correlation/difference between two versions of a program behavior. SoftLink displays call graphs of two executions on angled parallel planes in a 3D space, and constructs correlations between them. It provides developers with an intuitive interpretation of the testing results. A case study is presented.

1. Introduction

Software visualization is defined as "the use of the crafts of typography, graphic design, animation and cinematography with modern human computer interaction and computer graphics technology to facilitate both the human understanding and effective use of computer software" [38]. Visual clues such as color, shape, and metaphors ease the cognitive load of understanding software systems [7][40]. Numerous research has been proposed and various tools have been built to visualize different aspects of software systems, such as static program structures [11][17][21], dynamic program executions [8][14][16][29], software evolution[6][9], and debugging results [12].

Regression testing is an important type of software testing. During a software development and maintenance process, software may go through many changes due to system integration or software updates. Each change may introduce unwanted faults to software. Regression testing re-executes existing test cases after the source code is changed in order to determine whether the modified version has introduced regression faults into the previous working version [26]. Regression testing techniques heavily rely on the quality and sufficiency of test cases. Testers have to compromise between making thorough testing and lowering the cost. Numerous testing selection and prioritization approaches have been proposed [10][25][26][31][32][33].

Behavioral regression testing (BERT) [28][37] addresses this dilemma by identifying behavioral differences between two versions of a program through dynamic analysis. Since two consecutive revisions of a program usually do not differ significantly, BERT can reduce the number of test cases needed while achieving promising results. Behavioral regression testing relies on comparing the behavioral differences between two versions of a program through dynamic analysis to identify unforeseen side effects.

*Corresponding author
Email address: c-zhao@wiu.edu (Chunying Zhao)
congchenutd@gmail.com (Cong Chen)
kzhang@utdallas.edu (Kang Zhang)
jun.kong@ndsu.edu (Jun Kong)

DOI reference number: 10.18293/JVLC2020N2-010

Inspecting the differences between versions of program executions is tedious and error prone. To strengthen the effectiveness of BERT, this paper proposes Visual BEhavioral Regression Testing (ViBERT), a visual approach for comparing program behaviors in regression testing. We have built a semi-automatic tool called Softlink. It is a visual environment that displays and compares two or more program executions in a 3D space. It provides multiple viewpoints and directly shows the correlations between execution traces. The novelty of our approach is that it not only visually shows the differences and commonalities of two consecutive executions, but also provides a mental image of the location of the behavioral differences within the context of method calls. Our work enhances BERT with a visual representation. To our best knowledge, no studies have been conducted on comparing execution traces using 3D visualization in regression testing.

The rest of the paper is organized as follows. Section 2 illustrates a motivating example. Section 3 presents the overview of the approach. Section 4 describes how the execution traces are collected and abstracted. Section 5 shows the construction of SoftLink. Section 6 explains a case study and analyzes the results. Related work is reviewed in Section 7. Section 8 concludes the paper and presents our future work.

2. A Motivating Example

In this section, we present a motivating example to show how visualization can enhance BERT. The class *Money* is a Java package of JUnit4 library illustrating how to write unit tests with Junit [5]. As shown in Figure 1, Money.equals() is a method of Money that determines whether two monies are equal or not.

```
public boolean equals(Object anObject) {
    if (isZero())
        if (anObject instanceof IMoney)
            return ((IMoney)anObject).isZero();
    if (anObject instanceof Money) {
        Money aMoney = (Money)anObject;
        return  aMoney.currency().equals(currency())
&& amount() == aMoney.amount();
    }
    return false;
}
```

Figure 1: Original version of Money.equals().

Figure 2 shows the JUnit test cases for testing the method Money.equals().

```
public void testMoneyEquals() {
01  assertTrue ( !f12CHF.equals(null) );
    Money equalMoney = new Money(12, "CHF");
02  assertEquals (f12CHF, f12CHF);
03  assertEquals (f12CHF, equalMoney);
04  assertEquals (f12CHF.hashCode(),
                  equalMoney.hashCode() );
05  assertTrue ( !f12CHF.equals(f14CHF) );
}
```

Figure 2: Test cases in Money for Money.equals().

We deliberately remove the statement aMoney.currency().equals(currency()) && as shown in Figure 3, simulating a situation that a developer changes the code but introduces an error: the program omits checking currency when it compares two monies.

```
public boolean equals(Object anObject) {
    if (isZero())
        if (anObject instanceof IMoney)
            return ((IMoney)anObject).isZero();
    if (anObject instanceof Money) {
        Money aMoney = (Money)anObject;
        return  ~~aMoney.currency().equals(currency())
&&~~ amount() == aMoney.amount();
    }
    return false;
}
```

Figure 3: Modified version of Money.equals().

After running the test on the modified version, JUnit failed to catch the bug. The original Junit test cases are not sufficient to catch the error because the monies in the test cases have the same type of currency, such as f12CHF and f14CHF.

By visually comparing the runtime behaviors of two versions of the program, however, developers can easily identify the behavioral variations. Figure 4 correlates the behavior of two versions of the program on two 2D planes. There are observable differences in the visual presentation. The red circles drawn by hands on the left plane indicate the method invocations (currency()) executed in the original code but not executed in the modified version. With this visual hint, developers can easily locate the code affected by the modification and further check whether these behavioral variations are caused by errors or intended modifications. To further illustrate our approach, a case study has been conducted on an open-source program in Section 6. The initial results provide shows that our approach reveals the behavioral variations of the systems under study with a visual presentation.

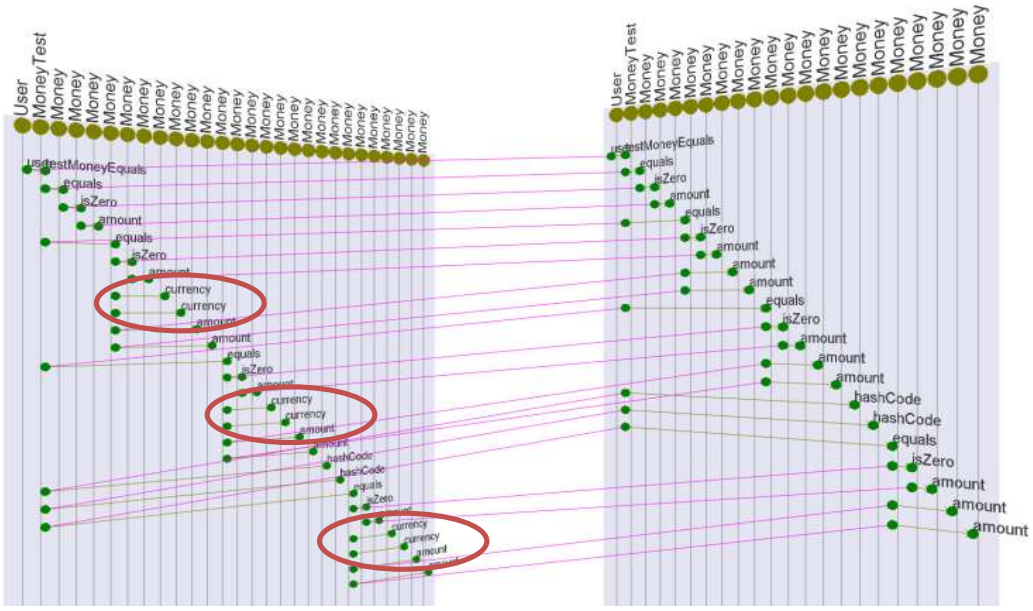

Figure 4: Visual representation of the Money executions. The left plane presents the original version, and the right one denotes the modified version. The purple correlations lines show the mapping between these two executions.

3. Visual Behavioral Regression Testing (ViBERT)

3.1 Approach Overview

Behavioral regression testing (BERT) has been used as an effective technique to identify behavioral differences between two versions of a program through dynamic analysis. Since two consecutive revisions of a program usually do not differ significantly, BERT can greatly reduce the number of test cases needed while achieving promising results. BERT typically works as follows [28]:

1) Analyze the changes between two versions and automatically generate a large number of test cases that cover the changed parts of the code.
2) Run the generated test cases on the old and new versions of the code and identify differences in the tests' outputs.
3) Analyze the identified differences and presenting them to the developer.

BERT analyzes behavioral differences by comparing the program outcomes. SoftLink complements BERT by visualizing behavioral variations. SoftLink can work seamlessly with existing BERT tools, and display the correlations between consecutive versions. As an enhancement to BERT, ViBERT works in the following steps:

1) Insert AspectJ instrumentations to the test suite automatically generated in Step 1 of BERT that focuses on the changed parts of the program.
2) Run Step 2 of BERT and generate traces for the two executions to be compared.
3) Use SoftLink to visualize the correlations between two versions of program executions and highlight their differences.

3.2 Design Characteristics

As a software visualization tool, SoftLink is specifically tailored to correlation visualization. SoftLink visualizes abstracted call graphs on 2D planes in a 3D space. SoftLink takes advantage of the benefits of angled and paralleled views. As the viewpoint is changed, the arrangements of planes can be dynamically updated accordingly, in a similar fashion as a camera model. Planes with corresponding correlations interesting to the user always face the user as depicted in Figure 5.

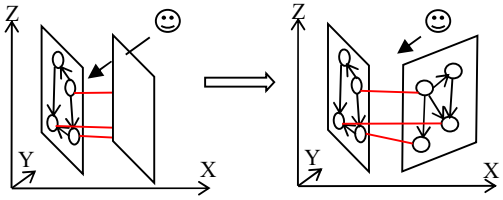

Figure 5: Auto-orienting views.

We design the visual features of SoftLink following the general functional requirements proposed by Kienle and Müller [23]:

- **Views** (linked static views and dynamically synchronized views): SoftLink incorporates three views: a 3D correlation view, a modulation view, and a source code view. These views are linked such that the change of one view automatically triggers the changes of the others. Moreover, these views, especially the 3D correlation view, are

dynamically synchronized with the underlying data.

- **Abstraction**: To effectively visualize complex software systems, a visualization tool should support adjustable granularities (i.e. abstraction levels) and provide sufficiently detailed information on demand. We use a multi-level abstraction on execution traces to enable in-dept exploration of program. In SoftLink, nested method calls can be folded or unfolded corresponding to the change of the abstraction level.
- **Search and Code Proximity**: Finding text strings in the source code corresponding to objects in the visual representation is considered "absolutely essential" [23]. SoftLink provides a query function with a search bar, where users can easily locate source code to their interests.
- **Automatic Layout**: SoftLink uses a multi-plane presentation to visualize multiple executions. It automatically displays execution planes in the 3D space. Planes are dynamically angled towards the user so that both individual executions and their correlations have the best exposure.
- **History/Undo**: SoftLink has an interactive interface that allows the user to click on visual objects while navigating in the 3D space. Iteratively, upon each click, a new nested plane visualizing the detailed information is popped up. All the upper-level planes are kept on the screen to show the browsing history.

4. Execution Traces

4.1 Execution Trace Collection

Obtaining execution traces is the first step to correlate executions. We choose AspectJ[2], a Java implementation of aspect-oriented programming, to intercept program execution metadata, because it can non-intrusively extract runtime traces with a high level of flexibility and expressiveness.

Using the following aspect in Figure 6, we record each method call's signature along with the name of the object it belongs to and the time the program enters and leaves the method.

```
public aspect Trace {
  pointcut allCalls() : execution( * *.*(..));
  before() : allCalls() {
    String signature =
thisJoinPointStaticPart.getSignature().toShortString();
    if(!signature.isEmpty()) {
      String log = "-> "+ signature+ "$" +
Thread.currentThread().getName()+ "*" +
System.currentTimeMillis() + "$";
      System.out.println(log);
    }
  }
  after() : allCalls() {
    String signature =
thisJoinPointStaticPart.getSignature().toShortString();
    if(!signature.isEmpty()) {
      String log = "<- "+ signature+ "$" +
Thread.currentThread().getName()+ "*" +
System.currentTimeMillis() + "$";
      System.out.println(log);
    }
  }
}
```

Figure 6: Definition of Aspect

The plain-text trace log captured by this aspect is then imported to SoftLink, and automatically transformed to call graphs specified in GraphML [3], an XML-based graph presentation. In Softlink, we enhance our previous work on program abstraction [43] and built an Abstracer to perform such transformation.

4.2 Execution Trace Representation and Abstraction

Execution traces need to be properly represented and abstracted before being analyzed as otherwise the user can be misled by partial information or overwhelmed by too much trivial information. Proper abstraction at various granularities makes it possible to display a large volume of program data on limited visual space. When comparing program executions, we identify equivalent substructures in two abstracted call graphs. We represent the call graph *G(N,E)* in a tree structure that consists of multiple caller-callee chains built from the GraphML runtime trace.

Definition 1: A call graph *G(N,E)* is a directed node-link graph, where the set of nodes *N* denote methods and the set of edges *E* represent method invocations.

Each edge directs from a caller to a callee. Each method invocation is annotated with two parameters: the *depth* and the *length* of the call chain. The depth here refers to the depth of the call chain through which we adjust the granularity of the visualization. The length of a call chain is defined as the number of nodes in the path from the root to the leaf in a call graph, which is used to prune short call chains in Abstracer. Each directed edge is annotated with the number of method call repetitions. For instance, there are three call chains in the call graph of Figure 7: *a-b*, *a-c-d*, and *a-c-e*. The lengths of each call chain are 2, 3, and 3, respectively. The depth of each method is as follows: $\theta_{depth}(a) = 1$, $\theta_{depth}(b) =$

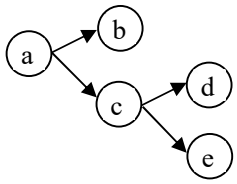

Figure 7: An example for abstraction

$\theta_{depth}(c) = 2$, $\theta_{depth}(d) = \theta_{depth}(e) = 3$.

Abstracer, the abstraction engine integrated into SoftLink, is used to remove less significant information not to be shown in the graphical representation, such as method calls that contribute little to the comprehension of program behavior. We consider three criteria for execution abstraction:

- **Continuous repetitions**. Continuously repeated method invocations can be collapsed to one occurrence. Such repetitions mostly manifest themselves as loops. For instance, in a sequence of method calls *EABCABCF* (a letter represents a method call), the call sequence *ABC* is considered duplicated. Thus, only one occurrence of *ABC* is shown in the call graph. We can label the corresponding edge in the call graph with the number of repetitions. Noncontiguous repetitions are not collapsed because they may belong to different abstraction levels.
- **Depth of methods in a call chain**. This type of abstraction relies on the depth threshold θ_{depth} (an integer specified by the user). Methods whose nesting depths in the call chain are deeper than this threshold can be collapsed. For instance, given θ_{depth} =3, the methods with a depth of 4 or more are collapsed, and not shown in the abstracted scenario. Low-level methods provide detailed information for high-level abstract events and can be unfolded if the user lowers the abstraction level.
- **Short call chains**. A long call chain including more method invocations may represent a significant function of a program. SoftLink uses parameter θ_{length} as a threshold to specify the minimum length of call chains. Method invocations with call chains shorter than θ_{length} are pruned. SoftLink abstracts the call graph by traversing it and collapse methods according to the customizable parameters θ_{depth} and θ_{length}.

5. SOFTLINK

5.1 Overview

Figure 8 shows the interface of SoftLink that includes three views: a 3D correlation view, a modulation view, and a source code view. The controls of SoftLink are on the menu bar. The user can use the *file* menu to import trace logs and specify the number of executions to be correlated. The *action* menu includes commands for specifying trace abstraction levels and constructing correlations. SoftLink first loads the selected plain-text trace files and transforms the files into call graphs in GraphML. It trims the call graphs based on the abstraction parameters set by the user. Call graphs are displayed on individual 2D planes, similar to sequence

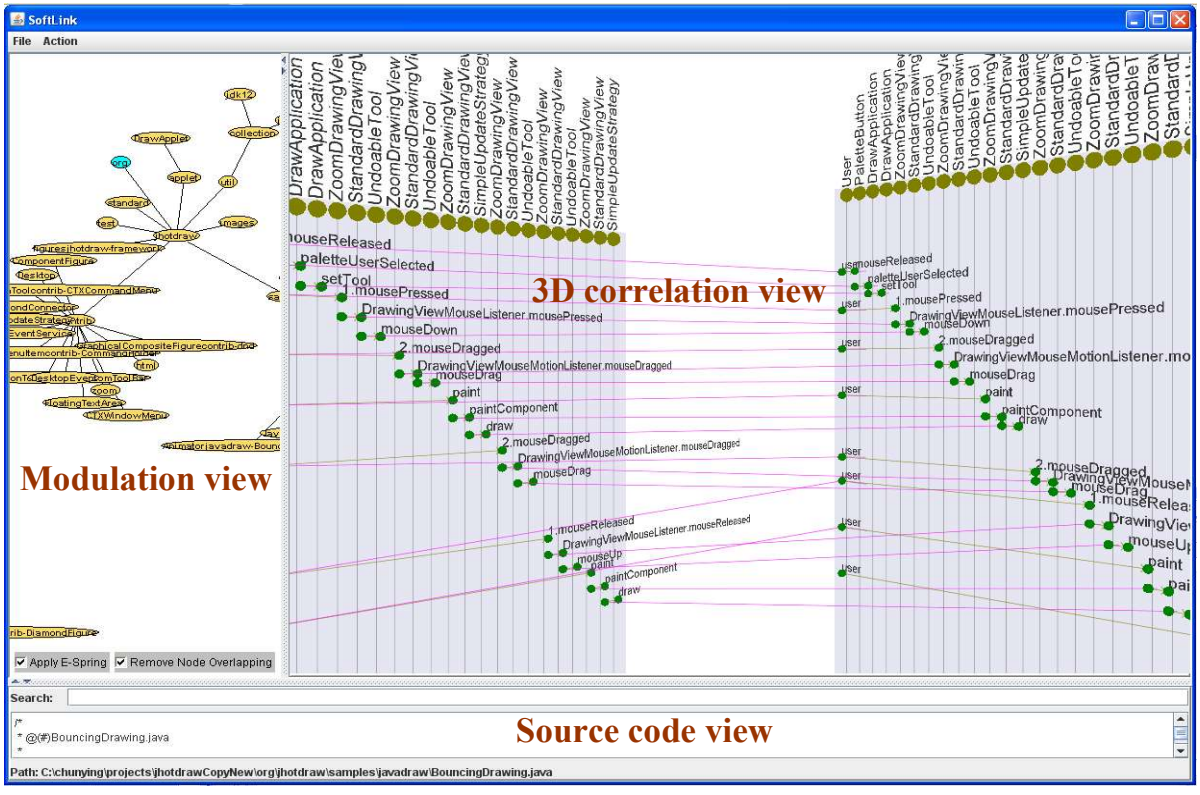

Figure 8: Three views of SoftLink.

diagrams. Then, the correlations are visualized in the 3D correlation view. The 3D scene is automatically rendered when the user changes the abstraction level.

5.2 Views in SoftLink

The 3D correlation view is built using Java3D. Each brown sphere represents an object. Green spheres represent methods. Horizontal green lines indicate the calling relationship between methods. Purple lines represent the correlations between two executions. These colors are selected to achieve some contrast against the background. This 3D scene provides users with multiple viewpoints to observe the relationship between executions. Users can choose to observe the differences or commonalities.

The modulation view shows the hierarchical packaging structure of the program in a force-directed layout using the E-spring Algorithm [24]. High-level organizations of system modules represent the composition of a system and are commonly visualized using the tree structure in a node-link graphical format. Each sub-package is a child node of its parent package, while tree leaves represent files. This modulation view gives developers an overview of the program structure.

The source code view provides a fast access to methods in the source file corresponding to the visual entity that the user is interested in. Having spotted desired information in the 3D visual representation, the user might need to check the corresponding source code. The source code view of SoftLink is synchronized with the visual representation in the 3D correlation view by highlighting the queried method in red. The file path of the searched method is shown in the status bar in the source code view.

5.3 Iterative Multi-level Nested Visualization

5.3.1 Zooming and Rotatable Scene

SoftLink provides efficient interaction and navigation capabilities. In the current implementation, mouse is used for picking and rotating individual planes and visual objects in the 3D correlation view. SoftLink allows the user to move or rotate each single plane to any angle around any axis in 3D space. The keyboard is used to control the entire 3D scene, such as zooming, rotating, and moving the user's viewpoint. To provide customizable views, when the viewpoint moves, each individual plane in the scene can be adjusted accordingly to the viewpoint.

5.3.2 Iterative Multi-level Nested Visualization

Program execution is hard to visualize if all the method invocations need to be displayed. Even if only a portion of a software system is executed, the collected

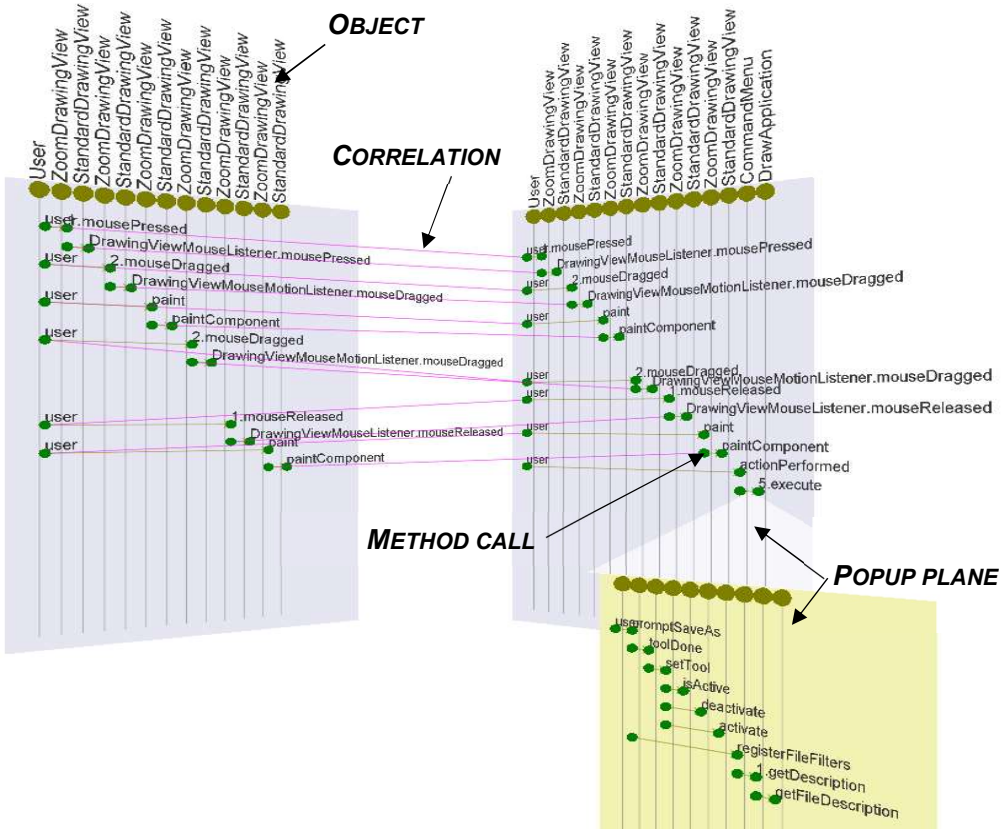

Figure 9: Visual representation of correlations between two executions.

traces can be millions of lines, making it incomprehensible. To address this limitation, SoftLink utilizes multi-level abstraction and iterative drill-down visualization as in Figure 9. When the user clicks a visual object representing a method, the nested interactions within that method are shown in a popup plane attached to the clicked method. Iteratively, the visual objects on the newly popped plane can also be unfolded upon the user's click. By drilling down through multiple planes, the user can get more detailed information. This capability is particularly suitable for a system equipped with an eye-tracker [22].

6. Case Study

We applied ViBERT on an industrial software package, *Joda-Time* [4], a Java date and time library. Joda-Time 1.6.2 has about 4615 classes in the source code, and 4080 classes in the testing code. We select a number of real bugs detected and fixed in the development process of Joda-Time.

Both BERT and ViBERT focus on the changes between two adjacent revisions, we select two revisions of Joda-Time to simulate the regression testing. Suppose revision r_i fixes a bug in revision r_j, we can interpret that revision r_j introduces the bug in *revision* r_i, and use this bug to simulate a real regression fault. We apply ViBERT to identify the regression faults in the changes between revisions of the program.

The *Subversion* repository[1] of Joda-Time contains about 1610 revisions. We search the history for the revisions that have fixed bugs in previous revisions. 156 version pairs <r_i, r_j> are found, where revision r_i fixes certain bugs in revision r_j. Finding regression faults in a software's history is time consuming, requiring a manual process:

(1) Search the revision history for a bug that has been fixed.

(2) Locate the revision and the source code where the bug first appears.

(3) Examine whether the interface of the source code has been changed between the revision that introduces the bug and the revision before it. If the interface is not changed, then the bug is considered a regression fault that was introduced by the new revision.

The selected revision pair is <r1576, r1577>. Revision r1577 fixes a bug in the method *AbstractDuration.toString()* in revision r1576. This method produces wrong output for negative inputs. Putting these two revisions in a regression testing setting, we take revision r1577 as the old version without the bug, and revision r1576 as the new version that introduces bugs.

To catch the bug using regression testing, developers first study the changes the new version has made to the source code, and then create test cases targeting the changes. A test suite containing nine JUnit test cases for the method *AbstractDuration.toString()* is defined as shown in Figure 10:

```
public void testToString() {
01  assertEquals("PT0S",
                  new Duration(0L).toString());
02  assertEquals("PT10S",
                 new Duration(10000L).toString());
03  assertEquals("PT1S",
                  new Duration(1000L).toString());
04  assertEquals("PT12.345S",
                 new Duration(12345L).toString());
05  assertEquals("PT-12.345S",
                new Duration(-12345L).toString());
06  assertEquals("PT-1.123S",
                 new Duration(-1123L).toString());
07  assertEquals("PT-0.123S",
                 new Duration(-123L).toString());
08  assertEquals("PT-0.012S",
                  new Duration(-12L).toString());
09  assertEquals("PT-0.001S",
                  new Duration(-1L).toString());
}
```

Figure 10: Test cases in Joda-Time for Duration.toString().

We first run the existing JUnit test suite on the new version of the program, the test suspends at test case 07, indicating that the actual output is not expected. By observing the changes that the new version has made to the source code, we can conclude that the program behavior starts to differ when the length of the output is greater than 8 (or 7 if the output is a positive number). In the given test suite, however, starting from test case 04, the lengths of the expected outputs are all greater than 8. The program actually behaves differently since test case 04. Therefore, although running the existing test suite can eventually capture the bug, it could have revealed the change earlier (from test case 04 instead of test case 07).

By exploring the behavioral variations of these two revisions, ViBERT intends to detect this potential problem, and alerts developers with visual hints. We run the test cases on both versions and compare their differences. As an enhancement to behavioral regression testing, we use visual representations to show behavioral difference. We obtain the execution traces by embedding AspectJ instrumentations into JUnit testing code. Then, ViBERT visualizes the correlations between the two executions. Figure 11 shows the result in SoftLink. The left plane represents revision r1577, and the right one represents r1576. The numbers correspond to the test cases in Figure 10.

The left plane successfully runs all the test cases. The right plane shows only 7 test cases, because the test stops at test case 07. We notice that since test case 04, two executions exhibit different behaviors due to the changes to the code. The method calls to "*appendPaddedInteger*" in the right panel (revision: r1576) do not exist in the left panel (revision: r1577). Via visual inspection, developers can identify the

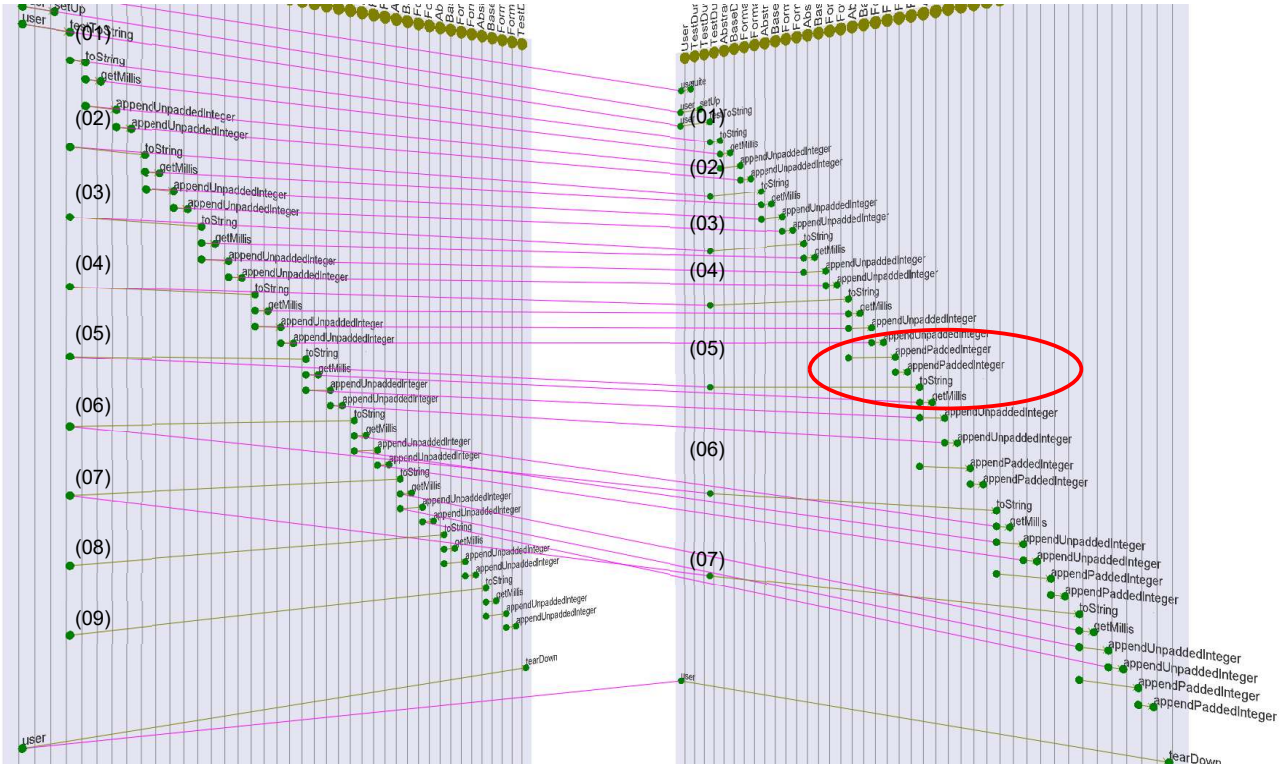

Figure 11: ViBERT on revision pair <r1577, r1576> of Joda-Time. The left plane visualizes revision r1577, and the right one presents r1576.

influence of the changes to the program behavior, and further analyze whether those changes introduce new errors.

As shown in the case study, ViBERT shows the differences between two executions using visual representations. It displays where are the differences and the context of the differences in the execution. In this experiment, we use one revision pair as one example. Other revision pair can be compared in a similar fashion. In software testing, there are many test coverage criteria and metrics. It is worthwhile to note that as Behavior Regression Testing focuses only on comparing method invocations in program executions, not all types of program errors can be identified by behavior regression testing.

7. Related Work

7.1 Program Behavior Comprehension

Numerous researchers have focused on visualizing program executions. Comprehensive surveys of dynamic analysis and software visualization [15][35] are available. Traditionally program behaviors are represented as node-link diagrams in a two-dimensional space. Examples include UML sequence diagrams [10], space-time diagrams, and call graphs [42].

Researchers utilize essential visual elements such as color, shape, and a variety of visual layouts to represent software information. Popular layouts include trees (e.g. tree map [34][39]), tables, graphs and diagrams. TraceVis [30] visualizes executed program instructions by sequentially displaying microprocessor instructions in a 2D plane. It supports queries, different levels of zooming, and annotations on colorful blocks. GAMMATELLA [27] visualizes executions in three levels in 2D: a file level represented in a miniaturized view, a system level using a tree map, and a statement level. MetropoIJS [34] visualizes static and dynamic aspects of largescale program written in Javascript with Treemaps. These approaches, however, focus on the visualization of single execution scenario and do not support a comparison of different program executions. Our study complements previous research by applying existing successful layouts on individual 2D planes in SoftLink.

Apart from 2D visualization, more 3D software visualization environments are built through virtual realities. Metaphors such as cities were used to represent software systems [11][36]. Fittkau *et al*. [20] designed controlled experiments to compare the trace visualization tools EXTRAVIS [14] and ExplorViz in program comprehension tasks. EXTRAVIS uses circular bundling and a massive sequence view, and ExplorViz uses the city metaphors. Scalability in software visualization are commonly addressed by using multiple levels of abstraction [19][41].

7.2 Regression Testing and Visualization

Regression testing aims at uncovering new errors after changes are made to a software system. The increasing size of software systems makes thorough

regression testing a costly endeavor. In addition to traditional test selection and prioritization techniques, researchers have applied visual analytics to regression testing. Engström et al. [18] utilize a heat map (mosaic visualization) to show test history and test covered items. Chen and Ince [13] design a tabular visual representation of regression test results. Different colors are assigned to the blocks on the table and fisheye enlarges the rows of users' interest.

BERT [28][37] is a differential testing technique that identifies behavioral differences between two versions of a program through automatically generated test cases and dynamic analysis. Different from previous testing work, ViBERT compares dynamic program behavior and complements the BERT technique with a visual tool SoftLink.

8. Conclusion and Future work

Regression testing aims at identifying unnoticed faults caused by changes to software. Behavioral regression testing uses dynamic analysis to compare new and old versions of a program in regression testing. This paper has proposed ViBERT, a visual approach to comparing program behavior. Specifically, we had built a 3D environment that allows developers to view the correlations and differences between two versions of program executions. In contrast to other visualization tools, our approach focuses on consecutive behavior comparison. It helps users to interpret the behavioral differences within the context of the executions.

Our future work is to conduct a usability study and gather more feedbacks from users. We also plan to integrate this environment with popular IDEs, such as Eclipse and IntelliJ. More experiments on larger software systems will also be conducted. Another possible extension is that the viewpoint-oriented representation can be enhanced with an eye tracker. The position of the pupil in the eye-tracking controller screen is mapped to that in the visual space. We can use the eye tracker to capture the user's visual focus, and as the viewer's focus moves, the orientations of planes will be automatically updated accordingly.

References

[1] Apache Subversion. http://subversion.apache.org/

[2] AspectJ.https://www.eclipse.org/aspectj/

[3] GraphML. http://graphml.graphdrawing.org/

[4] Joda-Time. http://joda-time.sourceforge.net/

[5] JUnit. http://www.junit.org/

[6] Alexandru C. V., Proksch S., Behnamghader P. and Gall H. C., "Evo Clocks: Software Evolution at a Glance," Working Conference on Software Visualization (VISSOFT), 2019, pp. 12-22.

[7] Ball T. and Eick S. G., "Software Visualization in the Large". Computer, vol. 29,1996, pp. 33-43.

[8] Beck F., Siddiqui H. A., Bergel A. and Weiskopf D., "Method Execution Reports: Generating Text and Visualization to Describe Program Behavior". IEEE Working Conference on Software Visualization, 2017, pp. 1-10.

[9] Behnamghader P., Alfayez R., Srisopha K., and Boehm B., "Towards Better Understanding of Software Quality Evolution through Commit-Impact Analysis". IEEE International Conference on Software Quality, Reliability and Security (QRS), 2017, pp. 251-262.

[10] Briand L.C., Labiche Y., He S., "Automating Regression Test Selection based on UML Designs". Information and Software Technology, Vol. 51, No.1, 2009, pp. 16-30.

[11] Capece N., Erra U., Romano S., and Scanniello G., "Visualising a Software System as a City Through Virtual Reality". Augmented Reality, Virtual Reality, and Computer Graphics, 2017, pp. 319–327.

[12] Castro D. and Schots M., "Analysis of Test Log Information through Interactive Visualizations". International Conference on Program Comprehension, 2018, pp. 156-166.

[13] Chen R. and Ince T., "Visualizing Regression Test Results". http://citeseerx.ist.psu.edu/viewdoc/download;jsessionid=B6B26126F10004A55199CC40E57E896D?doi=10.1.1.366.3623&rep=rep1&type=pdf.

[14] Cornelissen B., Holten D., Zaidman A., Moonen L., Wijk J. J. v., and Deursen A. V., "Understanding Execution Traces Using Massive Sequence and Circular Bundle Views". IEEE International Conference on Program Comprehension, 2007, pp. 49-58.

[15] Cornelissen B., Zaidman A., Deursen A. V., and Moonen L., "A Systematic Survey of Program Comprehension through Dynamic Analysis". IEEE transaction on Software engineering, Vol 35, No. 5, 2009, pp. 684-702.

[16] Cornelissen B., Zaidman A., and Deursen A. V., "A Controlled Experiment for Program Comprehension through Trace Visualization". IEEE Transactions on Software Engineering, Vol.37, No.3, 2011, pp.341-355.

[17] Eick S. C., Steffen J. L. and Sumner, E. E., "Seesoft - a tool for Visualizing Line Oriented Software Statistics". IEEE Transactions on Software Engineering, Vol. 18, No. 11,1992, pp. 957-968.

[18] Engström E., Mantylä M., Runeson P. and Borg M., "Supporting Regression Test Scoping with Visual Analytics". IEEE 7th International Conference on Software Testing, Verification and Validation, 2014, pp. 283-292.

[19] Feng Y., Dreef K., Jones J. A., and Deursen A. V., "Hierarchical Abstraction of Execution Traces for Program Comprehension". International Conference on Program Comprehension, 2018, pp. 86-96.

[20] Fittkau F., Finke S., Hasselbring W. and Waller J., "Comparing Trace Visualizations for Program Comprehension through Controlled Experiments". IEEE International Conference on Program Comprehension, 2015, pp. 266-276.

[21] Lanza M. and Ducasse S., "Polymetric views - a lightweight visual approach to reverse engineering". IEEE Transactions on Software Engineering, Sep. 2003, Vol. 29, No. 9, pp. 782–795.

[22] Jbara A. and Feitelson D. G., "How Programmers Read Regular Code: A Controlled Experiment Using Eye Tracking". IEEE International Conference on Program Comprehension, 2015, pp. 244-254.

[23] Kienle H. M. and Müller H. A., "Requirements of Software Visualization Tools: A Literature Survey". 4th IEEE International Workshop on Visualizing Software for Understanding and Analysis, 2007, pp. 2-9.

[24] Kumar P., Zhang K., Wang Y., "Visualization of Clustered Directed Acyclic Graphs without Node Overlapping". 12th International Conference on Information Visualization, 2008, pp. 38-43.

[25] Mostafa S., Wang X., Xie T., "PerfRanker: Prioritization of Performance Regression Tests for Collection-intensive Software". International Symposium on Software Testing and Analysis, 2017, pp. 23-34.

[26] Nardo D. D., Alshahwan N., Briand L. C., Labiche Y., "Coverage-based Regression Test Case Selection, Minimization and Prioritization: a Case Study on an Industrial System".

Software Testing, Verification, and Reliability, Vol 25, No.4, 2015, pp. 371-396.

[27] Orso A., Jones J. A., Harrold M. J., and Stasko J., "GAMMATELLA: Visualization of Program-execution Data for Deployed Software". 26th International Conference on Software Engineering, pp. 699-700, 2004.

[28] Orso A. and Xie T., "BERT: BEhavioral Regression Testing". International Workshop on Dynamic Analysis, pp. 36-42, 2008.

[29] Reiss S. P., "Visual Representations of Executing Programs". Journal of Visual Languages and Computing, vol. 18, pp. 126-148, 2007.

[30] Roberts J. and Zilles C., "TraceVis: An Execution Trace Visualization Tool". 1st Workshop on Modeling, Benchmarking and Simulation, 2005, pp. 31-38.

[31] Rothermel G., Harrold M. J., "Analyzing Regression Test Selection Techniques". IEEE Transactions on Software Engineering, Vol.22, No.8,1996, pp. 529-551.

[32] Rothermel G., Elbaum S. G., Malishevsky A. G., Kallakuri P., Qiu X., "On Test Suite Composition and Cost-effective Regression Testing". ACM Transaction on Software Engineering and Methodology, Vol. 13, No.3, 2004, pp.277-331.

[33] Rothermel G., "Improving Regression Testing in Continuous Integration Development Environments". keynote at ESEC/SIGSOFT FSE 2018.

[34] Scarsbrook J. D., K.L. KO R., Rogers B., Brainbriage D., "MetropolJS: Visualizing and Debugging Large-Scale JavaScript Program Structure with Treemaps". International Conference on Program Comprehension, 2018, pp.389-392.

[35] Teyseyre A. R. and Campo M. R., "An Overview of 3D Software Visualization". IEEE Transactions on Visualization and Computer Graphics, Vol. 15, No. 1, 2009, pp. 87-105.

[36] Wettel R. and Lanza M., "Codecity: 3D Visualization of Largescale Software". Companion of 30th International Conference on Software Engineering, 2008, pp. 921–922.

[37] Wei J., Orso A., and Xie T., "Automated Behavioral Regression Testing". 3rd International Conference on Software Testing, Verification and Validation, 2010, pp. 137-146.

[38] Stasko J. T., Brown M. H., Domingue J. B., Price B. A., Software Visualization: Programming as a Multimedia Experience, MIT Press, 1998.

[39] Yang Y.L., Zhang K., Wang J.R., and Nguyen Q.V., "Cabinet Tree: An Orthogonal Enclosure Approach to Visualizing and Exploring Big Data". Journal of Big Data, Springer, 2:15, December 2015.

[40] Zhang K., ed., Software Visualization - From Theory to Practice. Boston: Kluwer Academic Publishers, 2003.

[41] Zhao C., Zhang K., Hao J., and Wong W. E., "Visualizing Multiple Program Executions to Assist Behavior Verification". 3rd IEEE International Conference on Secure Software Integration and Reliability Improvement, 2009, pp. 113-122.

[42] Zhao C., Kong J. and Zhang K., "Program Behavior Discovery and Verification: A Graph Grammar Approach". IEEE Transactions on Software Engineering, Vol. 36, No. 3, 2010, pp. 431-448.

[43] Zhao C., Zhang K., and Lei Y., "Abstraction of Multiple Executions of Object-oriented Programs". ACM symposium on Applied Computing, 2009, pp. 549-550.