Assessing Textual Source Code Comparison: Split or Unified?

Alejandra Cossio Chavalier
Departamento de Ciencias Exactas e Ingenierias
Universidad Católica Boliviana "San Pablo"
Cochabamba, Bolivia
cossio@ucbcba.edu.bo

Juan Pablo Sandoval Alcocer
Departamento de Ciencias Exactas e Ingenierias
Universidad Católica Boliviana "San Pablo"
Cochabamba, Bolivia
sandoval@ucbcba.edu.bo

Alexandre Bergel
ISCLab, Department of Computer Science (DCC)
University of Chile
Santiago, Chile
http://bergel.eu

ABSTRACT

Evaluating source code differences is an important task in software engineering. *Unified* and *split* are two popular textual representations supported by clients for source code management. Whether these representations differ in supporting source code commit assessment is still unknown, despite its ubiquity in software production environments.

This paper performs a controlled experiment to test the causality between the textual representation of source code differences and the performance in term of commit evaluation. Our experiment shows that no significant difference was measured. We therefore conclude that both unified and split equally support the source code commit assessment for the tasks we considered.

CCS CONCEPTS

• **Software and its engineering** → *Software maintenance tools*; *Software version control*; • **General and reference** → **Empirical studies**.

KEYWORDS

software evolution, empirical study

ACM Reference Format:
Alejandra Cossio Chavalier, Juan Pablo Sandoval Alcocer, and Alexandre Bergel. 2020. Assessing Textual Source Code Comparison: Split or Unified?. In *Conference Companion of the 4th International Conference on the Art, Science, and Engineering of Programming ('20), March 23–26, 2020, Porto, Portugal*. ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3397537.3398471

1 INTRODUCTION

Comparing source code is an important activity in software engineering. Source code comparison is done for multiple purposes. From resolving merging conflicts when pushing the code to a source code repository to finding the roots of a functional or a performance failure cause by a code modification [9]. For this reason, source code repositories (*e.g.*, GitHub) and integrated development environments (IDE's) provide a number of tools to assist developers in the source code comparison process.

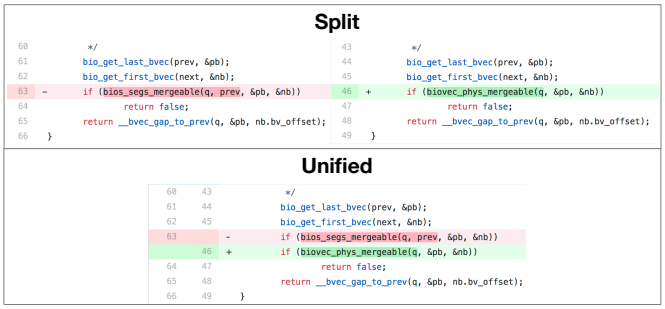


Figure 1: Unified and Split Representations: Source code change done in the function *bio_will_gap* in the GitHub *Linux* project.

Most popular tools visually show the source code difference using two main visual representations: Figure 1 illustrates both representations by showing the source code changes done in the method *bio_will_gap* in the *Linux* GitHub project. In the upper part, we have the *split* view, which shows two snapshots of the source code (before a commit on the left and after the commit on the right). In the lower part, we have the *unified* view, which shows the differences in the same textual panel. Although these two representations are widely used in practice, little is known about which one is more suitable for code comparison. Such textual representations are frequently employed to explain a change that introduces a bug or performance regression [1, 2, 10]. However, there is no empirical evidence to help developers and researchers choose a visual representation. Leaving open the following question: *Which of the* split *or* unified *representation is better to show source code differences?*

In this paper, we present a controlled experiment to answer this question. We measure the performance of practitioners in contrasting source code differences using both tools. We request participants to answer twenty true/false questions about four source code method modifications. Two methods using the unified representation and two with the split representation. Our results show that the visual representation does not impact on the correctness, the completion time, and the cognitive load of analyzing source code changes.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
'20, March 23–26, 2020, Porto, Portugal
© 2020 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-7507-8/20/07…$15.00
https://doi.org/10.1145/3397537.3398471

2 VISUALIZING SOURCE CODE DIFFERENCES

Both the split and unified representations are commonly used to compare any textual files revision. In this paper, we limit ourselves to assess source code revisions. This section, describes both visual representations provided by GitHub to compare software versions.



Figure 2: Unified Representation: Source code changes done in the method *ext4_setattr* in the *Linux* GitHub project.

2.1 Unified Representation

The unified view shows updated and existing content together in a single view. The content of both versions are merged in a single one that mainly highlights the line difference between versions. Normally, the merge algorithm determines the smallest set of line deletions and insertions to transform from one file to another. For instance, Figure 2 shows the unified representation of merging two versions of the method *ext4_setattr*. Red lines represent deleted lines and green lines represent added lines in the new version.

2.2 Split Representation

Similarly to the unified representation, the split representation highlights the smallest set of lines that we need to delete and add a revised source code. However, in this case the content of both versions are not merged into one single view. They are instead showed side by side. Since the number of lines may change from one version to another, a number of empty lines are added in both sides to synchronize each side. Figure 3 shows the same method *ext_4setattr* but this time using a split representation. Note that both representations use the same algorithm to compute the line difference, and the same colors to highlight line additions and deletions.

2.3 Discussion

Depending on the changes obtained from the differencing algorithm, the language syntax and the visual representation may make source code comparison difficult. For instance, consider the the previous example of the modification done in the method *ext4_setattr* (Figure 3 and Figure 2), in particular, the comment block in the source code. It has two sentences in one version and one sentence was deleted in the newer version. However, in the unified representation the first sentence appears twice, one in highlighted with red and the second one with green. On the other hand, the split representation highlights the sentences in the same way that the unified view, the sentence appears in the left and right side and the sentence is positioned in the same Y axis in both sides. Other case is the method call $ext4_wait_for_tail_page_commit(inode)$. The line remains the same in both versions, but it is also flagged with red and green, and in this case the split representation shows this line in different levels. Forcing practitioners manually map these lines during the code comparison.

3 EXPERIMENT DESIGN

3.1 Research Questions & Hypotheses

We design a methodology to answer the following questions:

- **Q1:** Does the visual representation (unified or split) impact the *correctness* of comparing two versions of the same method?
- **Q2:** Does the visual representation (unified or split) impact the *time needed* to compare two versions of the same method?
- **Q3:** Does the visual representation (unified or split) impact the *cognitive load* to compare two versions of the same method?

The null hypotheses and alternative hypotheses corresponding to the three questions are summarized in Table 1.

Table 1: Null and alternative hypotheses

Null hypothesis	Alternative Hypothesis
$H1_0$: The visual representation does not impact the correctness of comparing two versions of the same method	$H1$: The visual representation impacts the correctness of comparing two versions of the same method
$H2_0$: The visual representation does not impact the time needed to compare two versions of the same method	$H2$: The visual representation impacts the time needed to compare two versions of the same method
$H3_0$: The visual representation does not impact the cognitive load needed to compare two versions of the same method	$H3$: The visual representation impacts the cognitive load needed to compare two versions of the same method

3.2 Participants

We pick 24 participants all of them are second year students in computer science, at the Bolivian Catholic University. Participants did three programming courses using the *C/C++* programming language. Participant's *C/C++* experience ranges from 1 to 3 years, and their age ranges from 18 to 23 years. Six participants were women and only six participants had previous experienced with source code comparison tools.



Figure 3: Split Representation: Source code changes done in the method *ext4_setattr* in the *Linux* GitHub project.

3.3 Treatments

We use two visual representations: unified and split. Both implementations are provided by GitHub and are described in previous sections. For the experiment, participants do not use GitHub directly, instead they use a printed sheet gathered from GitHub.

3.4 Code under Study

We provide each participant four method modifications. Participants have to analyze two modifications using a unified representation and two modifications with split representation. The method modifications were collected from the linux GitHub repository. We randomly choose four methods using which meet the following criteria: their size is large enough to be printed in only one paper sheet and have a relative good mix of unmodified lines, modified lines, added lines and deleted lines.

3.5 Work Session

Each participant evaluated four method modifications using both treatments, two methods with one and two with the other. We defined the following work session:

- *Demographic Form* – We ask participants to fill a form with basic demographic questions.
- *Tutorials* – We provide the same tutorial of both representations (unified and split) to all participants.
- *Task1* – We request participants to use a visual representation (assigned randomly) to answer ten true/false questions for each of the two modification methods.
- *TLX Form* – The NASA Task Load Index (TLX) is a technique to assess how an effort is perceived[1]. We use TLX as a proxy to measure cognitive load. Participants fill a TLX Form with 5 questions about mental, physical, temporal demand; their performance, effort, and frustration. TLX Form is a widely used technique for measuring subjective mental workload [6].
- *Task2* – We request participants to use a visual representation (different that the one use in *Task1*) to answer ten true/false questions for each of the two modification methods.
- *TLX Form* – Participants fill a TLX Form with 5 questions.
- *Open Questions* – We ask participants an open question: *Do you have any additional comment about the task and the representations?*

3.6 Randomization

We randomly assign each participant to a treatment order, and the method modifications to analyze. We also take care of balancing this assignation, in a way that half participants first use the unified representation and the other half start with the split. We also carefully assign the method modifications, so we have a good balance order between method modifications and treatments.

3.7 True/False Questions

The true/false questions were done based in the four method modifications under analysis. These questions describe specific small changes about the method that may or not be true. The method modifications and the true/false questions are available online [2]. For instance, if a variable was deleted, if the particular control structure was modified or if a specific method call was added or deleted. By asking these questions, we intent that participants perform analyses of the source code modification and have a deterministic way to assign a score of their performance and correctness in their analysis.

3.8 Metrics

For comparing both visual representations, we use the following metrics:

[1] https://humansystems.arc.nasa.gov/groups/TLX/

[2] https://github.com/AleCossioCh/DiffTool

- *Correctness* – Each participant answers no more than twenty true/-false questions using each visual representation. We define correctness as the ratio between the number of correct answers of each participants and the total of questions for each representation.
- *Completion Time* – Each participant analyzes two method modifications for each representation in order to answering no more than twenty true/false questions. The completion time is the time spent for each participant in analyzing these two methods.
- *Cognitive Load Sum* – Each participant answers five questions about their cognitive load after the analysis of two methods with each representation. The participants are asked to rate their mental, physical, temporal demand, their performance, effort, and frustration by assigning a value between 1 to 21 [6]. The Cognitive Load Sum is the total of the score of the five questions, therefore it varies between 5 and 105.

4 RESULTS

4.1 Correctness

Figure 4 (left side) presents the box plot of the correctness of each representation. By using the split representation participants have an average of 87% of correctness and 85% when using the unified.

The difference in terms of precision is not significant. We run the Mann-Whitney test (two-tailed), we obtained: P = 0.5447, Mann-Whitney U = 258.5. We therefore conclude that the difference is not significant since we do not have $P < 0.05$. As a consequence, we cannot reject the null-hypothesis.

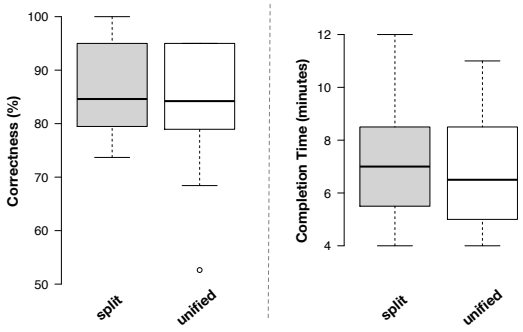


Figure 4: Correctness and Completion Time

4.2 Completion Time

Figure 4 (right side) gives the box plot of the completion time of each representation. Participants use seven minutes on average to complete the task with both visual representations. As Figure 4 illustrates, there is no difference between the completion time.

Similarly than for the precision, *the difference in terms of completion time is not significant.* We run the Mann-Whitney test (two-tailed), we obtained: P = 0.7216, Mann-Whitney U = 270.5. We therefore conclude that the difference is not significant since we do not have $P < 0.05$. As a consequence, we cannot reject the null-hypothesis.

4.3 Cognitive Load

Figure 5 summarizes the answers of the five questions in the TLX form of all participants. It shows that there is not a visual difference between box plots in all questions.

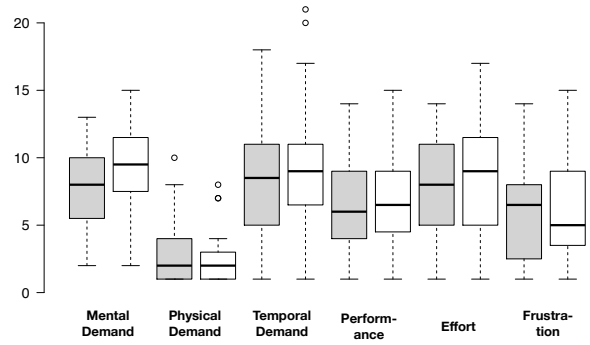


Figure 5: TLX Summary

Figure 6 gives the results of the *TLX Sum* metrics. It shows that there is a small difference between both representations. We cannot establish a strong affirmation because the sum may miss lead to individual results in each questions as we can see in Figure 5.

The difference in terms of cognitive load, measured at the TLX Sum is not significant. We run the Mann-Whitney test (two-tailed), we obtained: P = 0.4456, Mann-Whitney U = 250.5. We therefore conclude that the difference is not significant since we do not have $P < 0.05$. As a consequence, we cannot reject the null-hypothesis.

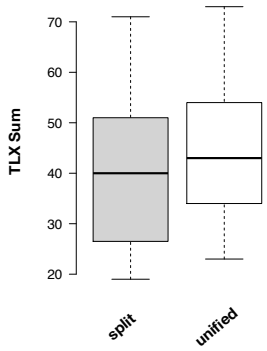


Figure 6: TLX Total Sum

4.4 Feedback

Similar to previous results, there is not a clear tendency between participants about which of the two treatments is the most difficult to use. However, six participants commented that the unified representations seems to be more useful while analyzing short methods. One participant mentioned that the unified representation is more useful to see deleted lines. Other participant mentioned that the split representation forced him to move his head (left to right) to resolve the tasks.

5 DISCUSSION & THREATS TO VALIDITY

5.1 Internal Validity

Participant Experience. Six participants have previous experience with comparison tools. Therefore, there is a chance that they perform better than the other participants. To reduce this possibility, the tasks and method analysis order was assigned randomly.

Learning Effect. Participants performance increase while they analyze the method modifications. To reduce this threat we randomly assign the order of the tasks. In a way that half of participants start the analysis using the unified representation and the other half using the split representation.

True/False Questions. The questions were designed in a way that participants have to analyze the difference to answer them and quantify the correctness of the activity. Although, we could use more open questions (*i.e.,* describe the changes done in this method), then the answers could be qualitative and subjective, making it more difficult to compare between approaches.

5.2 External Validity

Code under Study. We pick medium size method modifications to be able to print them in a sheet of paper. By using paper sheet we control that participants only use the visual insights of the representations to resolve the task. In addition, it controls that all participants have the same font size, and avoid scroll events that may also influence participants performance. Even though, we randomly pick the four methods under analysis with a good mix of changes. We are not sure how the participants performance may vary with larger methods.

Programming Language. Our experiment focuses on the *C/C++* programming language. Although there are many programming languages that were inspired by the *C/C++* syntax. The syntax of the programming language may also influence the performance of the task. As future work, we plan to replicate our experiment in other languages including JavaScript and Python, to see how the effect of the syntax on the comparison activity.

Participants. The experiment was conducted with students from the Bolivian Catholic University. Therefore, the results may be representative only in this academic context. As future work, we plan involve practitioners with industrial experience in our experiment.

Method Modifications. Our experiment was designed to evaluate the performance of comparing two versions of a same method. However, depending of the language and the project, there may be another kind of code changes. For instance, changes in the configuration settings, instance variables, or class hierarchy.

6 RELATED WORK

Diverse approaches have been proposed to improve the source code differencing algorithms [3–5, 7, 8]. For instance, Falleri *et al.* [3] introduced a a new algorithm for source code differencing, which instead of focusing on adding and deleting lines, they measure the difference at the abstract syntax tree granularity. As we see in Section 2, the comparison strategy may also affect the way we highlight the code difference in the visualization. Although there are a variety of differencing strategies, popular IDE's and code repositories mainly provide the split and unified representation for code comparison.

Our work does not compare the source code differencing algorithms. Instead, our motivation is about understanding whether the visual representation affects the comparison analysis. As far as we know, this is the first empirical study that compare practitioners performance while comparing source code differences.

7 CONCLUSION

This paper presents a controlled experiment run with 24 participants to evaluate whether or not the visual representation, unified and split, affects the code comparison between two versions of the method. Our results shows that the visual representation does not impact the correctness, and completion time of the code comparison analysis. However, participants total cognitive load is lower while using the split representation, but this difference is not significative.

ACKNOWLEDGMENTS

We want to thank LAM Research, Semantics S.R.L. and Object Profile for their effort and support.

REFERENCES

[1] Juan Pablo Sandoval Alcocer and Alexandre Bergel. 2015. Tracking Down Performance Variation Against Source Code Evolution. *SIGPLAN Not.* 51, 2 (Oct. 2015), 129–139.

[2] Guillermo de la Torre, Romain Robbes, and Alexandre Bergel. 2018. Imprecisions Diagnostic in Source Code Deltas. In *Proceedings of the 15th International Conference on Mining Software Repositories* (Gothenburg, Sweden) *(MSR '18)*. ACM, New York, NY, USA, 492–502. https://doi.org/10.1145/3196398.3196404

[3] Jean-Rémy Falleri, Floréal Morandat, Xavier Blanc, Matias Martinez, and Martin Monperrus. 2014. Fine-grained and Accurate Source Code Differencing. In *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering* (Vasteras, Sweden) *(ASE '14)*. ACM, New York, NY, USA, 313–324. https://doi.org/10.1145/2642937.2642982

[4] B. Fluri, M. Wuersch, M. PInzger, and H. Gall. 2007. Change Distilling:Tree Differencing for Fine-Grained Source Code Change Extraction. *IEEE Transactions on Software Engineering* 33, 11 (Nov 2007), 725–743. https://doi.org/10.1109/TSE.2007.70731

[5] V. Frick, C. Wedenig, and M. Pinzger. 2018. DiffViz: A Diff Algorithm Independent Visualization Tool for Edit Scripts. In *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. 705–709. https://doi.org/10.1109/ICSME.2018.00081

[6] Sandra G Hart and Lowell E Staveland. 1988. Development of NASA-TLX (Task Load Index): Results of empirical and theoretical research. *Human mental workload* 1, 3 (1988), 139–183.

[7] James W. Hunt. 1975. An Algorithm for Differential File Comparison.

[8] J. I. Maletic and M. L. Collard. 2004. Supporting source code difference analysis. In *20th IEEE International Conference on Software Maintenance, 2004. Proceedings.* 210–219. https://doi.org/10.1109/ICSM.2004.1357805

[9] J. P. Sandoval Alcocer, A. Bergel, S. Ducasse, and M. Denker. 2013. Performance evolution blueprint: Understanding the impact of software evolution on performance. In *2013 First IEEE Working Conference on Software Visualization (VISSOFT)*. 1–9. https://doi.org/10.1109/VISSOFT.2013.6650523

[10] Juan Pablo Sandoval Alcocer, Alexandre Bergel, and Marco Tulio Valente. 2016. Learning from Source Code History to Identify Performance Failures. In *Proceedings of the 7th ACM/SPEC on International Conference on Performance Engineering* (Delft, The Netherlands) *(ICPE '16)*. ACM, New York, NY, USA, 37–48. https://doi.org/10.1145/2851553.2851571