Aiding Developer Understanding of Software Changes via Symbolic Execution-based Semantic Differencing

Johann Glock
University of Klagenfurt
Klagenfurt, Austria
johann.glock@aau.at

ABSTRACT

According to a recent observational study, developers spend an average of 48% of their development time on debugging tasks. Approaches such as equivalence checking and fault localization support developers during debugging tasks by providing information that enables developers to more quickly identify and deal with unintended changes in program behavior. The accuracy and runtime performance of these approaches have seen continuous improvements throughout the years. However, the outputs of existing tools are often difficult to understand for developers due to a lack of context information and result explanations. Our goal is to improve upon this issue by developing a new equivalence checking approach that (i) is at least as accurate as existing approaches but (ii) provides more detailed descriptions of identified behavioral / semantic differences and (iii) presents these results in a way that is useful for developers, thus aiding developer understanding of equivalence checking results and corresponding software changes.

KEYWORDS

Program Comprehension, Semantic Differencing, Equivalence Checking, Symbolic Execution

ACM Reference Format:
Johann Glock. 2024. Aiding Developer Understanding of Software Changes via Symbolic Execution-based Semantic Differencing. In *2024 IEEE/ACM 46th International Conference on Software Engineering: Companion Proceedings (ICSE-Companion '24), April 14–20, 2024, Lisbon, Portugal.* ACM, New York, NY, USA, 3 pages. https://doi.org/xx.xxxx/xxxxxxx.xxxxxxx

1 MOTIVATION

Developers spend a lot of time debugging. In fact, developers self-report that 20-60% of their time is spent on debugging tasks [6]. A recent observational study of developers showed similar results, finding that debugging constitutes an average of 48% of total development time [3]. Furthermore, debugging not only takes place during dedicated bug fixing sessions but also takes up 40% of developer time during programming sessions dedicated to the implementation of new features [3]. These findings suggest that reducing the time required to perform debugging tasks promises significant cost savings and would free up developer resources for other purposes.

Various approaches have been developed throughout the years that aim to support developers during debugging by helping them more quickly understand and deal with source code changes that cause unexpected — and perhaps unintended — changes in program behavior. For example, equivalence checking informs developers whether the input-output behavior of a program has changed across versions [23], fault localization reports lines of code that might be the cause of unintended behaviors [27], and automated program repair aims to automatically fix programs that are crashing or otherwise failing tests after a modification has taken place [18, 20].

While the effectiveness and runtime performance of such approaches have seen continuous improvements throughout the years, developers criticize that existing tools do not provide enough context information to be useful to them [16, 22]. For example, equivalence checking tools generally only output whether two programs are non-/equivalent [4, 14], fault localization tools only provide line numbers of potentially fault-inducing lines of code [27], and program repair tools only output patches that fix identified crashes or test failures [19, 28]. However, neither approaches provide any further evidence or explanations to support their results, which commonly causes developers to disregard the provided outputs because they feel like they cannot understand or trust them [21, 22, 26].

2 PROBLEM STATEMENT

Based on the findings presented above, we hypothesize that developers' understanding of program changes and trust of corresponding tool outputs can be improved if these outputs are enriched with additional context information and result explanations. More specifically, our goal is to develop a new equivalence checking approach that (i) is at least as accurate as existing approaches but (ii) provides more detailed descriptions of identified behavioral / semantic differences and (iii) presents these results in a way that is useful for developers, thus aiding developers' change understanding. Toward this goal, we aim to answer the following research questions:

- **RQ1** How can we exploit symbolic execution to create an equivalence checking approach that is at least as accurate as existing approaches while providing more detailed context information and result explanations?
- **RQ2** How should the collected information about software changes (i.e., equivalence checking results, context information, result explanations) be presented to developers in order to be most useful for them?
- **RQ3** To which degree does the provided information affect the speed, reliability, etc. with which developers are able to reason about software changes?

In the remaining sections, we describe our research approach including preliminary results and finish with expected contributions.

ICSE-Companion '24, April 14–20, 2024, Lisbon, Portugal
© 2024 Copyright held by the owner/author(s).
ACM ISBN xxx-x-xxxx-xxxx-x/xx/xx.
https://doi.org/xx.xxxx/xxxxxxx.xxxxxxx

3 RESEARCH APPROACH

To answer the stated research questions, we split the corresponding research work into five work packages (WP1–WP5): two for RQ1, two for RQ2, and one for RQ3. These work packages are as follows.

3.1 Developing and Evaluating the Semantic Differencing Approach (RQ1)

Semantic Differencing (WP1): The goal of this work package was to develop an approach for the computation of equivalence checking results and any additional information that is required to explain these results to developers. The approach that we have developed for this primarily relies on symbolic execution [15] which has proven to be applicable to a wide range of software analysis tasks [5]. Desirable properties of symbolic execution include its ability to provide summaries of program behavior which can serve as the basis of result explanations, and its underapproximation of program behavior which avoids false-positive results that are often a source of developer frustration [22, 23]. The prototype implementation of our approach is publicly available on GitHub[1].

Benchmarking (WP2): In this work package, we evaluated the effectiveness and efficiency of the equivalence checking / semantic differencing approach developed in WP1. To ensure an unbiased evaluation of our approach, we applied it to the ARDiff [4] benchmark to compare its runtime performance and the correctness of the reported equivalence checking results to three state-of-the-art equivalence checking approaches: ARDiff [4], DSE [23], and PRV [7]. The results of this evaluation, which are currently undergoing peer review, demonstrate that our approach correctly classifies more cases in the ARDiff benchmark than the three existing approaches, albeit at the cost of moderate runtime increases. A preprint of our results is available on arXiv [11] and a replication package that contains all evaluation scripts as well as the raw and processed evaluation data is available on Zenodo [12].

3.2 Developing Appropriate Visualizations of the Semantic Differencing Data (RQ2)

Data Visualization (WP3): Existing studies have shown that source code analysis results are most useful to developers when presented inside of developers' IDEs [2, 17]. Consequently, the goal of this work package is to develop an IDE plugin for a commonly used IDE such as IntelliJ[2] or VS Code[3] that presents the data computed by our approach in a way that is appropriate for developers. Initial representations of the data will be based on the findings of existing user studies in related domains such as general program understanding [13, 24] as well as our own experiences and intuitions as developers. The representations will later be refined using developer feedback from the pilot study conducted in WP4.

Pilot Study (WP4): The goal of WP4 is to conduct a pilot study with around 5 developers to collect initial feedback for the IDE plugin developed in WP3. For each study participant, we will provide a demonstration of our plugin's features, have them complete a series of change understanding tasks with our plugin while thinking aloud [25], and then gather more in-depth feedback through a semi-structured interview [1]. Observations from the change understanding tasks as well as suggestions and criticism from the interview sessions will be used to improve our first IDE plugin prototype, thus making it more useful for a larger range of developers. Task completion data, interview guides, etc. will be made publicly available to the research community.

[1]https://github.com/glockyco/PASDA
[2]https://www.jetbrains.com/idea/
[3]https://code.visualstudio.com/

3.3 Evaluating the Usefulness of the Semantic Differencing Data (RQ3)

User Study (WP5): To evaluate the usefulness of the improved prototype created in WP4, we plan to conduct a user study with 20-30 developers. User study sessions will follow a similar structure as in the pilot study, consisting of a tool demonstration followed by the completion of change understanding tasks and a concluding interview. However, for the change understanding tasks, we will ask developers to not only use our own tool, but to also complete some tasks with an existing tool for source code differencing such as git-diff[4], IJM [9], GumTree [8], or ChangeDistiller [10]. By following this approach, we will be able to compare our tool to the state-of-the-art both quantitatively (via task completion times, accuracy, etc.) as well as qualitatively (via interview feedback).

4 EXPECTED CONTRIBUTIONS

In our research, we will adhere to open science principles. As mentioned throughout the previous sections, we therefore plan to make all created research prototypes, interview guides, raw and processed data, etc. publicly available. In particular, we expect to make the following contributions throughout our research work:

- **C1** an **approach** for equivalence checking of software programs that provides more information about behavioral / semantic differences than existing approaches (WP1),
- **C2** an open source **prototype** that implements the computation of the raw semantic differencing data as a standalone Java application (WP1).
- **C3** an open source **prototype** that implements the processing and visualization of the semantic differencing data as an IDE plugin (WP3),
- **C4** **benchmarking results** that compare the runtime requirements and accuracy of our approach to state-of-the-art equivalence checking approaches (WP2),
- **C5** **experimental results** that compare how quickly and accurately developers are able to complete change understanding tasks when using our IDE prototype vs. (prototypes of) state-of-the-art tools (WP4 + WP5),
- **C6** **developer feedback** that compares the perceived usefulness of our IDE prototype to (prototypes of) state-of-the-art tools (WP4 + WP5).

Through these contributions, we expect to provide reproducible evidence for (or against) our hypothesis that developers' understanding of program changes as well as trust in equivalence checking outputs can be improved if this information is enriched with additional context information and result explanations.

[4]https://git-scm.com/docs/git-diff

REFERENCES

[1] William C. Adams. 2015. *Conducting Semi-Structured Interviews.* John Wiley & Sons, Inc., Chapter 19, 492–505. https://doi.org/10.1002/9781119171386.ch19

[2] Abdulaziz Alaboudi and Thomas D. LaToza. 2021. Edit - Run Behavior in Programming and Debugging. In *Proceedings of the 2021 Symposium on Visual Languages and Human-Centric Computing.* IEEE, 1–10. https://doi.org/10.1109/VL/HCC51201.2021.9576170

[3] Abdulaziz Alaboudi and Thomas D. LaToza. 2021. An Exploratory Study of Debugging Episodes. arXiv:2105.02162 [cs.SE]

[4] Sahar Badihi, Faridah Akinotcho, Yi Li, and Julia Rubin. 2020. ARDiff: Scaling Program Equivalence Checking via Iterative Abstraction and Refinement of Common Code. In *Proceedings of the 28th Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering.* ACM, 13–24. https://doi.org/10.1145/3368089.3409757

[5] Roberto Baldoni, Emilio Coppa, Daniele C. D'Elia, Camil Demetrescu, and Irene Finocchi. 2018. A Survey of Symbolic Execution Techniques. *Comput. Surveys* 51, 3 (2018), 50:1–50:39. https://doi.org/10.1145/3182657

[6] Moritz Beller, Niels Spruit, Diomidis Spinellis, and Andy Zaidman. 2018. On the Dichotomy of Debugging Behavior Among Programmers. In *Proceedings of the 40th International Conference on Software Engineering.* ACM, 572–583. https://doi.org/10.1145/3180155.3180175

[7] Marcel Böhme, Bruno C. D. S. Oliveira, and Abhik Roychoudhury. 2013. Partition-based Regression Verification. In *Proceedings of the 35th International Conference on Software Engineering.* IEEE, 302–311. https://doi.org/10.1109/ICSE.2013.6606576

[8] 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 International Conference on Automated Software Engineering.* ACM, 313–324. https://doi.org/10.1145/2642937.2642982

[9] Veit Frick, Thomas Grassauer, Fabian Beck, and Martin Pinzger. 2018. Generating Accurate and Compact Edit Scripts Using Tree Differencing. In *Proceedings of the 2018 International Conference on Software Maintenance and Evolution.* IEEE, 264–274. https://doi.org/10.1109/ICSME.2018.00036

[10] Harald C. Gall, Beat Fluri, and Martin Pinzger. 2009. Change Analysis with Evolizer and ChangeDistiller. *IEEE Software* 26, 1 (2009), 26–33. https://doi.org/10.1109/MS.2009.6

[11] Johann Glock, Josef Pichler, and Martin Pinzger. 2023. PASDA: A Partition-based Semantic Differencing Approach with Best Effort Classification of Undecided Cases. arXiv:2311.08071 [cs.SE]

[12] Johann Glock, Josef Pichler, and Martin Pinzger. 2023. *Replication Package for: "PASDA: A Partition-based Semantic Differencing Approach with Best Effort Classification of Undecided Cases".* https://doi.org/10.5281/zenodo.7595851

[13] Jane Hoffswell, Arvind Satyanarayan, and Jeffrey Heer. 2018. Augmenting Code with In Situ Visualizations to Aid Program Understanding. In *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems.* ACM. https://doi.org/10.1145/3173574.3174106

[14] Marie-Christine Jakobs. 2021. PEQcheck: Localized and Context-Aware Checking of Functional Equivalence. In *Proceedings of the 9th International Conference on Formal Methods in Software Engineering.* IEEE, 130–140. https://doi.org/10.1109/FormaliSE52586.2021.00019

[15] James C. King. 1976. Symbolic Execution and Program Testing. *Commun. ACM* 19, 7 (1976), 385–394. https://doi.org/10.1145/360248.360252

[16] Pavneet Singh Kochhar, Xin Xia, David Lo, and Shanping Li. 2016. Practitioners' Expectations on Automated Fault Localization. In *Proceedings of the 25th International Symposium on Software Testing and Analysis.* ACM, 165–176. https://doi.org/10.1145/2931037.2931051

[17] Lucas Layman, Madeline Diep, Meiyappan Nagappan, Janice Singer, Robert DeLine, and Gina Venolia. 2013. Debugging Revisited: Toward Understanding the Debugging Needs of Contemporary Software Developers. In *Proceedings of the 2013 International Symposium on Empirical Software Engineering and Measurement.* IEEE, 383–392. https://doi.org/10.1109/ESEM.2013.43

[18] Claire Le Goues, Michael Pradel, and Abhik Roychoudhury. 2019. Automated Program Repair. *Commun. ACM* 62, 12 (2019), 56–65. https://doi.org/10.1145/3318162

[19] Sergey Mechtaev, Jooyong Yi, and Abhik Roychoudhury. 2016. Angelix: Scalable Multiline Program Patch Synthesis via Symbolic Analysis. In *Proceedings of the 38th International Conference on Software Engineering.* ACM, 691–701. https://doi.org/10.1145/2884781.2884807

[20] Martin Monperrus. 2018. Automatic Software Repair: A Bibliography. *Comput. Surveys* 51, 1 (2018), 17:1–17:24. https://doi.org/10.1145/3105906

[21] Yannic Noller, Ridwan Shariffdeen, Xiang Gao, and Abhik Roychoudhury. 2022. Trust Enhancement Issues in Program Repair. In *Proceedings of the 44th International Conference on Software Engineering.* ACM, 2228–2240. https://doi.org/10.1145/3510003.3510040

[22] Chris Parnin and Alessandro Orso. 2011. Are Automated Debugging Techniques Actually Helping Programmers?. In *Proceedings of the 2011 International Symposium on Software Testing and Analysis.* ACM, 199–209. https://doi.org/10.1145/2001420.2001445

[23] Suzette Person, Matthew B. Dwyer, Sebastian Elbaum, and Corina S. Păsăreanu. 2008. Differential Symbolic Execution. In *Proceedings of the 16th International Symposium on Foundations of Software Engineering.* ACM, 226–237. https://doi.org/10.1145/1453101.1453131

[24] Matúš Sulír, Michaela Bačíková, Sergej Chodarev, and Jaroslav Porubän. 2018. Visual Augmentation of Source Code Editors: A Systematic Mapping Study. *Journal of Visual Languages & Computing* 49 (2018), 46–59. https://doi.org/10.1016/j.jvlc.2018.10.001

[25] M. W. van Someren, Y. F. Barnard, and J. A. C. Sandberg. 1994. *The Think Aloud Method: A Practical Approach to Modelling Cognitive Processes.* Academic Press. https://dare.uva.nl/search?identifier=7fef37d5-8ead-44c6-af62-0feeea18d445

[26] Emily Winter, David Bowes, Steve Counsell, Tracy Hall, Saemundur O. Haraldsson, Vesna Nowack, and John R. Woodward. 2023. How do Developers Really Feel About Bug Fixing? Directions for Automatic Program Repair. *IEEE Transactions on Software Engineering* 49, 4 (2023), 1823–1841. https://doi.org/10.1109/TSE.2022.3194188

[27] W. Eric Wong, Ruizhi Gao, Yihao Li, Rui Abreu, and Franz Wotawa. 2016. A Survey on Software Fault Localization. *IEEE Transactions on Software Engineering* 40, 8 (2016), 707–740. https://doi.org/10.1109/TSE.2016.2521368

[28] Jifeng Xuan, Matias Martinez, Favio Demarco, Maxime Clement, Sebastian R. Lamelas Marcote, Thomas Durieux, Daniel Le Berre, and Martin Monperrus. 2017. Nopol: Automatic Repair of Conditional Statement Bugs in Java Programs. *IEEE Transactions on Software Engineering* 43, 1 (2017), 34–55. https://doi.org/10.1109/TSE.2016.2560811