Lightweight Semantic Conflict Detection with Static Analysis

Galileu Santos de Jesus
gsj@cin.ufpe.br
Informatics Center, Federal University of Pernambuco
Recife, Brazil

Paulo Borba
phmb@cin.ufpe.br
Informatics Center, Federal University of Pernambuco
Recife, Brazil

Rodrigo Bonifácio
rbonifacio@unb.br
Computer Science Department, University of Brasilia
Brasilia, Brazil

Matheus Barbosa de Oliveira
mbo2@cin.ufpe.br
Informatics Center, Federal University of Pernambuco
Recife, Brazil

ABSTRACT

Version control system tools empower developers to independently work on their development tasks. These tools also facilitate the integration of changes through merging operations, and report textual conflicts. However, during the integration of changes, developers might encounter other types of conflicts that are not detected by current merge tools. In this paper, we focus on dynamic semantic conflicts, which occur when merging reports no textual conflicts but results in undesired interference—causing unexpected program behavior at runtime. To address this issue, we propose a technique that explores the use of static analysis to detect interference when merging contributions from two developers. We evaluate our technique using a dataset of 99 experimental units extracted from merge scenarios. The results provide evidence that our technique presents significant interference detection capability (F1 Score of 0.50 and Accuracy of 0.60).

CCS CONCEPTS

• **Software and its engineering → Software configuration management and version control systems**.

KEYWORDS

Merge conflicts; Configuration management; Software evolution; Static analysis.

ACM Reference Format:
Galileu Santos de Jesus, Paulo Borba, Rodrigo Bonifácio, and Matheus Barbosa de Oliveira. 2024. Lightweight Semantic Conflict Detection with Static Analysis. 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/10.1145/3639478.3643118

1 INTRODUCTION AND MOTIVATION

Textual (line-based) merge tools identify conflicts when merging code versions that modify the same or consecutive lines. While these conflicts are common and well-studied, they represent just one aspect of the challenges that may arise when integrating code from two developers. As textual merge tools focus simply on combining lines, they aren't able to detect incompatible changes that occur in areas of the code separated by at least a single line.

For instance, textual merge tools also can't detect *dynamic semantic conflict* [1, 3, 4, 6, 7, 9–11] , that happens, for instance, when the changes made by one developer affect a state element that is accessed by code changed by another developer, who assumed a state invariant that no longer holds after merging. In such cases, textual integration is automatically performed generating a merged program, a build is created with success for this program, but its execution reveals unexpected behavior. Following Horwitz et al. [6], we put this more formally by considering dynamic semantic conflicts as *undesired interference*. Interference is defined as follows: separate changes L and R to a base program B interfere when the integrated changes do not preserve the altered behavior of L or R, or the unchanged behavior of B.

As dynamic semantic conflicts, hereafter simply semantic conflicts, can negatively impact development productivity and the quality of software products, researchers [4, 6, 9] have proposed techniques to detect them. In fact, as developer's desire (specification) is hard to capture and is often not available for automated tools, these techniques simply try to detect interference. The techniques based on theorem proving [9] and static analysis with system dependence graphs (SDGs) [1, 2] are computationally expensive. Although an existing technique based on testing [4] has been proven less expensive, it suffers from low recall.

These limitations motivate us to explore lightweight static analyses algorithms for approximating interference when merging changes made by two developers. We conduct an experiment to understand how *accurate* and *computationally efficient* our analyses are for detecting interference. The results show that our analyses have significant interference detection capability, but with potentially significant costs for handling false positives (analysis incorrectly reports interference). In this extended abstract, we briefly introduce the static analysis algorithms and partially detail the results of our experiment.

2 STATIC ANALYSIS ALGORITHMS

We propose a technique that runs static analyses algorithms on the merged version of the code, which we annotate with metadata indicating instructions modified or added by each developer. For each of the four analyses enumerate below, we implemented the algorithms targeting the Java programming language and took advantage of the Soot framework as the underlining infrastructure for Java program analyses.

- **Interprocedural Data Flow (DF:** *def-use* relationships involving Left and Right contributions as *Data Flow* (DF).
- **Interprocedural Confluence (CF):** Interference can also occur when there is no data flow between the integrated changes, but there are data flows from the integrated changes to a common statement.
- **Interprocedural Override Assignment (OA):** we capture situations where state elements updates from one developer are overridden by the other.
- **Program Dependence Graph (PDG):** our implementation was based on the work of [5, 6]. Deals with situations where a contribution from a developer introduces a new control-flow dependency with a statement that another developer contributes to.

Two analyses, OA and CF, were specifically tailored to interference detection in code merging. DF, though based on an SVFA implementation, includes adaptations required for interference detection. Our implementation incorporates new flows to statements such as conditionals and returns, which are not considered by SVFA (and doesn't make sense for standard data flow analysis), but are necessary for detecting interference. PDG has nothing new, we simply apply it for semantic conflict detection.

3 EMPIRICAL ASSESSMENT

We start with open-source Java projects from GitHub. We rely on a number of Java projects and scenarios that appeared in previous studies on semantic merge conflicts: 35 units from [8], 31 from [1], and 30 from [9].

Together with the three new units that first appear in this work, we have a total of 99 experimental units, associated with 54 merge scenarios extracted from 39 projects. At least two researchers manually analyze each unit and check for interference using the definition and conditions that imply interference [6]. We use the results of this analysis to build a ground truth.[1]

For each scenario in our dataset, we build the merge commit version and generate a JAR file without external project dependencies. We then run the algorithms we implemented to detect interference. Finally, we compare the interference ground truth with the logical disjunction of the analyses results, and compute precision, recall, F1, and accuracy metrics. We also summarize and analyze the execution time information in a number of ways.

We depict the confusion matrix of our experiment in Table 1 , which shows that our sample contains 33 units with interference, and 66 without interference. The logical disjunction of our analyses results reports 26 (26.2%) false positives (incorrectly predicted interference) and 13 (13.1%) false negatives (non reported interference), resulting in a F1 Score of 0.50 and an Accuracy of 0.60.

[1]There is a state element x such that Base, Left, and Right compute different values for x; Left (or Right) computes a different value for x compared to both the Base program and the merged version; or Base, Left, and Right compute the same value for x, but the merged version computes a different value.

		Actual		
		Positive	Negative	Total
Predicted	Positive	20	26	46
	Negative	13	40	53
	Total	33	66	99

Table 1: Confusion matrix.

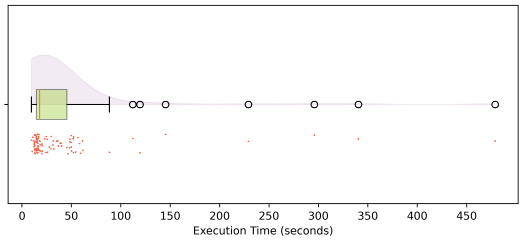

Figure 1: Summary of the analyses execution time

We carry out our experiments using an Ubuntu docker container running on an Intel Xeon Gold 6338 server with 2Tb of RAM, as we run our four analysis 10 times for each unit. Variation across the 10 executions is small, with standard deviation ranging from 0.04s (in a scenario with 11.74s average) to 11.74s (in a scenario with 340.45s average). The median is at 17.8s for executing together the four analyses, including their setup, call graph construction, interference checking, and result reporting (see Figure 1).

4 CONCLUSIONS

We present a technique and a set of static analyses that show significant interference detection capability, and potential for detecting dynamic semantic conflicts. It comes, though, with potentially significant costs for handling false positives. Comparing with previous work [1, 8, 9], our technique shows much better F1 score and recall than the dynamic analysis technique, but with much worse precision. Our precision is comparable to the ones presented by the SDG and theorem proving techniques. The performance results show that our analyses should have much better performance than previous techniques. In summary, besides existing limitations, our experience suggests that our technique is feasible, and might complement dynamic analysis alternatives that aim to detect interference.

ACKNOWLEDGMENTS

For partially supporting this work, we would like to thank INES (National Software Engineering Institute) and the Brazilian research funding agencies CNPq, FACEPE (grants IBPG-0029-1.03/20), and CAPES. We thank Rafael Alves, Léuson da Silva and Vinícius dos Santos for the support when creating our dataset.

REFERENCES

[1] Roberto Souto Maior de Barros Filho. 2017. *Using information flow to estimate interference between same-method contributions.* Master's thesis. Federal University of Pernambuco.
[2] David Binkley, Susan Horwitz, and Thomas Reps. 1995. Program integration for languages with procedure calls. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 4, 1 (1995), 3–35. https://doi.org/10.1145/201055.201056

[3] Yuriy Brun, Reid Holmes, Michael D Ernst, and David Notkin. 2013. Early detection of collaboration conflicts and risks. *IEEE Transactions on Software Engineering* 39, 10 (2013), 1358–1375. https://doi.org/10.1109/TSE.2013.28

[4] Leuson Da Silva, Paulo Borba, Wardah Mahmood, Thorsten Berger, and João Moisakis. 2020. Detecting semantic conflicts via automated behavior change detection. In *2020 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 174–184. https://doi.org/10.1109/ICSME46990.2020.00026

[5] Jeanne Ferrante, Karl J Ottenstein, and Joe D Warren. 1987. The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 9, 3 (1987), 319–349. https://doi.org/10.1145/24039.24041

[6] Susan Horwitz, Jan Prins, and Thomas Reps. 1989. Integrating noninterfering versions of programs. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 11, 3 (1989), 345–387. https://doi.org/10.1145/65979.65980

[7] Danhua Shao, Sarfraz Khurshid, and Dewayne E Perry. 2009. SCA: a semantic conflict analyzer for parallel changes. In *Proceedings of the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*. 291–292. https://doi.org/10.1145/1595696.1595747

[8] Léuson Mário Pedro da Silva. 2022. *Detecting, understanding, and resolving build and test conflicts*. Ph. D. Dissertation. Federal University of Pernambuco.

[9] Marcelo Sousa, Isil Dillig, and Shuvendu K Lahiri. 2018. Verified three-way program merge. *Proceedings of the ACM on Programming Languages* 2, OOPSLA (2018), 1–29. https://doi.org/10.1145/3276535

[10] Wuu Yang, Susan Horwitz, and Thomas Reps. 1992. A program integration algorithm that accommodates semantics-preserving transformations. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 1, 3 (1992), 310–354. https://doi.org/10.1145/131736.131756

[11] Jialu Zhang, Todd Mytkowicz, Mike Kaufman, Ruzica Piskac, and Shuvendu K Lahiri. 2022. Using pre-trained language models to resolve textual and semantic merge conflicts (experience paper). In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*. 77–88. https://doi.org/10.1145/3533767.3534396