Hybrid Regression Test Selection by Synergizing File and Method Call Dependences

Luyao Liu
National University of Defense Technology
College of Computer
Changsha, China
lyliu@nudt.edu.cn

Guofeng Zhang*
National University of Defense Technology
College of Computer
Changsha, China
zhangguofeng16@nudt.edu.cn

Zhenbang Chen†
National University of Defense Technology
College of Computer
Changsha, China
zbchen@nudt.edu.cn

Ji Wang*
National University of Defense Technology
College of Computer
Changsha, China
wj@nudt.edu.cn

ABSTRACT

Regression Test Selection (RTS) minimizes the cost of regression testing by selecting only the tests affected by code changes. We introduce a novel hybrid RTS approach, JcgEKS, which enhances Ekstazi by integrating static method call graphs. It combines the advantages of both dynamic and static analyses, improving the precision from class level to method level without sacrificing safety and reducing the overall time. Moreover, it safely addresses the challenge of handling callbacks from external libraries at the static method-level RTS. To evaluate the safety of JcgEKS, we insert log statements into code patches and monitor the relationship between the test and the output during execution to gauge the test's impact accurately. The preliminary experimental results are promising.

CCS CONCEPTS

• **Software and its engineering** → **Software testing and debugging**.

KEYWORDS

Regression test selection, regression testing, change-impact analysis

ACM Reference Format:
Luyao Liu, Guofeng Zhang, Zhenbang Chen, and Ji Wang. 2024. Hybrid Regression Test Selection by Synergizing File and Method Call Dependences. In *Companion Proceedings of the 32nd ACM International Conference on the Foundations of Software Engineering (FSE Companion '24), July 15–19, 2024, Porto de Galinhas, Brazil.* ACM, New York, NY, USA, 2 pages. https://doi.org/10.1145/3663529.3663872

*Also with State Key Laboratory of High Performance Computing, National University of Defense Technology
†Corresponding Author

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.
FSE Companion '24, July 15–19, 2024, Porto de Galinhas, Brazil
© 2024 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 979-8-4007-0658-5/24/07
https://doi.org/10.1145/3663529.3663872

1 INTRODUCTION

Regression Test Selection (RTS) [5] enhances the regression testing process by selectively executing tests affected by code changes. It comprises dynamic [2, 7], which tests previous versions to collect dependencies, and static [3, 6], which infers dependencies using static analysis. RTS is categorized into control flow, method, file, and module levels based on dependency granularity.

Recent studies indicate that file-level RTS (FRTS) has the shortest end-to-end time compared to other granularity levels [2, 7]. However, FRTS still suffers from imprecision, sometimes selecting unnecessary tests. On the other hand, fine-grained analysis offers higher precision but is often impractical due to the overhead [4]. For instance, dynamic method-level RTS (MRTS) instruments each method, which introduces substantial overhead. Static MRTS does not impose an additional runtime burden but inherits the limitations of static analysis, *e.g.*, the imprecision and incompleteness caused by advanced language features. Moreover, static analysis may struggle to assess the impact of the dependencies when calling external libraries. Therefore, it is natural to combine different RTS techniques [5, 7] to improve the effectiveness and efficiency further.

In this extended abstract, we present ongoing work of synergizing dynamic FRTS and static MRTS to improve regression testing. Dynamic FRTS is effective in practice and safer than static FRTS. However, it still faces challenges in selecting non-redundant tests due to lacking program semantics, especially for large-scale programs. We can employ lightweight and efficient static analysis to improve dynamic FRTS. Based on this insight, we propose a hybrid RTS approach that leverags the advantages of dynamic FRTS and static MRTS, reducing the end-to-end testing time without sacrificing safety. Besides, the dynamic information collected during dynamic FRTS is also used to improve the precision of the static analysis in static MRTS (*e.g.*, reflection and inheritance).

We have implemented a prototype, called JcgEKS, for Java programs based on Ekstazi (*i.e.*, the state-of-the-art dynamic FRTS tool [2]) and Java-callgraph [1], *i.e.*, a tool for generating static call graphs for Java programs. The preliminary results of applying the prototype on several real-world open-source Java programs indicate the promise of our method.

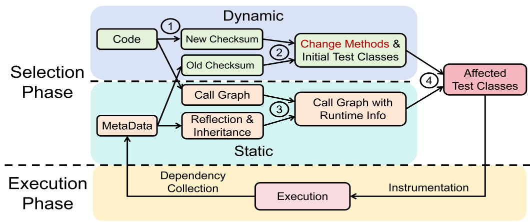

Figure 1: The framework of our hybrid RTS

2 FRAMEWORK

Figure 1 shows our RTS framework, following the procedure of dynamic RTS divided into two phases: selection and execution. In the selection phase, JcgEKS identifies test classes affected by code changes, which are subsequently executed during the execution phase. To collect file dependency information dynamically, JcgEKS instruments the program to record the loaded classes. The collected class dependency information is stored in metadata, which will be used to filter tests for the next revision. The selection phase, as shown in the figure 1, can be described in four steps:

- Calculate the checksum of each class and method (2.1);
- Compare the checksum differences (2.2);
- Generate the enhanced method call graph (2.3);
- Refine the affected test classes at the method level (2.4).

2.1 Checksum Computation

The first step calculats the checksum for each class and method in the new version to monitor code modifications. It identifies the files with altered checksums in the new version and then compares the methods with changed checksums within those files to pinpoint the modified files and the modified methods. This process of identifying changed methods enhances the precision in locating tests impacted by the code changes, thereby improving selection accuracy.

2.2 Test Classes Selection

The second step compares checksum differences at the file level to identify the test classes impacted by code changes. It extracts test classes dependent on the changed files, utilizing the metadata recorded in the previous execution phase. However, this initial identified test classes may be redundant. To enhance accuracy further, the process should be refined by incorporating additional information, such as method dependence.

2.3 Call Graph Generation

The third step constructs a method call graph enriched with runtime information. We will *incrementally* generate a static call graph for the new code version. Besides, using the runtime information such as reflections extracted from metadata, we can improve the precision and completeness of the call graph. Hence, our call graph construction method combines dynamic and static analysis. This step provides a reliable foundation for the subsequent refinement step, ensuring the safety and precision of the final result.

2.4 Test Classes Refinement

The fourth step entails further method-level refinement of initially identified coarse-grained affected test classes from the second step. Each selected test class is individually assessed to determine if it directly or indirectly calls the changed method, using the method call graph generated in the third step. Test classes unaffected at the method level, despite file-level impact, are excluded. After evaluating all the initially selected test classes, the final outcome is the refined set of tests. Recognizing static call graphs' limitations in analyzing external libraries, we consider scenarios where callbacks to project code might occur due to dynamic dispatch.

3 PRELIMINARY EVALUATION

We developed a prototype based on Ekstazi and Java-callgraph. The evaluation uses three metrics: the percentage of chosen test cases, the total testing duration, and the identification of missing tests (*i.e.*, method's safety). The percentage of selected test cases helps gauge the accuracy of our method selection. The total testing duration indicates the time saved by our method overall. To assess the safety of RTS tools, we suggest inserting logging statements into the modified methods and executing all test cases to accurately identify the affected tests through analysis of the execution logs.

We evaluated the effectiveness on 150 revisions across three open-source projects [1] [2] [3] and compared it with three baseline RTS tools (*i.e.*, Ekstazi, STARTS and FineEkstazi). Initial experimental results show that, on average, the number of the test cases chosen by our approach is 46.9%, 27.1%, and 67.9% of those selected by three other tools, respectively. Furthermore, our approach's end-to-end time is 72.7%, 33.1%, and 83.3% of those needed by the baseline tools, respectively. These results indicate the promise of our approach. In addition, our approach and Ekstazi do not miss tests, but STARTS and FineEkstazi miss some tests.

4 NEXT STEP

The subsequent steps involve three key aspects: (1) Evaluation on more benchmarks; (2) Assessing the security of RTS tools; (3) Applying our method to other programs in different languages.

ACKNOWLEDGMENTS

This research was supported by National Key R&D Program of China (No. 2022YFB4501903) and the NSFC Programs (No. 62172429 and 62032024).

REFERENCES

[1] 2024, java-callgraph. [n.d.]. https://github.com/gousiosg/java-callgraph.
[2] Milo Gligoric and et.al. 2015. Practical regression test selection with dynamic file dependencies. *ISSTA 2015* (2015). https://doi.org/10.1145/2771783.2771784
[3] Owolabi Legunsen and et.al. 2017. STARTS: STAtic regression test selection. In *ASE 2017, Urbana, IL, USA, October 30 - November 03, 2017*. 949–954. https://doi.org/10.1109/ASE.2017.8115710
[4] Yingling Li, Junjie Wang, and et.al. 2020. An extensive study of class-level and method-level test case selection for continuous integration. *J. Syst. Softw.* 167 (2020), 110614. https://doi.org/10.1016/J.JSS.2020.110614
[5] Yu Liu, Jiyang Zhang, and et.al. 2023. More Precise Regression Test Selection via Reasoning about Semantics-Modifying Changes. In *ISSTA 2023, Seattle, WA, USA, July17-21, 2023*. https://doi.org/10.1145/3597926.3598086
[6] August Shi and et.al. 2019. Reflection-aware static regression test selection. *PACMPL* 3 (2019), 1 – 29. https://doi.org/10.1145/3360613
[7] Lingming Zhang. 2018. Hybrid regression test selection. In *ICSE 2018, Gothenburg, Sweden, May 27 - June 03, 2018*. 199–209. https://doi.org/10.1145/3180155.3180198

Received 2024-03-15; accepted 2024-04-26

[1] https://github.com/sonyxperiadev/gerrit-events
[2] https://github.com/apache/commons-imaging
[3] https://github.com/apache/commons-codec