Selecting Test Cases based on Similarity of Runtime Information: A Case Study of an Industrial Simulator

Kazumasa Shimari*, Masahiro Tanaka†, Takashi Ishio*, Makoto Matsushita†, Katsuro Inoue‡, Satoru Takanezawa§
* Graduate School of Science and Technology, Nara Institute of Science and Technology, Nara, Japan
Email: {k.shimari, ishio}@is.naist.jp
† Graduate School of Information Science and Technology, Osaka University, Osaka, Japan
Email: {m-tanaka, matusita}@ist.osaka-u.ac.jp
‡ Faculty of Science and Technology, Nanzan University, Aichi, Japan. Email: inoue599@nanzan-u.ac.jp
§ Daikin Industries, Ltd., Osaka, Japan. Email: satoru.takanezawa@daikin.co.jp

***Abstract*—Regression testing is required to check the changes in behavior whenever developers make any changes to a software system. The cost of regression testing is a major problem because developers have to frequently update dependent components to minimize security risks and potential bugs. In this paper, we report a current practice in a company that maintains an industrial simulator as a critical component of their business. The simulator automatically records all the users' requests and the simulation results in storage. The feature provides a huge number of test cases for regression testing to developers; however, their time budget for testing is limited (i.e., at most one night). Hence, the developers need to select a small number of test cases to confirm both the simulation result and execution performance are unaffected by an update of a dependent component. In other words, the test cases should achieve high coverage while keeping diversity of execution time. To solve the problem, we have developed a clustering-based method to select test cases, using the similarity of execution traces produced by them. The developers have used the method for a half year; they recognize that the method is better than the previous rule-based method used in the company.**

***Index Terms*—Test Selection, Dynamic Analysis, Software Dependency, Clustering**

I. Introduction

Regression testing is performed to check the changes in behavior before and after a software change. The cost of regression testing is a significant problem for software developers because they have to repeatedly perform regression testing even when the software is not directly changed.

Daikin Industries, Ltd., a collaborating company, developed an industrial thermodynamic and fluid mechanic simulator. The users of this simulator are engineers of the company who design new products. The simulator accepts various parameters specifying a structure of an electromechanical product and simulates the physical behavior; the users can estimate the efficiency of the structure without constructing an actual prototype. The simulator is now recognized as one of the most important simulators for their product development.

Software developers in the company periodically update the simulator and its dependencies and then perform regression testing. For effective regression testing, the simulator automatically records all the users' requests and the simulation results in storage. When the developers update the simulator, they execute some simulations again and compare the new results with the recorded results. Instead of all the recorded requests, the software developers selected a small subset of test cases that cover available simulation components so that they can finish their test within a limited time budget.

Although the software developers believed their strategy was sufficient, they failed to detect an incompatibility between JDK versions. They updated the JDK version from Oracle JDK 8 to AdoptOpenJDK 11 and performed regression testing as usual. However, a simulator failure occurred a few months after the regression testing. The cause was a change in the behavior of the Java standard library, which prevented several simulations from running properly. Even though the test cases have been selected to cover simulation components, they did not cover a corner case in a wide variety of behavior of the simulation components. Although test case selection is a popular topic in the software testing community, we could not identify a suitable method to select a small number of effective test cases that can be executed within at most one night from a large number of test cases.

In this paper, we report our current regression testing method developed for this simulator. The objective of the method is to select test cases satisfying the following concerns of software developers in the company:

- The time budget for regression testing is limited. They would like to select only a small number of test cases.
- The selected test cases should exercise functional components as much as possible. In other words, code coverage should be high.
- The selected test cases should include both short simulations and long simulations reflecting regular usage patterns so that they can check the performance of the

simulator after an update.

To address the concerns, we developed a method to select test cases using their runtime information. Our method employs k-means clustering to identify similar executions and select representative test cases from each of the clusters. As the parameter k corresponds to the number of test cases, developers can easily specify a value according to their time budget. The developers have used the method to dependency update tasks for a half year; they successfully update dependent components.

In the remainder of the paper, Section II describes our test case selection method. Then, Section III describes the case study, and Section IV applies our method to an industrial simulator. Section V describes the related work. Finally, Section VI concludes the paper and describes future work.

II. Our Test Case Selection Method

Our method takes as input a set of test cases and a parameter k that specifies the number of test cases to be selected. Our method consists of two steps. Figure 1 shows an overview of our method. In STEP Ⅰ, we collect runtime information for each test case by executing test cases with an execution trace recorder. This step is performed in advance. In STEP Ⅱ, we classify test cases based on the similarity of their runtime information and select their representative test cases.

A. STEP I: Collect Runtime Information

At first, we collect runtime information for each test case. Each test case for the simulator system is a pair of a user request and a simulation result. The user request specifies the type and combination of the circuit parts in the simulation. The simulation result includes thermal efficiency and their execution time, and scripts that send and receive these kinds of information to and from the simulator. Developers can use the scripts to reproduce the simulation result.

Our method translates the runtime behavior of a test case into a vector of integers with test ID. Test ID is specified sequentially based on the timestamps original users' test cases are executed. Each element of a vector is the number of occurrences of a runtime event within the execution of the test case. To collect runtime events from test cases, we use a modified version of our tool SELogger version 0.2.3, which is an execution trace recorder publicly available on GitHub [8]. The tool can record the execution order of Java bytecode instructions such as method call and local variable access. We use the tool with the *freq* mode, which assigns IDs to each of the instructions and counts their occurrences in an execution. We simply use a vector produced by the tool as a feature vector of the test case.

To extract only relevant runtime events to the simulator, we exclude third party libraries from logging. This is because the target simulator is a web application running on Tomcat. While the original version of SELogger records all runtime events including the behavior of Tomcat, our version separately collects only runtime events of the simulator. An execution trace is separately collected for each test case.

Figure 1 shows an example of collecting runtime information. The executed instructions, consisting of runtime events and their location, and their execution counts are recorded as an execution trace for each test script. In this example, the loop from line 3 to line 4 is executed three times, so some events are repeatedly executed, such as *Local Variable Load* event, which means the value is read from the variable, at line 4. From execution traces, we create the instruction vectors whose elements are the instruction execution counts in the combination of the event name and location. The instruction execution count is used in addition to the coverage so that the execution characteristics such as a different number of loops with the same coverage can be reflected in the vector. The length of the vector in our method is the number of bytecode instructions in the program.

B. STEP II: Classify Similar Execution

We classify vectors obtained from Step Ⅰ into groups using the k-means clustering method. We adopt the k-means method simply because users can specify the number of test cases to be selected according to their time budget. In addition, the k-means method is less computational complexity than hierarchical clustering. We use Euclidean distance when measuring between vectors. The distance distinguishes a short simulation from a long simulation even if they use the same set of instructions.

After clustering, we select the oldest test case from each cluster as the representative test cases because they are the first test cases that triggered new behaviors of the simulator.

III. Case Study

To understand the usefulness of our method for updating dependencies, we measure the code coverage and diversity of the execution time of test cases selected by our method. We compare our method with three baselines: a variant of our method using 0-1 vectors ignoring the number of occurrences, a random selection method, and a method previously used in the company.

A. Settings

Ideally, we should run all the existing test cases and apply our method. However, the simulator system has already recorded tens of millions of test cases. It is difficult to execute all of them to record the runtime information in a short period of time due to resource constraints. Instead, we collect ten days of test cases for the case study, with a maximum of one thousand test cases per day, and then select representative test cases from the collected cases. The number of collected test cases is 9,612. We decided to use $k = 100$ after a discussion with developers. The developers prefer a small number in order to ensure test cases can be executed within a short time period.

The target simulator under test has 61 classes, 509 methods, and 21,527 lines. Our clustering method took ten minutes on Windows 10 running a Xeon(R) W-2123 @3.60 GHz processor, 32 GB DRAM, and an HDD.

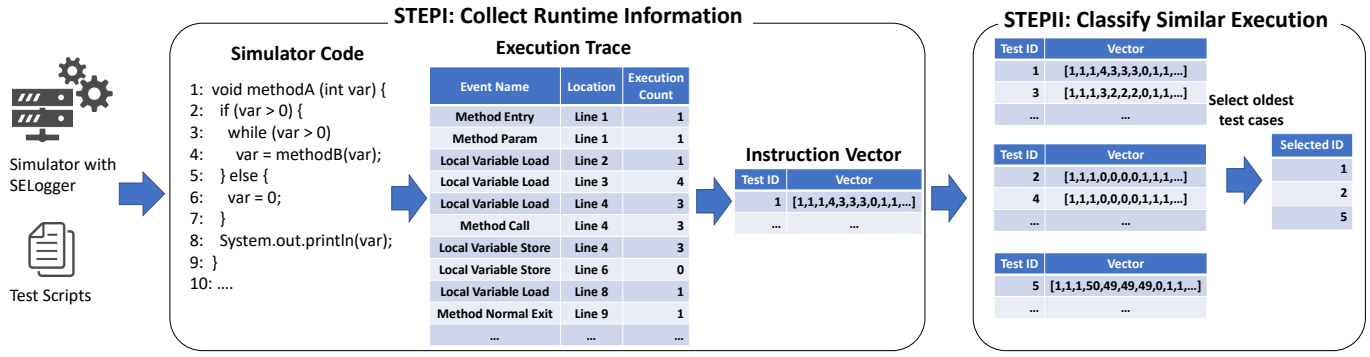

Fig. 1. An Overview of Our Method.

TABLE I
RESULT OF THE COVERAGE.

Test Case Selection Method	Relative Instruction Coverage
Our clustering-based selection	99.76%
Boolean-based selection (Baseline1)	99.88%
Random selection (Baseline2)	65.72%
Component-based selection (Baseline3)	97.73%
All Test Cases Coverage	100.00%

We use three baselines: *Boolean-based selection*, *Random selection* and *Component-based selection*. The Boolean-based selection is a variant of our method using a 0-1 vector, which ignores the number of occurrences of runtime events. It is introduced to evaluate the importance of execution counts. In Random selection, we arranged the test cases in order when they were executed, and selected to become chronologically random. Component-based selection is the method used by software developers in Daikin Industries, Ltd. We selected test cases so that each simulation component's number of test cases was as equal as developers typically do. In this case study, we select 7 or 8 test cases for each of fourteen simulation components, to become chronologically random and include the oldest one same as our method, 100 in total. For each method, we obtain the code coverage of selected test cases and the distribution of the execution time.

B. Coverage Results

Table I shows the relative coverage of each method. In this research question, we define relative coverage of a selection method as the percentage of the instructions executed by 100 test cases selected by the method divided by the instructions executed by all 9,612 test cases. First, we can see that the accuracy of the Component-based method is much higher than the Random method. This result indicates that the simulation component is highly related to the coverage. Our clustering-based method and the Boolean-based method outperform other baselines. In other words, we can select test cases that cover different execution paths using the runtime information. The result is better than the Component-based method used in the company because the execution path or loop counts differ depending on the parameters, even if the simulation component is the same. On the other hand, the Boolean-based method is slightly better than our clustering-based method. This is because our method may select test cases with different instruction execution counts even if they have the same instructions.

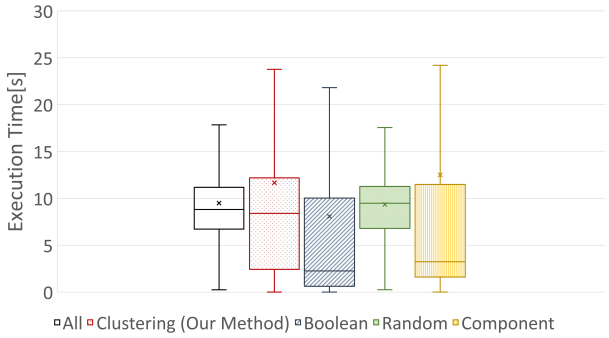

Fig. 2. The Distribution of the Execution Time without Outlier.

C. Execution Time Diversity Results

Figure 2 shows the distributions of the execution time excluding outliers for each method. In this figure, outliers for each method are removed to focus on the distribution, since the outliers for each method are much longer than regular executions. The longest execution of all the test cases is 328 seconds. Our method and the Random selection resulted in similar distributions to the distribution of the original (All) test cases. On the other hand, Boolean-based method and the Component-based method resulted in significantly different distributions from all test cases. By using the information of instruction execution count, we distinguished complex executions that take a long time from those that do not and thus achieved a diverse distribution of execution times.

Results of the Case Study

Our method using runtime information allows selecting test cases with high coverage. By taking the number of occurrences into account, the method keeps the diversity of execution time.

IV. Field Experiment

Based on the case study results, the software developers decided to use our method for their dependency update activity. They updated their JDK from AdoptOpenJDK 11 to Eclipse Temurin 17 and tested their simulator using 100 test cases selected by our method.

We received a comment from the software developers after the update as follows: *The test execution was finished in half a day. This execution time was short enough that it did not affect the execution of the simulator. The test cases did not find any incompatibilities. The simulator is working without failure for six months until now. We considered that these selected test cases contained various execution times and coverage, which are better than the test cases selected by our previous Component-based method.*

From this comment, we conclude that our test case selection method is useful for selecting the representative test cases from a large number of test cases.

In this experiment, our method did not detect any defects. However, this does not mean that failures will not occur in the future. The most challenging part of this research is that no one can ensure that selected test cases are sufficient for regression testing. High code coverage does not imply a high coverage of library usage scenarios implemented in the system. This might be a reason that developers prefer 100 test cases with 99.76% code coverage to a larger set of test cases with 100% code coverage. An advanced method to evaluate the effectiveness of test cases in the context of dependency updates would be helpful for software developers.

V. Related Work

Rothermel et al. [7] and Zhang et al. [9] proposed methods for test case selection using runtime information and source code change information to find source code that should be tested. We cannot simply apply these existing methods in our situation because the software itself does not change when the dependencies are updated.

Many researches [1] [3] [5] proposed test case selection methods using the method call information. While our method is close to these approaches, we adopt bytecode instructions and their number of occurrences so that we can represent both code coverage and diversity of execution time as a vector.

Adithya et al. [6] proposed a system that performs data-driven test minimization. Machalica et al. [4] also proposed a new predictive test case selection strategy. Their methods use historical test outcomes of the target software and select the test cases based on the predicting test results. While these methods are useful, they are not applicable to our simulator because they need to maintain a history of repeatedly executed test results for a large number of test cases.

Gligoric et al. [2] proposed a test case selection technique that can integrate well with testing frameworks. When developers modify test files, this technique detects and selects the affected test cases from the dynamic dependencies. However, we could not employ only this technique because a dependency update may change many files at the same time, especially in the case of this Java update.

VI. Conclusion

We report the current test case selection practice to update dependencies in a company that maintains an industrial simulator. We found that software developers in the company used a very small number of test cases to quickly check the impact of dependency updates, even though they had a sufficient number of test cases achieving 100% code coverage. The software developers recognized our method as useful because the selected test cases included a wider range of parameters than their previous set of test cases. The software developers used 100 test cases selected by the method when they updated JDK for the simulator.

In future work, we would like to combine our method with the Component-based selection so that we can efficiently select test cases having various execution paths from tens of millions of test cases in a practical time. We also would like to define metrics for the diversity of execution time so that we can adopt state-of-the-art test case selection methods. We are also interested in how to automatically find the optimal number of clusters that meets developers' requirements.

Acknowledgment

This work was supported by JSPS KAKENHI Grant Numbers JP18H04094 and JP20H05706.

References

[1] S. Chen, Z. Chen, Z. Zhao, B. Xu, and Y. Feng, "Using semi-supervised clustering to improve regression test selection techniques," in *Proceedings of the IEEE International Conference on Software Testing, Verification and Validation*, 2011, pp. 1–10.

[2] M. Gligoric, L. Eloussi, and D. Marinov, "Practical Regression Test Selection with Dynamic File Dependencies," in *Proceedings of the International Symposium on Software Testing and Analysis*, 2015, pp. 211–222.

[3] E. Juergens, B. Hummel, F. Deissenboeck, M. Feilkas, C. Schlögel, and A. Wübbeke, "Regression test selection of manual system tests in practice," in *Proceedings of the European Conference on Software Maintenance and Reengineering*, 2011, pp. 309–312.

[4] M. Machalica, A. Samylkin, M. Porth, and S. Chandra, "Predictive test selection," in *Proceedings of the International Conference on Software Engineering: Software Engineering in Practice*, 2019, pp. 91–100.

[5] D. Mondal, H. Hemmati, and S. Durocher, "Exploring test suite diversification and code coverage in multi-objective test case selection," in *Proceedings of the International Conference on Software Testing, Verification and Validation*, 2015, pp. 1–10.

[6] A. A. Philip, R. Bhagwan, R. Kumar, C. S. Maddila, and N. Nagppan, "FastLane: Test Minimization for Rapidly Deployed Large-Scale Online Services," in *Proceedings of the IEEE/ACM International Conference on Software Engineering*, 2019, pp. 408–418.

[7] G. Rothermel and M. J. Harrold, "A Safe, Efficient Regression Test Selection Technique," *ACM Transactions on Software Engineering and Methodology*, vol. 6, no. 2, pp. 173–210, 1997.

[8] K. Shimari, T. Ishio, T. Kanda, N. Ishida, and K. Inoue, "NOD4J: Near-omniscient debugging tool for java using size-limited execution trace," *Science of Computer Programming*, vol. 206, 102630 (13 pages), 2021.

[9] C. Zhang, Z. Chen, Z. Zhao, S. Yan, J. Zhang, and B. Xu, "An Improved Regression Test Selection Technique by Clustering Execution Profiles," in *Proceedings of the International Conference on Quality Software*, 2010, pp. 171–179.