Regression Test Selection Tool for Python in Continuous Integration Process

Kauhanen, Eero Olavi

Regression Test Selection Tool for Python in Continuous Integration Process

Eero Kauhanen[1], Jukka K. Nurminen[1], Tommi Mikkonen[1], Matvei Pashkovskiy[2]

[1]*Department of Computer Science*, University of Helsinki, Helsinki, Finland

first.[initial].last@helsinki.fi

[2]*F-Secure Corporation*, Helsinki, Finland

matvey.pashkovskiy@f-secure.com

Abstract—In this paper, we present a coverage-based regression test selection (RTS) approach and a developed tool for Python. The tool can be used either on a developer's machine or on build servers. A special characteristic of the tool is the attention to easy integration to continuous integration and deployment. To evaluate the performance of the proposed approach, mutation testing is applied to three open-source projects, and the results of the execution of full test suites are compared to the execution of a set of tests selected by the tool. The missed fault rate of the test selection varies between 0–2% at file-level granularity and 16–24% at line-level granularity. The high missed fault rate at the line-level granularity is related to the selected basic mutation approach and the result could be improved with advanced mutation techniques. Depending on the target optimization metric (time or precision) in DevOps/MLOps process the error rate could be acceptable or further improved by using file-level granularity based test selection.

Index Terms—Regression test selection, test automation, continuous integration, mutation testing, software engineering.

I. INTRODUCTION

In a large software project, running the full test set can take a long time. In particular, when working in the modern DevOps style, developers are frequently checking in their modifications, and the resulting updates are tested and deployed on the fly. Having to wait – often only for some minutes, but in extreme case hours – at each check-in slows down development and is frustrating for developers. Furthermore, it consumes computing resources. For example, Mozilla estimates each check-in to cost over $25 in Amazon Web Services fees [1] while Google suggests that their annual continuous integration (CI) system execution is in millions of dollars [2].

Regression test selection (RTS) – selecting the most appropriate test cases to ensure identification of bugs in new code – has been studied for a long time (see e.g. [3]). Engström et al. [4] and Yoo and Harman [5] are recent surveys regarding the different RTS techniques, while Engström et al. [6] is a survey of empirical evaluation approaches to RTS. However, the importance of RTS has only become crucial in modern DevOps and MLOps environments. Identifying and rerunning only the relevant tests after code changes is necessary for productivity and efficient resource use in CI.

Previous RTS studies have largely focused on Java and other compiled languages [7]. The rationale has been that large, complex software, which would benefit the most from RTS, was relying on these languages. However, the situation is rapidly changing, and Python and other interpretable languages have become increasingly popular. One of the reasons for that increase is the growth of MLOps culture where Python plays a key role along with for example PySpark framework for data analysis and building ML models. RTS can bring even more benefits to the MLOps process because testing data transformation logic and model building are extremely expensive operations. This in turn has raised a growing interest in RTS in the context of Python.

In this work we take a new look at RTS, placing Python at the core. In particular, our key contributions are:

1) We develop a regression test tool, which integrates with Pytest, Coverage.py and Git to support CI workflow (Section III).
2) This toolset will be the core of the demo part of this paper, describing a demo setup that showcases these features at the hands-on level (Section IV).
3) We present early results of our experiment (Section V) and investigate how well our system finds bugs after code changes (sometimes called *inclusiveness* [3]), how much computing effort it saves, and how much resources are needed for bookkeeping (Section VI).

II. BACKGROUND AND RELATED WORK

RTS has been studied since the 1980s, and numerous tools have been proposed. Rothermel and Harrold [3] is an early and influential study of RTS techniques. They divide the techniques into three different sets: (i) *Coverage techniques* aim to locate changed program components and select tests that exercise those; (ii) *Minimization techniques* aim to choose the smallest set of tests for modified program components; (iii) *Safe techniques* aim to select all tests that can expose faults in updated programs. Our focus is primarily on coverage techniques, but, at the same time, we aim to minimize the needed tests.

RTS technique should be safe and precise. Safe means that all relevant tests are executed, precise means that only necessary tests are executed. Some studies, e.g. Shi et al. [8] focus on improving safety when dealing with different language constructs. Others, including us, accept that full safety is not possible and integrate the RTS approaches to development processes in such a way that occasional test omissions are acceptable.

Like our study, many of the latest works have focused on integrating RTS into software development. Ekstazi [9] is a dynamic RTS technique for Java programs. It is integrated with popular testing frameworks (JUnit and ScalaTest) and build systems (Maven and Ant). Likewise, searching for solutions that work in practice, Gyori et al. [10] study test selection opportunities in a very large open-source ecosystem. Finally, some tools, e.g. Jest JavaScript test framework, have built-in RTS support.

Novel approaches to RTS include using machine learning [11] and using natural language processing [12]. We also seek to extend our work in this direction.

III. Introducing the Tool

It is important to consider the usage of RTS tools from two perspectives: from developers and CI pipelines. The main difference between those is the fact that the changes done by developers on their local workstations could be identified even without version control systems (VCS), but if test selection is used on CI server VCS plays an important role in the identification of recent changes. Tools like Ekstazi [9] and Pytest-testmon[1] address RTS from a local development perspective whereas our tool is also addressing test selection on CI servers.

Our tool aims to integrate with the VCS of the project under development. This allows RTS to be performed in a typical workflow where new features are developed in separate version control branches and a CI-pipeline is configured to run the tests. Currently, two types of tools are available for RTS: code coverage based and history-based. The current code coverage based tools, such as Ekstazi [9] and Pytest-testmon, only work on the developer's local machine by creating a mapping database completely independent from any version control data. Without version control data linked to the mapping database, selecting correct tests between software versions becomes extremely difficult. The history-based tools, such as ChangeEngine[2], require data from previous test runs, and their output can be inaccurate if the data is scarce.

Git VCS and Coverage.py are used for tracking changes in the code. For the test runner, Pytest is used. As already mentioned, there are two settings where the tool is used, the developer's machine and CI server. To start using the tool initialization of a mapping database has to be performed on the developer's machine: an initial run of all tests has to be done. While performing the first run of the tests, a locally stored SQLite database is constructed with the coverage data provided by Coverage.py.

After the initial full test suite run, the tool is ready to be used. On the developer's machine, when changes are made to the target project's files, the tool checks for changes in the Git working directory. The tool first constructs a list of changed files according to Git and checks which of those files are either source code files or test code files in our local database. After the tool has determined which files are taken
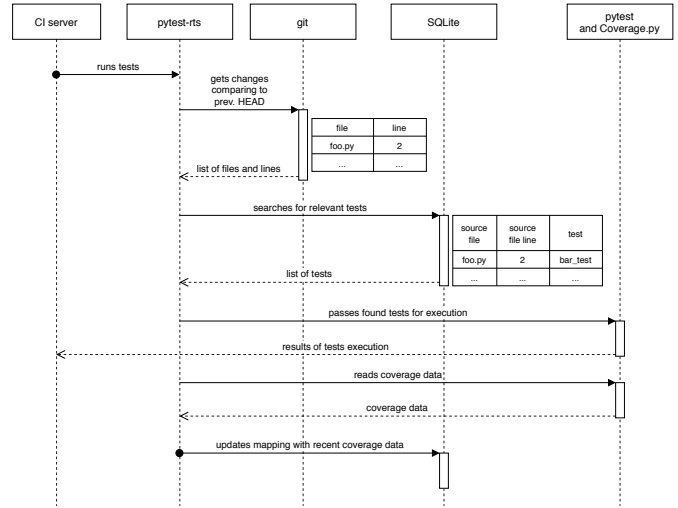
Fig. 1. CI workflow with our tool

into consideration, it checks the Git diff output for each of those files. From this 'diff', the tool can determine which lines have changed and which lines have shifted from their original position. Then the tool can query all the test functions from our database according to the list of line numbers for the changed lines and run them with Pytest. No database state updating is performed during this. If a user wishes to make these changes final, a Git commit operation is required. When the changes are committed and obtained by the CI server, an agent on the server runs the tool and checks whether the current Git HEAD hash differs from the one that is marked as a last update hash in the mapping database. If so, the tool searches for relevant tests, queries the changes and tests for those changes and checks for newly added test functions by checking what test functions Pytest can find and comparing it to the current state in the database. The tool also calculates how unchanged lines have shifted in the files and performs a database update based on this information. When the tests are run after this, new coverage data is collected and inserted into the database. The RTS sequence in the CI environment is presented in Figure 1.

IV. Tool demo

The tool demo shows how a developer can use the tool in an open-source project. The steps include (i) initializing the tool, (ii) making changes in the Git working directory, and (iii) committing the changes. The operation that handles committed changes highlights how the tool could work in a CI-environment. By initializing the tool in the master branch and then checking out a feature branch results in a scenario where the tool compares the current feature branch Git HEAD to the one in master. The resulting test set from this comparison could be used in pull requests to determine which tests are required to test the actual changes.

The tool is called pytest-rts and the source code is available in our Github repository.[3] Pytest-rts currently supports RTS

based on changes in the current Git working directory, as well as committed changes. A simple test prioritization algorithm is also implemented by storing the run times of each test case and sorting the queried tests based on this information. The tests with the shortest run time are ran first to attempt finding a fault as fast as possible.

V. Experimental Setup

We evaluated our tool with a simple mutation approach. A random line was deleted and our system selected and executed the tests it considered relevant. The use of existing mutation testing tools was considered (see e.g. [13] for a review mutation testing approaches for Python code). However, many of the mutation tools work at abstract syntax tree or byte code level, making it hard to attribute a specific mutation to a certain code line. On the other hand, studying C# programs, Derezińska [14] observe that deletion mutations perform well when compared to more complex mutations.

Our approach, to evaluate the case where a random source code line was deleted, first required a full test suite run to build the local mapping database. Then, all the target project's source code files were queried from the database to a list. We then ran multiple iterations where a source code file was randomly selected from this list and a randomly chosen, non-empty line was changed to an empty line. Next, we checked the line number of this mutated line and queried all the tests for that specific line, based on our mapping database. Then, we queried all the tests that had mapping information for any of the line numbers for the file in question. At this point we had three test sets; the full test suite, a line-level granularity test set for the change, and a file-level granularity test set for the change. Then we fed these test sets to Pytest one by one and saved the Pytest exit code for each runs. We also added a fake exit code labeled '−1' to indicate that a timeout had appeared while running the tests. This was to prevent our mutation causing eternal loops.

We plan to run better mutation testing in the future with mutation testing tools and make the evaluation step the core of the tool's module test automation (TA). This could provide better data on how our approach handles changes in a real-life scenario. At the moment, the TA for the tool module is built with GitHub Actions.[4] Our future plans also include the use of historical data of existing projects in Github. This is naturally more realistic. However, it is not a very scalable approach, because it can be used with only some git updates. When import dependencies change – something which frequently happens in Python – the system needs to be manually reconfigured. We are investigating ways to automate this step further.

VI. Results

For the experimentation we collected 1) the number of cases where the reduced test set failed to find the error; 2) the reduction in the number of tests; and 3) the size of

[4]https://github.com/features/actions

TABLE I
EXPERIMENTAL RESULTS OF RANDOM LINE REMOVAL TEST - 10 000 ITERATIONS PER PROJECT

Project name	Flask	Rich	Python-rsa
Lines of code	16143	22762	7149
Tests	487	391	94
Mapping database size (MB)	4.43	2.91	0.28
Line-granularity missed fault rate (%)	22.83	35.23	23.94
File-granularity missed fault rate (%)	2.43	0.00	2.25
Line-granularity test set size (%)	2.98	2.06	4.05
File-granularity test set size (%)	32.46	14.13	26.00

the dependency database (e.g. Machalica et al. [11] claim that the dataset becomes prohibitively large)

We evaluated our tool's operation in three open source projects: Flask[5], Rich[6], and Python-rsa.[7] Flask is a popular Python micro-framework for building web applications, Rich is a library for terminal text formatting and Python-rsa is a Python implementation of RSA public-key cryptosystem. Table I contains the data for our evaluation results. We ran our evaluation for 10 000 iterations per project. This amount was chosen because it provided a good indication on what types of lines were problematic for our tool. The missed fault rate is calculated by taking the number of cases where the test set did not contain any tests or only a set of passing tests was selected, but running the entire test suite found an error. We also extracted the average line-level and average file-level test set sizes. With that data, we calculated how many tests different levels of selection yields. The table shows how the test set sizes were compared to the original. We use the terms 'line-granularity' and 'file-granularity' to indicate from which selected test set the results are. The term line-granularity means that our tool selected tests on the line-level granularity and the term file-granularity means that our tool did the selection on the file-level granularity.

The data shows that the line-granularity missed fault rate is quite high and inspection of the individual failure cases indicates that the coverage based approach has trouble keeping track of lines that contain a function or class definition (e.g. 'def foo() / class foo()'), decorators (e.g. '@decorator') or module imports (e.g. 'import module'). Further investigation based on these results shows that the tool's mapping database has missing coverage data for these types of lines. This seems to be an issue with the method of mapping coverage data: the aforementioned lines are executed in Python before the coverage collection is started for any specific test function. By removing these cases from our evaluation results, we are left with new missed fault rates shown in Table II.

Inspecting the cases after this adjustment reveals no other clear classes of error-causing lines that could be pinpointed to a specific fault in our coverage collection. Only one observation could be made: many of the errors were caused by

[5]https://github.com/pallets/flask
[6]https://github.com/willmcgugan/rich
[7]https://github.com/sybrenstuvel/python-rsa

TABLE II
ADJUSTED MISSED FAULT RATES

Project name	Flask	Rich	Python-rsa
Line-granularity missed fault rate (%)	15.73	24.01	18.29
File-granularity missed fault rate (%)	2.09	0.00	1.60

removing the start or end of a multi-line string, which caused the code to be invalid. The file-granularity missed fault rate is significantly lower even after the adjustment, but the error causing lines seem to be similar to the ones in line-granularity error cases. It seems that the average file-granularity test set size is much bigger than the average line-granularity test set size. This has most likely helped the system to find a fault in cases where a non-covered line was deleted.

VII. CONCLUSIONS

In this paper, we have introduced a coverage-based RTS approach and a tool for Python, providing easy integration to continuous integration and deployment. The tool was evaluated by benchmarking its performance with open source software. Mutation testing was applied and the results of the execution of full test suites were compared to the execution of a set of tests selected by the tool. In the tests, the missed fault rate of the test selection at file-level and line-level granularities vary between 0–2% and 16–24%, respectively. The high missed fault rate at the line-level granularity is related to the selected basic mutation approach and the result could be improved with advanced mutation techniques. Depending on the target optimization metric (time or precision) in DevOps/MLOps process the error rate could be acceptable or further improved by using file-level granularity based test selection.

In our future work, there are two directions we look into. First, to better consider timing constraints, we plan to study the possibilities for prioritizing tests. Then, we could ensure that the most important tests are always run, whereas some less important ones might be omitted to save time. In addition, flaky tests, or tests that both pass and fail periodically without any code changes, is a direction for future research.

VIII. ACKNOWLEDGEMENT

REFERENCES

[1] J. O'Duinn. (2013) The financial cost of a checkin. Accessed 2020-08-11. [Online]. Available: https://oduinn.com/2013/11/20/the-financial-cost-of-a-checkin-part-1/

[2] M. Hilton, T. Tunnell, K. Huang, D. Marinov, and D. Dig, "Usage, costs, and benefits of continuous integration in open-source projects," in *2016 31st IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2016, pp. 426–437.

[3] G. Rothermel and M. J. Harrold, "Analyzing regression test selection techniques," *IEEE Transactions on software engineering*, vol. 22, no. 8, pp. 529–551, 1996.

[4] E. Engström, P. Runeson, and M. Skoglund, "A systematic review on regression test selection techniques," *Information and Software Technology*, vol. 52, no. 1, pp. 14–30, 2010.

[5] S. Yoo and M. Harman, "Regression testing minimization, selection and prioritization: a survey," *Software testing, verification and reliability*, vol. 22, no. 2, pp. 67–120, 2012.

[6] E. Engström, M. Skoglund, and P. Runeson, "Empirical evaluations of regression test selection techniques: a systematic review," in *Proceedings of the Second ACM-IEEE international symposium on Empirical software engineering and measurement*, 2008, pp. 22–31.

[7] B. G. Ryder and F. Tip, "Change impact analysis for object-oriented programs," in *Proceedings of the 2001 ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, 2001, pp. 46–53.

[8] A. Shi, M. Hadzi-Tanovic, L. Zhang, D. Marinov, and O. Legunsen, "Reflection-aware static regression test selection," *Proceedings of the ACM on Programming Languages*, vol. 3, no. OOPSLA, pp. 1–29, 2019.

[9] M. Gligoric, L. Eloussi, and D. Marinov, "Ekstazi: Lightweight test selection," in *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, vol. 2. IEEE, 2015, pp. 713–716.

[10] A. Gyori, O. Legunsen, F. Hariri, and D. Marinov, "Evaluating regression test selection opportunities in a very large open-source ecosystem," in *2018 IEEE 29th International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, 2018, pp. 112–122.

[11] M. Machalica, A. Samylkin, M. Porth, and S. Chandra, "Predictive test selection," in *2019 IEEE/ACM 41st International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*. IEEE, 2019, pp. 91–100.

[12] S. Sutar, R. Kumar, S. Pai, and B. Shwetha, "Regression test cases selection using natural language processing," in *2020 International Conference on Intelligent Engineering and Management (ICIEM)*. IEEE, 2020, pp. 301–305.

[13] A. Derezinska and K. Halas, "Experimental evaluation of mutation testing approaches to Python programs," in *2014 IEEE Seventh International Conference on Software Testing, Verification and Validation Workshops*. IEEE, 2014, pp. 156–164.

[14] A. Derezińska, "Evaluation of deletion mutation operators in mutation testing of C# programs," in *International Conference on Dependability and Complex Systems*. Springer, 2016, pp. 97–108.