Overview of Regression Test Selection Techniques

Yigit K. Erinc

Seminar: Software Quality
Technical University of Munich
{yigit.erinc}@tum.de

Abstract. Regression Test Selection is a vital technique for reducing the time of testing and the cost of running the tests which is of uttermost importance, especially for the companies with a big codebase containing thousands of tests. In the past 45 years [15], many approaches have been tried and proposed to reduce the number of tests that needs to be run based on a given set of changes. In this work, we make an attempt to summarize and categorize those approaches based on the techniques they are utilizing.

Keywords: Regression Test Selection · Software Engineering · Software Testing.

1 Introduction

Testing undoubtedly builds confidence in one's code and allows the programmer to immediately detect if their changes breaks anything. In paradigms such as Test Driven Development, the programmer is supposed to write the tests even before the actual code since it is believed to give programmer a thorough understanding of the possible corner cases in problem domain and design the code better. In modern Software Engineering, there are many types of different testing techniques, with examples such as: Unit Tests, Integration Tests, End to End Tests and System Tests which in turn results in a rapid increase in number of tests. When the number of tests start to grow in the thousands scale, the time it takes to run the whole test suite can range from a few minutes to an hour as reported in [13]. Waiting time for the tests to complete running results in distraction and frustration for the developer. It also causes the Continuous Integration (CI) process to run for a lot longer time than it should, causing a waste of compute time which would also result in increased costs if the company is running their tests in the Cloud. Mozilla has reportedly estimated the cost of each check-in to cost over $ 25 in Amazon Web Services fees. [18] Google also reported their annual continuous integration (CI) system execution to be in millions of dollars. [17]

Existing test runners, run the tests in either random order or in the order they are declared in the test file and thus discovered. Test Case Prioritization (TCP) tools aim to minimize the time to failure by running the tests that are most likely to produce errors based on a given set of code changes. [22] [2] On the other hand, test selection tools aim to minimize the number of tests that will be run by skipping the tests that should not be affected by the changed code and only run the ones that depend on changed code. [9] [27] [35] [30] [21] [31]

In the context of Software Quality, the testing makes sure that the functionality of the software is working as expected. With the regression test selection in place, developers can run the tests quickly and have confidence in their code. They can also add more tests without worrying about increasing the total running time of the test suite which may result in a higher quality test suite.

In this work, state of the art techniques of regression test selection will be presented and compared with details and reader will be able to compare and contrast multiple methods after reading this.

2 Approach

The literature research for this work was mostly done on Google Scholar by using the key words <regression test selection >, <regression testing >, <test selection >. Not all papers were found with search but some were selected because they were commonly cited by the papers in field. Although the first solution that is presented is dated 2005 [27], the emphasis was put on more recent techniques (2015 and after). The papers were ordered inside each section based on the year they were published.

For the sake of completeness, we should mention that the presented techniques are neither the first techniques in the field nor do we present a complete classification of proposed approaches. The older techniques have been purposefully excluded because they are mostly irrelevant and because of this decision, we will not be mentioning all types of techniques, skipped techniques include but not limited to: Firewall-based approach [20][32], CFG analysis[28][29], test selection based on UML design[8][7][10], model-based Regression Test Selection (RTS) [23][1][5] and slicing-based techniques.[4][3]

3 Results

As mentioned in the introduction, there are two main driving factors to optimize the running of tests: The first one is to provide faster feedback to the developer who committed the changes and needs to see if their changes are passing the CI process or not. The second one is to cut the costs of running the CI process on the cloud. Test selection is the approach that companies that do large-scale software engineering take when the goal is to decrease cost of running the test suite. In this paper, we will provide our focus towards various test selection approaches. In this section, we will provide an overview of the Test Selection and categorize them based on how they select the tests.

In this section we will take a look at multiple proposed strategies to select a subset of tests based on a set of code changes. Test selection techniques can be grouped into a few categories based on how they select tests: Based on build dependencies, static code analysis, dynamic analysis and the ones that use Machine Learning/Deep Learning methods. There may be other ways to do test selection but these seem to be the most common techniques among various proposed approaches.

3.1 Code Analysis Based Approaches

Code analysis-based tools perform techniques such as static analysis in compile time or dynamic analysis while the code is loaded in the memory and running.

A very simple approach to building a test selection strategy is to analyze code changes in build time and make use of the build tools that create the dependency graph while building the project. It works by detecting the transitive dependencies and running the tests for all the code that has been either changed itself or dependent on one of the changed files (technically speaking, a transitive dependency). Facebook mentioned that they were using such a method initially before the necessity to develop a more sophisticated approach presented itself. [21]

In 2004, Ren et al. proposed Chianti [27] as a tool for analyzing the impact of code changes with a static analysis, specifically for Java. Although Chianti was not designed as a RTS tool but rather a tool that focuses on assisting developers by helping them understand impact of program edits, Chianti was also proven to be effectively usable for RTS use case. Chianti does not depend on previous test executions' information while selecting the tests. It works by finding the set of atomic changes from the program changes finds the correlation between the changes and nodes and edges in the generated call graphs. Their static analysis works at a method-level granularity. It works as follows: Chianti first analyzes the code to come up with the set of interdependent atomic changes. Then the tool generates a call graph. It is able to use either dynamic call graphs or pre-constructed static call graphs. Previously presented techniques such as Orso et al.'s [24] made use of Control Flow Graphs whereas Chianti opted for using the Java Inter-class Graphs. The tool selects the tests by correlating the set of atomic changes to the call graphs to find which tests must be affected by a given change.

When it comes to code analysis, it is important to mention the effect of granularity. Proposed techniques that aim to decide which tests should be run based on code changes based on static code analysis either do it in the file granularity or method granularity or a mix of both. In 2012, Zhang et al. [35] presented the FaultTracer as a regression test selection tool for Java programs. FaultTracer has been found to decrease the number of selected tests slightly while performing better in terms of detecting the affecting change determination when compared to Chianti and outperforming Chianti by 20% on average. [34] Just like Chianti [27], FaultTracer is another tool that performs code analysis with a method level granularity. The architecture of FaultTracer can be seen in the Figure 2. It takes the original and changed version of the program, runs an analysis to detect the set of atomic changes. It then generates an extended call graphs for tests which will be used to rank the tests in terms of how likely it is that they have been affected by the changes.

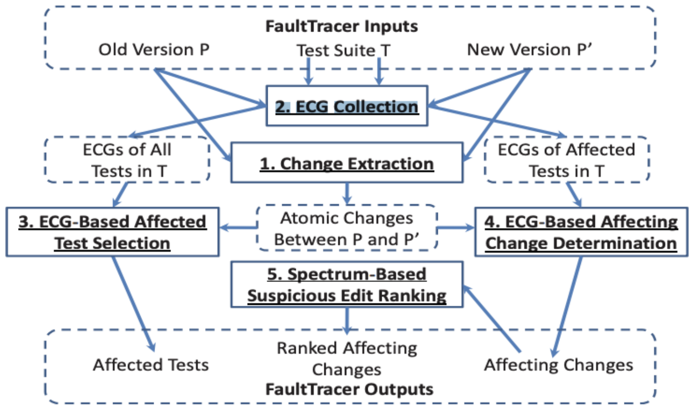


Fig. 1. Fault Tracer Architecture.
Source: Taken from [35]

In 2015, Gligoric et al. proposed Ekstazi [16] as a lightweight test selection tool. Ekstazi is again developed for Java, just like Chianti [27] and FaultTracer [35]. It has been successfully integrated with the popular Java testing framework JUnit [1] which is itself well-integrated with the Java build tools such as Maven [2] and Ant [3], transitively providing the same level of integration to Ekstazi. It requires no integration with source code tools such as Git. Authors argue that, based on their experiments Ekstazi was able to provide a 32% reduction on end to end testing time on average, and 54% reduction for longer-running test suites with test selection. Gligoric et al. suggests that the evaluation of testing frameworks should include the end to end effect of using regression test selection. They suggest we should view the test selection procedure as a 3 step procedure:

1. Analysis
2. Execution
3. Collection

Analysis phase is the pre-procedure to select tests to run and any pre-computation. Execution phase is the actual time to run the tests, this is the

[1] JUnit: https://junit.org/junit5/
[2] Maven: https://maven.apache.org/
[3] Apache Ant: https://ant.apache.org/

step where we should be seeing the speedup benefits of running less tests. Collection phase is the phase that algorithm collects data from the test session for the following analyses.

Gligoric et al. state that most analyses do not take the Analysis and Collection steps into account when comparing RTS techniques but only mention the improvements in the number of selected tests.

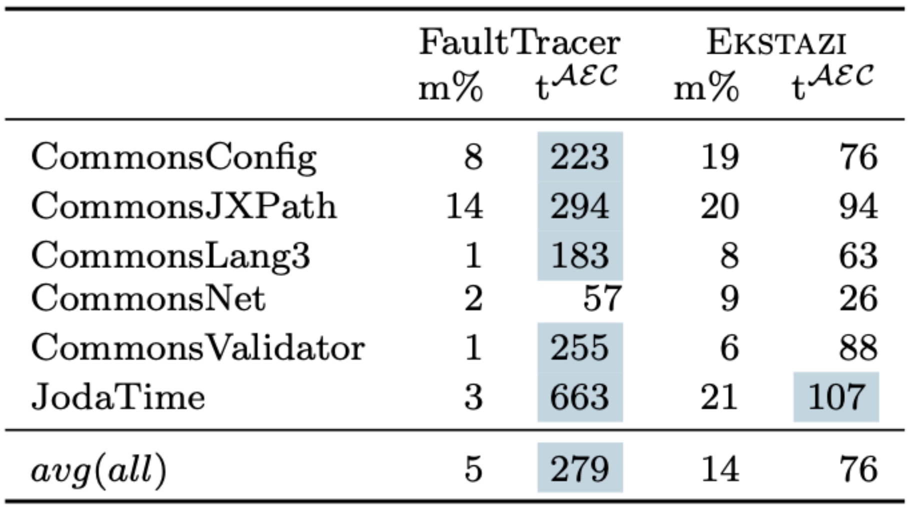
To realistically evaluate the speedup of a RTS technique, we can use the following formula:

$$\textit{speedup}(\%) = 1 - \frac{\text{Analysis} + \text{Execution} + \text{Collection}}{\text{Time without RTS}}$$

The formula states that if the end to end time with overhead included is smaller than the time of running the test suite without the RTS, it is beneficial.

On the contrary to Chianti [27] and FaultTracer [35] that use method-level granularity, Ekstazi [16] uses File level granularity. Tool firstly saves the mapping of each test to the files they depend on. They store the checksum of the file and use that checksum to detect any changes in the files in a very efficient way. If the checksum of any of the files has changed, Ekstazi will then pick that test. Gligoric et al. [16] emphasize that by calculating the checksum, they are able to do it in a much easier way in comparison to history based tools that make use of the version control system to compare old and new revisions.

It is also important to note that the analysis performed by Ekstazi is a dynamic analysis since it is done while the test suite is running and the classes are loaded into the JVM. Ekstazi collects data during the testing and it uses that information in the following test runs.



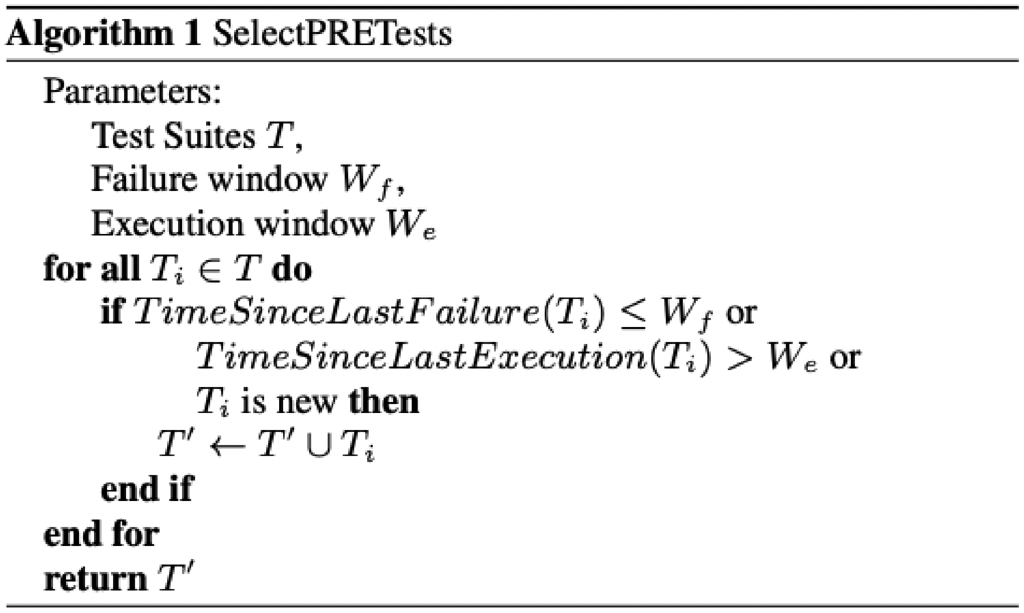
	FaultTracer		Ekstazi	
	m%	$t^{\mathcal{AEC}}$	m%	$t^{\mathcal{AEC}}$
CommonsConfig	8	223	19	76
CommonsJXPath	14	294	20	94
CommonsLang3	1	183	8	63
CommonsNet	2	57	9	26
CommonsValidator	1	255	6	88
JodaTime	3	663	21	107
avg(all)	5	279	14	76

Fig. 2. Fault Tracer versus Ekstazi Performance.
Source: Taken from [16]

Their experiments show that the Ekstazi, although selects more tests than FaultTracer, is around 3.5 times faster than FaultTracer [35] when it comes to end to end time because method level analysis made by FaultTracer is taking too much time in the analysis phase. Authors also added method level granularity support to Ekstazi tool and they noted that the method selection granularity is slower for their tool as well.

3.2 Heuristics Based Approach

In 2014, Elbaum et al. from Google [11] has reported that the existing approaches are inapplicable to their code-base since most of the techniques require an analysis to be made and/or the maintenance of coverage data. It is unfeasible to keep using the same coverage data or analysis for subsequent test runs for a code-base with a very high churn-rate since the analysis results will quickly become out-dated. As a result, they have come up with a new technique based on heuristics.



Algorithm 1 SelectPRETests

```
Parameters:
    Test Suites $T$,
    Failure window $W_f$,
    Execution window $W_e$
for all $T_i \in T$ do
    if $TimeSinceLastFailure(T_i) \leq W_f$ or
            $TimeSinceLastExecution(T_i) > W_e$ or
            $T_i$ is new then
        $T' \leftarrow T' \cup T_i$
    end if
end for
return $T'$
```

Fig. 3. Google Test Selection Algorithm
Source: Taken from [11]

As the Figure 5 explains, algorithm is very simple, the test selection is made based on two windows that are maintained: Failure window and Execution window. They do the selection on test suites and not the individual tests. The algorithm iterates over all the test suites to check if they haven't been run for a particular number of test runs or a test in that suite have failed recently and if that is the case it runs the suite. Also, if the test suite is newly created, it again chooses to run it.

As an example, let's say that Failure window is 15 and execution window is 20. If a test suite has failed in the last 15 test runs or if hasn't been executed for the last 20 test runs it will be selected.

3.3 Artificial Intelligence Based Approaches

In the recent years, the attention has been turned into using Machine Learning and Natural Language Processing techniques for test selection problem.[31][21] These techniques aim to take advantage of the development in Machine Learning to infer which tests should be selected by training the test selection model with the data of previous testing outcomes.

In [21], the authors mention that most test selection methods are unusable for a multi-language monorepo codebase with a tremendous scale and thus creating the necessity to developing a new method. Using Machine Learning techniques and the data of previous changes and their corresponding test runs, their strategy was able to produce very promising results, catching over 95% of individual test failures and over 99.9% of faulty code changes. They also report that the technique selects fewer than a third of the tests in comparison to their previous approach with build dependencies and a decrease in the total infrastructure cost of testing by a factor of two. [21]

Their technique works by extracting features from the code changes and test targets such as file extensions, project name and failure rates. They have decided to use a gradient boosted decision tree classifier for their model. The model calculates a score for each pair (change, test) and selects the test when the score of pair is exceeding a certain threshold.

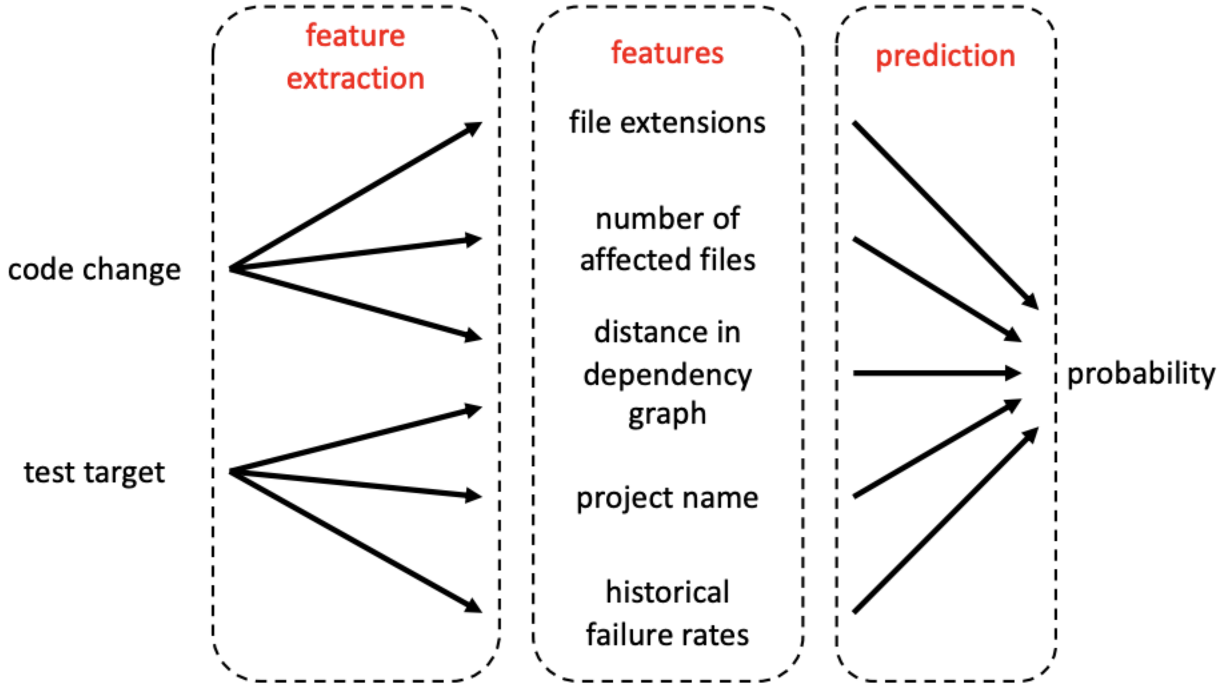


Fig. 4. Feature Extraction and Prediction. Source: Taken from [21]

They have also applied feature selection to their model to figure out which features are actually improving the model. This is an interesting inference that helps us understand the nature of test selection problem better. It was found out that the following features are the most important: file extensions, change history, failure rates, project name, number of tests and minimal distance (between target and change).

While we cannot compute exactly the set of impacted tests for a particular change, we can approximate this computation by learning to identify which tests would have reported a failure, based on historical data.

Although the model may theoretically miss a few test cases and may not have the precision of dynamic analysis, it is preferable in some cases since it does not store any coverage or analysis data. Another advantage is that it is directly applicable to any change without requiring an additional analysis step. It does not say exactly which tests were affected by the change but rather focus on calculating the probability of which tests may fail based on a given change.

There have been attempts in test case prioritization to use lexical information such as variable, method and class names to correlate tests with code. [22] In [31], NLP techniques has been used for the RTS problem. Authors try to extract the semantic similarity between two test cases and the defects by making use of Machine Learning (ML) techniques.

They follow a multiple step procedure:

1. Corpus Formation (Feature Extraction from Text)
2. Text Preprocessing (Tokenization)
3. Vectorization
4. Perform Similarity Measures
5. Select tests over threshold

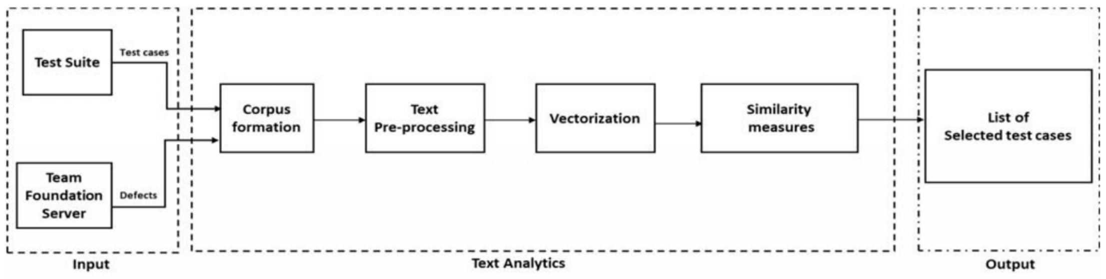


Fig. 5. Solution Overview Source: Taken from [31]

Their technique makes use of the "Bag of Words" model for vectorization and calculated the Term Frequency-Inverse Document Frequency (TF-IDF). Then they calculate the similarity between documents by using Cosine Similarity metric.

$$\cos(\mathbf{t}, \mathbf{e}) = \frac{\mathbf{t}\mathbf{e}}{\|\mathbf{t}\|\|\mathbf{e}\|} = \frac{\sum_{i=1}^{n} \mathbf{t}_i \mathbf{e}_i}{\sqrt{\sum_{i=1}^{n} (\mathbf{t}_i)^2} \sqrt{\sum_{i=1}^{n} (\mathbf{e}_i)^2}} \tag{1}$$

The results report that [31] approach was able to detect nearly 92% of the bugs.

3.4 Coverage Based Approach

Another way to perform regression test selection is to look at the difference between last revision of a repository and the current state of code or the difference between 2 or more commits. For modern programming languages, there are existing tools that perform dynamic code analysis to generate test coverage data. This coverage data, along with the differences between old revision and new revision, can be used to decide which tests should be run.

In the article by Kauhanen et al. [19] the suggested approach was to create the coverage with the help of Coverage.py [4] and read the dependencies of tests to files in the following test runs to decide which tests should be run. In their work, they argue that Ekstazi addresses the RTS problem for local development but their tool is more focused for optimizing the running time of tests on the CI servers. They claim that without the use of version control data, selecting the correct tests with the changing versions become difficult.

[4] Coverage.py: `https://coverage.readthedocs.io/en/6.4.2/` for documentation

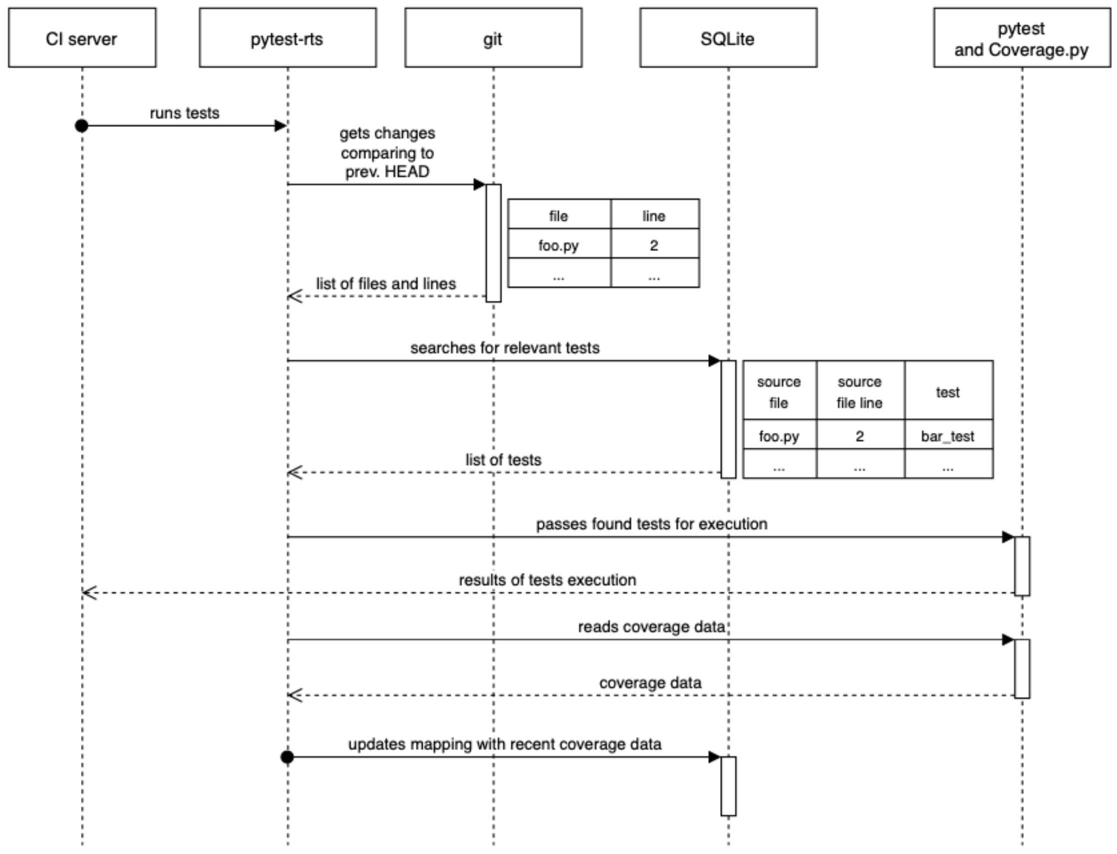

CI server
pytest-rts
git
SQLite
pytest and Coverage.py
runs tests
gets changes comparing to prev. HEAD
file | line
foo.py | 2
... | ...
list of files and lines
searches for relevant tests
source file | source file line | test
foo.py | 2 | bar_test
... | ... | ...
list of tests
passes found tests for execution
results of tests execution
reads coverage data
coverage data
updates mapping with recent coverage data

Fig. 6. Pytest RTS tool CI Workflow.
Source: Taken from [19]

The details of the technique can be seen in the Figure 7. First the CI server runs the tests using the modified pytest command which gets the changes by comparing to the previous HEAD. Tool then checks the coverage database to map the changes to corresponding tests. After running the tests it updates the mappings with recent coverage data.

4 Discussion

In this section, we will mention some factors and tradeoffs that needs to be considered when designing a new RTS strategy and try to summarize the methods that were described in previous section. This section can provide guidance in evaluating the strategies and also shed a light on the effects of different design strategies.

4.1 Evaluating RTS Performance Correctly

As we have mentioned in Section 5 and suggested in [16], the creators of new test selection strategies *must* take into account the analysis time and the collection time of the tests and do their time measurements for the whole cycle instead of only counting the number of selected tests. A proposed test selection strategy is only useful if it results in a decreased end to end testing time on average in comparison to *RetestAll*.

4.2 Test Selection Granularity Tradeoff

In the test selection strategies, there is a tradeoff between the granularity of the static/dynamic analysis and the number of selected tests. To understand why that is the case, we can think about one strategy that analyzes in file granularity and another one with method granularity: Let's say we have a test A that is only dependent on method B inside a class C and none of the other methods of the class. A strategy that can only check the file changes will automatically select test A and all other tests that are dependent on class C whereas a method-level analysis would not.

Although, it may seem like it is a good thing at first, a more precise analysis comes with its overhead in the code analysis and possibly collection phases. This overhead leads to slower end to end testing time than the strategies that perform a more coarse grained analysis such as file level. An example of this can be seen in the Fig.3.

In the [16] it is even argued that the FaultTracer [35] is even slower than *RetestAll* thus providing no real benefits by doing test selection and the reason is pointed to its method-level analysis.

4.3 We usually cannot guarantee safety

There are 2 different ways to analyze a source code:

1. Static Code Analysis
2. Dynamic Code Analysis

In static analysis, the code analysis is done in the compile time whereas with dynamic code analysis, the analysis is done while the code is running, e.g. collecting coverage/dependency data during the execution of a test suite. With static analysis we do not have a proven safe RTS technique.

Only techniques that are proven to be safe under certain assumptions and circumstances are techniques that use some type of dynamic analysis such as [11][25]. One fundamental problem with dynamic analysis is: We cannot prove that it will be safe in all cases because there can be corner cases depending on execution. The reasoning is described by Machalica et al. in [21] as follows: "Note that all dynamic methods must be treated as approximate, in a sense that they may ignore tests that would detect regression in a particular change. This is due to the fact it is fundamentally impossible to know control flow of a test execution before the test is exercised. Since test selection must take place before the test is run on a particular version of the code, it cannot be based on dynamic analysis of tests behavior on this exact version.".

Therefore, we can say that in most cases we cannot prove that an RTS technique will be safe at all circumstances but we can have a certain confidence with some proposed techniques [11] [25] [13] and that should be sufficient in most cases.

4.4 Scale Presents New Challenges

Monorepo means storing all or most projects of an organization in a single code repository. This approach has many benefits [26] some being avoiding the dependency problem, enables library changes for all projects at once and encourages code transparency among teams. Although having a code-base of that scale comes with its own unique set of challenges and the RTS strategy employed by such companies should have some properties. Efficient test selection is even more important since the number of tests drastically as the code-base grows, [21] expressed that the magnitude of number of tests that needs to be executed corresponding the weekly changes is in ten thousands. Given this magnitude, it also means that the cost of running the regression tests on the cloud is costing them millions of dollars each year.

Some methods such as coverage analysis is not applicable because that would just be too much data to store the coverage of a repository in Google scale. This applies to all the dynamic techniques that require recording traces of previous executions such as [16][9]. Language-specific tools also not preferred because most organizations of scale have multi-lingual code-bases. These requirements leave not too many options for the organizations of scale.

4.5 ML Provides New Opportunities

The recent approaches to RTS make use of Machine Learning ML and Natural Language Processing techniques. [21][31] These techniques are language agnostic and can be applicable to enormous code-bases because it does not have to store the coverage and dependency information which would be a problem. They have also proven to show good results in terms of precision with a very little compromise of safety due to the approximation being in the nature of solution. Although results are good, they are not necessarily better than simple heuristic-based approaches such as [11]. Recent research [12] show that the heuristics approaches outperform complex ML models most of the time.

5 Related Work

There have been attempts to categorize RTS techniques. Yoo [33] surveyed 159 papers to provide comprehensive insights into not only the RTS techniques but also related problems TCP and Test Set Minimization (TSM) by analyzing papers between 1977 and 2009 with 87 of the papers being related to RTS. Swarnendu [6] classified different approaches to RTS with critical evaluations of those classes. Engström [14] surveyed 27 RTS papers dating between 1997 and 2006.

6 Conclusion

In this paper we have provided an analysis of different RTS techniques with a focus on more recent techniques, made an attempt to classify them based on the solution approach. To our knowledge, there is no other survey on recent research of last ten years. In the recent years, there is a trend towards using ML for RTS solutions but it has also been shown [12] that these techniques are not essentially better in terms of safety and precision in comparison to heuristics-based approaches. It seems clear that the research is moving to use class-based granularity for the code analysis based approaches rather than the method or statement granularity because of performance reasons. The modern-age software development mostly requires an RTS solution to be able to work across language boundaries and to be integrated to CI pipelines with ease.

References

1. Al-Refai, M., Ghosh, S., Cazzola, W.: Model-based regression test selection for validating runtime adaptation of software systems. In: 2016 IEEE International Conference on Software Testing, Verification and Validation (ICST). pp. 288–298 (2016). https://doi.org/10.1109/ICST.2016.24
2. Bajaj, A., Sangwan, O.P.: A systematic literature review of test case prioritization using genetic algorithms. IEEE Access **7**, 126355–126375 (2019). https://doi.org/10.1109/ACCESS.2019.2938260
3. Binkley, D.: Reducing the cost of regression testing by semantics guided test case selection. In: Proceedings of the International Conference on Software Maintenance. p. 251. ICSM '95, IEEE Computer Society, USA (1995)
4. Binkley, D.: Semantics guided regression test cost reduction. IEEE Transactions on Software Engineering **23**(8), 498–516 (1997). https://doi.org/10.1109/32.624306
5. Biswas, S., Mall, R., Satpathy, M., Sukumaran, S.: A model-based regression test selection approach for embedded applications. SIGSOFT Softw. Eng. Notes **34**(4), 19 (jul 2009). https://doi.org/10.1145/1543405.1543413, `https://doi.org/10.1145/1543405.1543413`
6. Biswas, S., Mall, R., Satpathy, M., Sukumaran, S.: Regression test selection techniques: A survey. Informatica (Ljubljana) **35** (01 2011)
7. Briand, L., Labiche, Y., Soccar, G.: Automating impact analysis and regression test selection based on uml designs. In: International Conference on Software Maintenance, 2002. Proceedings. pp. 252–261 (2002). https://doi.org/10.1109/ICSM.2002.1167775

8. Briand, L., Labiche, Y., He, S.: Automating regression test selection based on uml designs. Information Software Technology **51**, 16–30 (01 2009). https://doi.org/10.1016/j.infsof.2008.09.010
9. Celik, A., Vasic, M., Milicevic, A., Gligoric, M.: In: Regression test selection across JVM boundaries. pp. 809–820 (08 2017). https://doi.org/10.1145/3106237.3106297
10. Chen, Y., Probert, R.L., Sims, D.P.: Specification-based regression test selection with risk analysis. In: Proceedings of the 2002 Conference of the Centre for Advanced Studies on Collaborative Research. p. 1. CASCON '02, IBM Press (2002)
11. Elbaum, S., Rothermel, G., Penix, J.: Techniques for improving regression testing in continuous integration development environments. In: Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering. p. 235245. FSE 2014, Association for Computing Machinery, New York, NY, USA (2014). https://doi.org/10.1145/2635868.2635910, https://doi.org/10.1145/2635868.2635910
12. Elsner, D., Hauer, F., Pretschner, A., Reimer, S.: Empirically evaluating readily available information for regression test optimization in continuous integration. In: Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis. p. 491504. ISSTA 2021, Association for Computing Machinery, New York, NY, USA (2021). https://doi.org/10.1145/3460319.3464834, https://doi.org/10.1145/3460319.3464834
13. Elsner, D., Wuersching, R., Schnappinger, M., Pretschner, A., Graber, M., Dammer, R., Reimer, S.: Build system aware multi-language regression test selection in continuous integration. In: 2022 IEEE/ACM 44th International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP). pp. 87–96 (2022). https://doi.org/10.1109/ICSE-SEIP55303.2022.9793870
14. Engström, E., Runeson, P., Skoglund, M.: A systematic review on regression test selection techniques. Inf. Softw. Technol. **52**(1), 1430 (jan 2010). https://doi.org/10.1016/j.infsof.2009.07.001, https://doi.org/10.1016/j.infsof.2009.07.001
15. Fischer: (1977)
16. Gligoric, M., Eloussi, L., Marinov, D.: Practical regression test selection with dynamic file dependencies. In: Proceedings of the 2015 International Symposium on Software Testing and Analysis. p. 211222. ISSTA 2015, Association for Computing Machinery, New York, NY, USA (2015). https://doi.org/10.1145/2771783.2771784, https://doi.org/10.1145/2771783.2771784
17. Hilton, M., Tunnell, T., Huang, K., Marinov, D., Dig, D.: Usage, costs, and benefits of continuous integration in open-source projects. In: Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering. p. 426437. ASE 2016, Association for Computing Machinery, New York, NY, USA (2016). https://doi.org/10.1145/2970276.2970358, https://doi.org/10.1145/2970276.2970358
18. John, O.: The financial cost of a checkin (part 1) (Nov 2013), https://oduinn.com/2013/11/20/the-financial-cost-of-a-checkin-part-1/
19. Kauhanen, E., Nurminen, J.K., Mikkonen, T., Pashkovskiy, M.: Regression test selection tool for python in continuous integration process. In: 2021 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER). pp. 618–621 (2021). https://doi.org/10.1109/SANER50967.2021.00077
20. Leung, H.K.N., White, L.: Insights into testing and regression testing global variables. Journal of Software Maintenance **2**(4), 209222 (dec 1991). https://doi.org/10.1002/smr.4360020403, https://doi.org/10.1002/smr.4360020403

21. Machalica, M., Samylkin, A., Porth, M., Chandra, S.: Predictive test selection. In: 2019 IEEE/ACM 41st International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP). pp. 91–100. IEEE (2019)
22. Mattis, T., Hirschfeld, R.: Lightweight lexical test prioritization for immediate feedback. ArXiv **abs/2002.06213** (2020)
23. Naslavsky, L., Ziv, H., Richardson, D.J.: A model-based regression test selection technique. In: 2009 IEEE International Conference on Software Maintenance. pp. 515–518 (2009). https://doi.org/10.1109/ICSM.2009.5306338
24. Orso, A., Apiwattanapong, T., Harrold, M.J.: Leveraging field data for impact analysis and regression testing. SIGSOFT Softw. Eng. Notes **28**(5), 128137 (sep 2003). https://doi.org/10.1145/949952.940089, `https://doi.org/10.1145/949952.940089`
25. Philip, A.A., Bhagwan, R., Kumar, R., Maddila, C.S., Nagppan, N.: Fastlane: Test minimization for rapidly deployed large-scale online services. In: 2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE). pp. 408–418 (2019). https://doi.org/10.1109/ICSE.2019.00054
26. Potvin, R., Levenberg, J.: Why google stores billions of lines of code in a single repository. Commun. ACM **59**(7), 7887 (jun 2016). https://doi.org/10.1145/2854146, `https://doi.org/10.1145/2854146`
27. Ren, X., Shah, F., Tip, F., Ryder, B.G., Chesley, O.: Chianti: A tool for change impact analysis of java programs. In: Proceedings of the 19th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications. p. 432448. OOPSLA '04, Association for Computing Machinery, New York, NY, USA (2004). https://doi.org/10.1145/1028976.1029012, `https://doi.org/10.1145/1028976.1029012`
28. Rothermel, G., Harrold, M.: A safe, efficient algorithm for regression test selection. In: 1993 Conference on Software Maintenance. pp. 358–367 (1993). https://doi.org/10.1109/ICSM.1993.366926
29. Rothermel, G.: Efficient, Effective Regression Testing Using Safe Test Selection Techniques. Ph.D. thesis, USA (1996), aAI9703440
30. Shi, A., Zhao, P., Marinov, D.: In: 2019 IEEE 30th International Symposium on Software Reliability Engineering (ISSRE). pp. 228–238 (2019). https://doi.org/10.1109/ISSRE.2019.00031
31. Sutar, S., Kumar, R., Pai, S., BR, S.: Regression test cases selection using natural language processing. In: 2020 International Conference on Intelligent Engineering and Management (ICIEM). pp. 301–305 (2020). https://doi.org/10.1109/ICIEM48762.2020.9160225
32. White, L., Narayanswamy, V., Friedman, T., Kirschenbaum, M., Piwowarski, P., Oha, M.: Test manager: A regression testing tool. In: 1993 Conference on Software Maintenance. pp. 338–347 (1993). https://doi.org/10.1109/ICSM.1993.366928
33. Yoo, S., Harman, M.: (1 2012)
34. Zhang, L., Kim, M., Khurshid, S.: In: Localizing failure-inducing program edits based on spectrum information. pp. 23–32 (09 2011). https://doi.org/10.1109/ICSM.2011.6080769
35. Zhang, L., Kim, M., Khurshid, S.: In: FaultTracer: a change impact and regression fault analysis tool for evolving Java programs (11 2012). https://doi.org/10.1145/2393596.2393642