Test Selection with Moose In Industry

Impact of Granularity

Vincent Blondeau[1,2] Nicolas Anquetil[1]
Stéphane Ducasse[1]

[1] Univ. Lille, CNRS, Inria, Centrale Lille,
UMR 9189 - CRIStAL,
F-59000 Lille, France
{firstname.lastname}@inria.fr

Sylvain Cresson[2] Pascal Croisy[2]

[2] Worldline,
Z.I A Rue de la Pointe
59113 Seclin, France
{firstname.lastname}@worldline.com

Abstract

Automatic testing constitutes an important part of everyday development practice. Worldline, a major IT company, is creating more and more tests to ensure the good behaviour of its applications and gain in efficiency and quality. But running all these tests may take hours. For this reason tests are not launched as often as they should and are mostly run at night. The company wishes to improve its development and testing process by giving rapid feedback to developers after a change. An interesting solution is to reduce the number of tests to run by identifying only those exercising the piece of code changed. Two main approaches are proposed in the literature: static and dynamic. The static approach creates a model of the source code and explores it to find links between changed methods and tests. The dynamic approach records invocations of methods during the execution of test scenarios. Moose, a tool allowing to create static models of source code is a good candidate to carry this approach. Thanks to the partnership created with Worldline, we investigate on three industrial, closed source, cases to compare static and dynamic approaches. We evaluate the impact on the results of the frequency of modification of methods or considering groups of methods instead of single ones. We found that considering commits instead of individual methods tends to worsen the results, perhaps due to their large size.

Keywords Test selection, Moose, Dynamic, Static, Industrial case

1. Introduction

Worldline, a major IT company, like many other companies (Ekelund and Engström 2015), is facing a contradiction: to guarantee high quality level solutions, developers write tests; however, tests become so numerous that several hours are required to run them all. In an industrial environment where each line of code has to be written as fast as possible, the developers often bypass the tests during the day and an automatic testing job is launched during the night to run all the tests. However, developers need feedback on the behaviour of their implementation as soon as possible to avoid spending time on potential future debugging. Wishing to improve its development process while improving the quality of the developed solutions, Worldline made a partnership with Inria RMod team for auditing its projects and provide solutions to reduce the feedback time of the tests and keep the test quality it provides to its client.

The envisioned solution consists in reducing the time needed to run the tests by reducing the number of tests. For a given change of the source code, some tests are not relevant because they do not cover the changed source. So, running only a subset of tests can be a solution to get faster feedback.

Among the known approaches to select a subset of tests to execute to verify a change in the code, we take interest in the dynamic and static ones. *Static approaches* consist in creating a model of the source code. This model can then be navigated, going up the chain of method calls, from a changed method back to the tests that exercise it. *Dynamic approaches* involve executing the tests and recording the methods invoked by each test. The test subset supplied by this approach is trivially composed of the tests executing (covering) a changed piece of code. Moose (Ducasse et al. 2000), a tool dedicated to static analysis source code, will be used to experiment a solution.

In this paper, we propose a concrete study of the consideration of commits for test selection on large industrial cases at Worldline. We studied the impact of the frequency of change of the methods on the overall results but also the impact of

considering commits (*i.e.,* groups of methods) instead of individual methods.

The contribution of this paper is to evaluate the impact of highly modified methods and considering groups of methods instead of individual ones. We found that considering commits instead of individual methods tends to worsen the results, perhaps due to their large size.

In Section 2, we present the test case selection problem and define the existing approaches. Then, in Section 3, we describe an experiment to evaluate the impact of each problem on the performances of an approach. Section 4 analyses and discusses of the results of the experiment on the closed source projects, Section 5 evaluates the validity of the experiment, and Section 6 presents the related works. Finally, we conclude in Section 7.

2. Problem Description

In large industrial projects, executing all tests after each change can turn into a costly operation requiring several hours. To get feedback on the changed code faster, it is important to reduce drastically this time. The solution generally adopted for this consists in trying to reduce the number of tests to run. A theoretical perfect approach would select only the tests demonstrating a flaw in the behaviour of the application after the change (called *modification-revealing tests* (Biswas et al. 2011; Yoo and Harman 2012)). But a real approach can only approximates this selection. For this, one usually concentrates on the tests covering the changed code (that is to say the tests that should lead to the execution of the changed code). The hope is that this real approach selects a suitable and small set of tests to detect a regression in the application behaviour.

2.1 The Problem of the Company

Worldline is the European leader in the payments and transactional-services industry. It is present in 17 countries across the globe with approximately 7500 employees. Worldline's end-to-end customized solutions help customers to optimize the performance of their digital transactions. Behind the scene, Worldline connects its clients with their customers through integrated and personalized digital services providing a seamless customer and citizen experience.

Tests are crucial for Worldline for different reasons. First, the company provides payment and transactional-services that are critical to its customers. Errors, bugs or denial of service are not allowed. Second, it provides solutions from design to deployment and maintenance. Maintainers can use information from test to help them understand, debug, and retest programs (Agrawal et al. 1998). However, running all the tests on a project may take hours. In a daily development process, developers can not run the tests after a change to check the impacts of their modifications. Since they have no tool to detect tests impacted by a change, they very often skip tests during the day and these only run at night thanks to continuous integration servers.

We decided with Worldline to improve this situation. We work closely with a transversal team that provides tools, expertise and support to the development teams. This team is aware of the issues met by the field teams and look for adapted solutions to simplify developers work while guaranteeing quality. The company provides real projects to analyse. The collaboration between academia and industry occurs mainly through the first author of this paper who is a PhD student paid by Worldline. To convince upper management of possibly imposing a change in work practices of thousands of developers, the transversal team needs convincing hard data on the pros and cons of the technique it will propose. We report in this paper some conclusions on our first studies.

2.2 Test Case Selection

Test case selection techniques seek to reduce the number of test cases to execute after modifying the code. The selection is not only temporary (*i.e.,* specific to the current version of the program) but also focused on the identification of the modified parts of the program. Test cases are selected because they are relevant to the changed parts of the system under tests (Yoo and Harman 2012).

Test case selection approaches are based on the notion of dependency graph. The general idea is that tests can be said to depend on the source code that they exercise. After a piece of code is changed, a test case selection technique just needs to go back from this piece to the tests that depend on it. Figure 1 illustrates this principle for two methods and two tests. `testMethod1` depends on `method1` and `method2` (for example `testMethod1` calls `method1` and `method2`), `testMethod2` depends on `method2`.

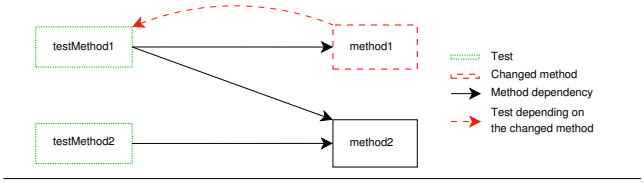

Figure 1. Test Selection Simple Case

This is, of course, a simplified example, in real cases, the dependency graph is much larger and deep, or, some other factors may make it very difficult for a given approach to find out which tests depend on a piece of code. In this paper, we investigate what is the impact of these factors.

2.3 Dynamic and Static Approaches

Literature (*e.g.,* Engström et al. (2008, 2010); Ernst (2003)) recognizes several types of approaches for test case selection: some are based on source code introspection (*e.g.,* dynamic) or code analysis (*e.g.,* static), others are using specifi-

cations, meta-data or UML models. We will compare a main approach: the static to other known ones.

The *dynamic approach* consists in executing the tests and recording the code executed during each test. This is the execution trace of a test. A test depends on a piece of code if this piece of code is in its execution trace. For example, using this approach on the Figure 1, will result with a mapping for `method1` to `testMethod1`, and for `method2` to `testMethod1` and `testMethod2`. If `method1` is modified, the mapping will be used to relaunch only `testMethod1`.

The *static approach* does not require executing the tests. It relies on computing the dependency graph from the source code or some representation of it (*e.g.,* bytecode for Java). For example, by considering arrows in Figure 1 as calls between methods, a static approach will create a model of the source code where `method1` callers are `testMethod1`, and `method2` callers are `testMethod1` and `testMethod2`. If `method1` is modified, the `testMethod1` caller will be relaunched. Note that in real projects, the call graph is much deeper and recursion is used until a test is found.

Different kind of granularity can be considered (Engström et al. 2010) from functions/methods (*e.g.,* Elbaum et al. (2003); Zheng et al. (2007)) to external components (*e.g.,* Willmor and Embury (2005)) passing through classes (*e.g.,* White et al. (2005); Hsia et al. (1997)).

2.4 Known Issues

While experimenting with some existing test case selection tools, we were confronted with different issues: Problems in test case selection approaches arise when there is a break in the dependency graph representing the system. Such breaks may occur for several reasons. In our case, we identified four categories of reason. We detailed these issues in a previous paper (see Blondeau et al. (2015)). But they are summarized here:

Third-party breaks: The application uses external libraries or frameworks for which the source code is not available. In this case, a static analysis of the code cannot trace dependencies through the third-party code execution.

Multi-program breaks: The application consists in several co-operating programs (*e.g.,* client/server application). In this case, an analysis focused on one single program cannot trace dependencies into the other program.

Dynamic breaks: The application contains code treated as data (*e.g.,* lambda-expressions or reflexive API). Specific instructions allow to execute this code in another location than its definition. In this case, an analysis of the source code cannot yield the dependencies that will occur at execution.

Polymorphism breaks: The application uses polymorphism. In this case, a dependency analysis may reach a class on which nobody else depends because all dependencies point to a superclass of it.

3. Experimental Setup

As Worldline is considering putting in practice test case selection, we experimented with large representative projects of this company, written in Java, with the idea of evaluating the impact of the consideration of commits on test selection results. For this purpose, Moose and dynamic open-source test selection tool have been used. This section presents the tools and software projects used to carry out these experiments.

3.1 Experimental Protocol

We made our experiments at method level. The idea was to strike a balance between accuracy of the selection on one hand and processing time and data size on the other hand. This choice is discussed in the threats to validity (Section 5).

We will be looking for the answers to the following Research Questions:

RQ1: What is the impact on the test selection results of changing the same methods repeatedly (as occurs in real life)?

RQ2: What is the impact on the results of considering real commits (that change several methods jointly)?

Problems of Section 2.4 proper to Worldline projects and impacting the static approach are resolved by the Moose approach considered in this paper.

The experiment follow the same pattern that we will illustrate with RQ1. To assess the impact of the third-party breaks we will compare the results of two similar experiments: one bypassing the problem, the other not. Results on the impact of the third-party breaks issue involve:

i. We fixed one version of the source code on which we work. This version never changes, all changes are virtual. More or less we will ask the question "If Java method `m()` was changed, would we be able to recover the tests that cover it?" This decision was necessary because fetching the source code and the dependencies, recompiling, and running the tests for one version is resource intensive and could not be computed in reasonable time and space for such a large experiment.

   Test coverage is given by the dynamic approach (Jacoco tool) that is our baseline. The coverage from Jacoco for the baseline is perfect because our version of the source code never actually changes. The baseline is only computed once.

ii. We consider as "changed" each and all Java method of the application that is covered by at least one test. Considering Moose static approach, we try to recover the test cases covering this Java method.

iii. From the test cases recovered, we compute different metrics (see Section 3.4)

iv. The metrics values are averaged over all Java methods (covered by at least one test) to produce a result for the static approach considered.

v. The same process is repeated for another static approach and we compare their respective results to answer the research question. The difference in the results is considered as the impact of the problem that one of the two static approaches solves.

To answer RQ1, we apply Moose on all Java methods. The difference in the two static approaches is in the way the metrics results are averaged (Step iv.). Here, we use a weighted mean where each Java method has a weight corresponding to the number of commits (in the history of the system) where it appears in. A Java method must appear in at least in one commit (at its creation) but may be modified frequently (more than one hundred times for some cases). The weighted mean is considered more realistic as in any system, all methods are not changed with the same frequency.

For RQ2, we still apply Moose on all Java methods in one case and all system commits in the other case. As for RQ1, we use all commits in the history of the system that touched at least one method appearing in the version of the code we use. Commits differ from individual methods in that they may change many Java methods (up to 125 in one case). We study past commits on one single code version where oracle is calculated. As commit impacts several methods, we consider the union of the selected tests for each method to compute the metrics. We believe this is acceptable because they still indicate that several Java methods were changed together and we would like to know if we would be able to recover all the tests if that were the case again.

3.2 Projects

To perform our experiments, we selected three projects (P1, P2 and P3). P1 and P2 are financial applications with more than 400 KLOC. P1 is a service (in term of Service Oriented Architecture (SOA)) dealing with card management. P2 is an issuing banking system based on SOA and reusing the card management system developed in P1 (P2 uses P1 as a third party). P3 has no relation with the two other projects, and is an e-commerce application. P2 and P3 test suites are mainly composed of integration tests, that ensure the good behaviour of the application with its dependencies and the data base. P1 test suite includes mainly unit tests that guarantee the results of the algorithms. In these projects, each test is a Java method using JUnit[1]. P1 includes 5,323 valid tests; P2, 168; and P3, 3,035. We were surprised to find that some tests in each project were failing outright. This impedes to use the dynamic approach (no execution trace) and thus to use them in our experiments. P1 and P3 (2009) are 6 years older than P2 (2015), with respectively 2,217, 467, and 2,115 commits that we considered. P2 is actually a rework on some legacy code dating back from 2010. However, we were not able to recover the commits from the early version of the project. These projects are still alive and evolving regularly. P1, P2 and P3 have respectively 2,217, 467, and 2,115 commits that we considered. Commits seem rather big: 100+ methods, 18 files. One of the possible and positive outcome of our whole research project would be to see the developers doing smaller commits that would be easier to test. Test execution (compilation and test execution included) requires 3 hours for P1; 2 hours for P2; and 30 minutes for P3.

3.3 Moose and Other Approaches Tooling

To analyse the source code, Moose[2] (Ducasse et al. 2000) relies on the FAMIX meta-model (Ducasse et al. 2011) and proposes to represent source code entities in a model. This model gathers entities such as packages, classes, methods, and the links between them (invocations, references, inheritances, and accesses); statements are omitted. A method dependency graph, linking a changed method to the tests, is thus available.

Moose is a tool dedicated to pure static analysis, and so does not need any compilation of the source code. However, the source code has to be parsed to create the model. This parsing can take up to several minutes for large applications. In our experiment, the result overhead was smaller than for the other approaches.

For the dynamic approach, we used a coverage tool named Jacoco[3] (Lingampally et al. 2007). This tool aims to compute the test coverage of an application. For this purpose, the Java Virtual Machine (JVM) is instrumented by adding an agent to add behaviour to the source code and to record method dependencies during the tests execution. No recompilation nor modification of the source code is needed. However, a synthesis of the results is needed after the execution. It can have an impact on the execution time. For the studied projects, about one hour is needed for this tests coverage synthesis for each project.

However the data provided by Jacoco is not directly usable for our experiments. Information concerning the different tests are mixed up. We modified the tool in order to separate information relative to each test and thus know what test cover what Java methods.

3.4 Metrics

An optimal approach selects the smallest set of all test cases covering a change. According to the experiment protocol (Section 3.1), the methods are not changed but considered as such each in turn. As a consequence, the dynamic approach

[1] http://junit.org/

[2] http://www.moosetechnology.org/

[3] http://eclemma.org/jacoco/

is both precise and safe: all the tests selected by the dynamic approach cover the changes; no other test covers a given change.

Due to the issues we identified, static approaches may miss some tests covering a change and select others that do not cover it. To compare the approaches, we use four metrics that can be computed from the traditional quantities:

- **True Positives** (TP) is the number of tests selected by the static approach and the dynamic approach;
- **False Positives** (FP) is the number of tests selected only by the static approach;
- **False Negatives** (FN) is the number of tests selected only by the dynamic approach;
- **True Negatives** (TN) is the number of tests selected neither by the dynamic nor by the static approaches.

From these quantities we compute the following metrics: *Selected tests*, *Precision*, *Recall*, and *F-Measure*.

Selected tests represents the number of tests selected by the approach as a ratio. It compares the number of selected tests to the total number of tests.

$$Selected\ tests = \frac{TP + FP}{TP + FN + TN + FP}$$

Precision is the fraction of retrieved tests that covers the modified method. A high precision means that the static approach selects essentially tests that cover the changed method.

$$Precision = \frac{TP}{TP + FP}$$

Recall is the fraction of tests that are covering the modified method that are retrieved. A high recall means that the approach is safe and that the tests covering a given changed method are selected by the static approach.

$$Recall = \frac{TP}{TP + FN}$$

F-Measure is the harmonic mean of *Precison* and *Recall* to show the overall performance of an algorithm.

$$F\text{-}Measure = \frac{2 * Recall * Precision}{Recall + Precision}$$

Priority will be given to a higher *Recall* to make sure one does not overlook a test that could discover a bug in the changed code. However, achieving good recall is easy by selecting many tests. This would show up in the *Precision* and *Selected tests* metrics. We must remember that our goal is ultimately to be able to give rapid and useful feedback to the developer right after he commits a change. So, better *Precision* would mean no useless test run. But *Precision* typically comes at the expense of *Recall*, so good *Precision* could also mean some needed tests would not be run, and, therefore, we could not give any guarantee to the software engineers about the quality of their development. This would defeat the purpose of testing the application.

4. Results and Discussion

This section presents our experimental results to answer the seven research questions. Table 1 gives the metrics for each static approach.

4.1 RQ1 – Weighting of Results with the Number of Commits

In these experiments, the results of each methods are weighted according to the number of commits the method appears in (Section 3.1). The idea is that the most committed Java methods could have consistently good (or bad) results.

In summary, the results are not very different. This new experiment consistently brings marginal decrease in *Precision* and *Recall* and small increase in *Selected tests*. Since the value for one given method is the same in both experiments, the difference in average can only come from the weighting of the methods results and, therefore, it seems that "bad" methods would have higher weight.

This would suggest that the methods where static approaches only select the wrong tests are more frequently committed. This is not good news, but the differences are small (typically one percentage point) and would need to be more formally tested in a specific experiment.

4.2 RQ2 – Aggregation of the Results by Commit

To answer this last Research Question, we again replicate the experiment, but working with commits instead of individual methods. All the results are summarised in Table 1.

The first observation regards the number of *Selected tests*. Since commits comprise many Java methods (average for P1 is 24, for P2 it is 129), it is expected that more tests would be selected. This is the case with our baseline (Jacoco) with a larger percentage of all tests selected (P1, from 0.8% to 8%; P2, from 1% to 21%; P3, from 0.4% to 14%). This is an increase in the range of an order of magnitude. However, we see that the static approaches (mainly at the method granularity level) tend to exhibit a smaller increase in the number of selected tests (P1, from 3% to 4%; P2, from 0.8% to 3%; P3, from 2% to 6%).

So even-though the static approaches selected more tests, one could conclude that they are actually more selective than necessary here: P1 and P3 improve their *Precision* (resp. from 43% to 55%; and from 34% to 49%), but P2 decreased (from 61% to 45%). So good news for P1 and P3, that are more selective but also more precise.

Being more selective, the *Recall* results were bound to worsen: the approaches select less tests than they should according to our baseline. This is what happens with P1 and P2 (resp. from 91% to 81%; and from 64% to 45%), but P3, which previously had the lower *Recall*, improved it (from 41% to 56%).

Table 1. Comparison of the static approaches to the dynamic one for test case selection considering all Java methods individually

		Selected Tests			Precision			Recall			F-Measure		
		P1	P2	P3	P1	P2	P3	P1	P2	P3	P1	P2	P3
All methods individually	Jacoco	0.8%	1%	0.4%	-	-	-	-	-	-	-	-	-
	Moose	3%	0.8%	2%	43%	61%	34%	91%	64%	41%	50%	62%	29%
Weighting of methods with the number of commits	Jacoco	1%	2%	1%	-	-	-	-	-	-	-	-	-
	Moose	3%	1%	2%	42%	59%	33%	92%	62%	39%	50%	60%	28%
Methods grouped in commits	Jacoco	8%	21%	14%	-	-	-	-	-	-	-	-	-
	Moose	4%	3%	6%	55%	45%	49%	81%	45%	56%	56%	40%	42%

One conclusion is that it does not seem to be the case that one Java method in a commit "covers" for another one. That might indicate that the commits touch various concerns for which different subsets of tests are necessary. That would be coherent with the large size of the commits that we already mentioned.

4.3 Overall Conclusions

To conclude on these experiments, considering commits instead of individual Java methods tend to worsen the results with approaches that are too selective to keep the same level of good results. The large size of the commits might be an important factor in this behaviour. As already stated, a positive consequence of the application of the test case selection in the environment of Worldline's developers (still in progress) would be to see them make smaller commits and would also give them better and faster feedback on their changes. We hope that this work changes the developers habits and that they develop, launch selected tests (as it takes less time), and commit just after if tests pass. Future work will validate these hypotheses.

For our experiments, we used the dynamic approach as oracle because it is safe and accurate. However, this approach has two major drawbacks: First, this approach is not generic; it depends strongly on the data used for the tests. Second, if a test is failing, no execution trace is recorded and it cannot be selected by this approach.

5. Evaluation of Validity

This section discusses the validity of our case study using validation scheme defined by Runeson and Höst (2009).

5.1 Construct Validity

Construct validity indicates whether the studied operational measures really represent what is investigated according to the research question. The purpose of this study is to evaluate pros and cons of different approaches for test case selection and compare static and dynamic approaches to select the optimal test set to execute after a change in the application.

Metrics Validity. Four metrics have been chosen: the ratio of selected tests, the *Precision*, the *Recall* and the *F-Measure*. These metrics are considered relevant in Biswas et al. (2011) and Engström et al. (2010).

A point related to the time gain one may hope from test selection is linked to the initialization of the tests. A part of the total time to test is caused by long initialization. This initialization can occur for the entire test suite, for each test-class or each test-method. Selecting tests has the potential to reduce the both initializations.

Granularity level Validity. For these experiments, we mainly considered a method granularity. We could have also used a class granularity. But, by comparing both approaches, the cost reduction for Moose/class over Moose/method is much worse: up to 10 times more of tests are selected. The precision is also much worse, but, the recall is improved. As we want to have a high *Recall* and a small set a tests selected, we have no clear argument yet to choose the best granularity level.

5.2 Internal Validity

Internal validity indicates whether no other variables except the studied one impacted the result.

There was an issue with anonymous classes that cannot be identified by their name (obviously!) from one experiment to the other. For this reason, their methods have not been considered as "changed" in the experiments.

We have been careful to test every solution independently of the other before combining them. We are confident that our small modifications of Infinitest and Jacoco did not introduce unwanted errors in the results of these two tools as we did not touch the algorithms part but how they are run on the tests.

Finally, Tengeri et al. (2016) argue that Jacoco which is based on bytecode instrumentation may produce erroneous results compared to source code instrumentations methods. Jacoco misses some really covered methods. But as we consider only covered methods to retrieve the tests, the impact is reduced.

5.3 External Validity

External validity indicates whether it is possible to generalize the findings of the study.

We are fully aware that our results cannot be easily generalized as such and took precaution to present them in their true context. This gave a richness of information and possible interpretations of the phenomenon under study. The results presented in this paper involves only three Java projects of one company. These three projects are different considering their size, number of tests, test coverage, and used frameworks and annotations. Moreover, they face different issues (*e.g.,* P1 does not present the attribute initialization issue).

5.4 Reliability

Reliability indicates the extent to which the data and analysis are dependent on a specific researcher.

The tools used for the experiments are all open-source. However, each of them had to be altered to ease the gathering and interpretation of results. Moreover, we used a small set of projects of the company that is, for us, representative of Worldline ecosystem and is different of the ones used by other researchers.

6. Related Works

Two other studies (Soetens et al. 2013; Ekelund and Engström 2015) implement tests selection approaches and provide the same metrics as for our experiment (selected tests, precision, and recall). Table 2 gathers the results of these studies.

Table 2. Comparison of the static approaches to the dynamic one to select the tests after a method change

	Project/Approach	Selected Tests	Precision	Recall
Our Approach	P1	4%	55%	81%
	P2	3%	45%	45%
	P3	6%	49%	56%
Soetens et al.	PMD	1%	83%	58%
	Cruisecontrol	1%	87%	77%
Ekelund and Engström	Wide approach	37%	1.5%	95%
	Narrow approach	4%	7.4%	79%

Soetens et al. (2013) propose a static approach at method granularity based on the FAMIX meta-model. Their approach relies on real change sets gathered in commits. For each of these change sets, their static approach is compared to a dynamic one used as reference. The dependency graph they use only contains links between methods. Two open-source applications (PMD and Cruisecontrol) are used as input data for their experiment. 1% of the test cases is selected for both applications. They obtain respectively for each application a recall of 77% and 58% and a precision of 84% and 83%. These results are better than our Moose/method approach. First, P1 and P2 are wider projects than PMD and Cruisecontrol that counts around 20 less times lines of code. Second, P1 and P2 use a lot of frameworks and libraries what seems not the case for PMD and Cruisecontrol. However, by solving the identified issues, we obtain close results to Soetens et al.: despite our Precision is lower (close to 30% less), we achieve a better recall for P1 with 81%. So, resolving the issues seems to improve the recall, to select more tests and to lower the precision.

Ekelund and Engström (2015) select tests based on test result history. This history archives changes at package granularity and corresponding test results for each build of the application (*i.e.,* the execution of all the test cases by a continuous integration server). Such an approach has been defined since no other existing approach based on source code, bytecode, or dynamic analysis was possible due to the huge size of the studied application that counts several million of lines of code. When a package changes, thanks to history data mining, the authors know the potentially affected tests and select them. These tests have at least once failed when, in the past, this package changed (narrow approach) or whatever the package changed (wide approach). The accuracy of the selection algorithm is related to the number of builds used. However, considering a too large history may introduce noise in the selection mechanism since the source code may have evolved a lot. The authors found that the algorithm is optimal for a history containing 100 builds. This approach is language independent and uses few resources but relies on a history of the build results. Such a history does not often exist in companies and requires time and effort to be built. In the case of the wide approach, the ratio of selected tests reaches 37%, the precision and the recall are respectively 1.5% and 95%. In the case of the narrow approach, only 4% of the tests are selected with a precision of 7.4% and a recall of 79%. The experiments of Ekelund and Engström lead to recall with the same order of magnitude than our project P1. However, precision results are very low, because they work at package level.

7. Conclusion

Testing an industrial application can take several hours whereas quality would require to test often. Reducing the testing time is thus essential for companies. One solution is to reduce the number of tests to execute by selecting only tests that may possibly fail after a change.

In a partnership with a major IT company we looked how developers could get faster feedback when modifying some code. We experimented different approaches on three of their Java projects counting several thousand of lines of code. During this experiment, we met several issues that we generalized and categorized. Solutions to these issues were also proposed and implemented.

From the experiment we carried out, we draw that considering commits instead of individual methods tends to worsen the results, perhaps due to their large size, but the results are still acceptable because of low number of selected tests.

The contributions of this research for the company with which we are working will be a better understanding of how tests are used by its developers and a tool to help them get faster feedback from tests when they make a change.

As future work, we identified some issues that cannot be solved with a static or dynamic approach alone. We plan to investigate these pending issues by experimenting a hybrid solution. Moreover, we foresee to conduct such experiments on real changes.

Acknowledgements

This work was supported by Worldline and by Ministry of Higher Education and Research, Nord-Pas de Calais Regional Council, CPER Nord-Pas de Calais/FEDER DATA Advanced data science and technologies 2015-2020.

References

H. Agrawal, J. L. Alberi, J. R. Horgan, J. J. Li, S. London, W. E. Wong, S. Ghosh, and N. Wilde. Mining system tests to aid software maintenance. *IEEE Computer*, 31(7):64–73, 1998. doi: 10.1109/2.689678. URL http://dx.doi.org/10.1109/2.689678.

S. Biswas, R. Mall, M. Satpathy, and S. Sukumaran. Regression test selection techniques: A survey. *Informatica (03505596)*, 35 (3), 2011.

V. Blondeau, S. Cresson, P. Croisy, A. Etien, N. Anquetil, and S. Ducasse. Dynamic and Static Approaches Comparison for Test Suite Reduction in Industry. In *BENEVOL'15: 14th BElgian-NEtherlands software eVOLution seminar*, Lille, France, Dec. 2015. URL https://hal.inria.fr/hal-01317498.

S. Ducasse, M. Lanza, and S. Tichelaar. Moose: an Extensible Language-Independent Environment for Reengineering Object-Oriented Systems. In *Proceedings of the 2nd International Symposium on Constructing Software Engineering Tools*, CoSET '00, June 2000. URL http://scg.unibe.ch/archive/papers/Duca00bMooseCoset.pdf.

S. Ducasse, N. Anquetil, U. Bhatti, A. Cavalcante Hora, J. Laval, and T. Girba. MSE and FAMIX 3.0: an interexchange format and source code model family. Technical report, RMod – INRIA Lille-Nord Europe, 2011. URL http://rmod.inria.fr/archives/reports/Duca11c-Cutter-deliverable22-MSE-FAMIX30.pdf.

E. D. Ekelund and E. Engström. Efficient regression testing based on test history: An industrial evaluation. In *International Conference on Software Maintenance and Evolution*. IEEE Computer Society, 2015.

S. Elbaum, P. Kallakuri, A. G. Malishevsky, G. Rothermel, and S. Kanduri. Understanding the effects of changes on the cost-effectiveness of regression testing techniques. *Journal of Software Testing, Verification, and Reliability*, 2003.

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*, pages 22–31. ACM, 2008.

E. Engström, P. Runeson, and M. Skoglund. A systematic review on regression test selection techniques. *Information and Software Technology*, 52(1):14–30, 2010.

M. D. Ernst. Static and dynamic analysis: Synergy and duality. In *WODA 2003: ICSE Workshop on Dynamic Analysis*, pages 24–27. Citeseer, 2003.

P. Hsia, X. Li, D. Chenho Kung, C.-T. Hsu, L. Li, Y. Toyoshima, and C. Chen. A technique for the selective revalidation of oo software. *Journal of Software Maintenance: Research and Practice*, 9(4):217–233, 1997.

R. Lingampally, A. Gupta, and P. Jalote. A multipurpose code coverage tool for java. In *System Sciences, 2007. HICSS 2007. 40th Annual Hawaii International Conference on*, pages 261b–261b, jan 2007. doi: 10.1109/HICSS.2007.24.

P. Runeson and M. Höst. Guidelines for conducting and reporting case study research in software engineering. *Empirical software engineering*, 14(2):131–164, 2009.

Q. D. Soetens, S. Demeyer, and A. Zaidman. Change-based test selection in the presence of developer tests. In *Software Maintenance and Reengineering (CSMR), 2013 17th European Conference on*, pages 101–110. IEEE, 2013.

D. Tengeri, F. Horváth, Á. Beszédes, T. Gergely, and T. Gyimóthy. Negative effects of bytecode instrumentation on Java source code coverage. In *Proceedings of the IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER'16)*, pages 225–235, Mar. 2016.

L. White, K. Jaber, and B. Robinson. Utilization of extended firewall for object-oriented regression testing. In *Software Maintenance, 2005. ICSM'05. Proceedings of the 21st IEEE International Conference on*, pages 695–698, sep 2005. doi: 10.1109/ICSM.2005.101.

D. Willmor and S. Embury. A safe regression test selection technique for database-driven applications. In *Software Maintenance, 2005. ICSM'05. Proceedings of the 21st IEEE International Conference on*, pages 421–430, sep 2005. doi: 10.1109/ICSM.2005.15.

S. Yoo and M. Harman. Regression testing minimization, selection and prioritization: a survey. *Software Testing, Verification and Reliability*, 22(2):67–120, 2012. ISSN 1099-1689. doi: 10.1002/stvr.430. URL http://dx.doi.org/10.1002/stvr.430.

J. Zheng, L. Williams, B. Robinson, and K. Smiley. Regression test selection for black-box dynamic link library components. In *Incorporating COTS Software into Software Systems: Tools and Techniques, 2007. IWICSS '07. Second International Workshop on*, pages 9–9, may 2007. doi: 10.1109/IWICSS.2007.8.