Dynamic and Static Approaches Comparison for Test Suite Reduction in Industry

Vincent Blondeau [1,2] Sylvain Cresson[1] Pascal Croisy[1] Anne Etien[2] Nicolas Anquetil[2] Stéphane Ducasse[2]

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

[2] RMod team, Inria Lille Nord Europe,
University of Lille, CRIStAL, UMR 9189
59650 Villeneuve d'Ascq, France
{firstname.lastname}@inria.fr

***Abstract*—Automatic testing constitutes an important part in everyday development practice. IT companies are creating more and more tests to ensure the good behaviour of their applications and gain in efficiency and quality. But running all these tests consumes developer time (hours). This is especially true for the use of large systems involving, for example, the deployment of a web server, or the communication with a database. For this reason tests are not launched as often as they should. Reducing this testing time is a main concern for developers in order to get quick feedback after a change. An interesting solution is to reduce the number of tests to run by identifying those exercising the piece of code changed. Two main approaches seem to be distinguished in the literature: the static and the dynamic. The static approach creates a model of the system and explores it to find the links between the changed methods and the tests. The dynamic approach records the invocations of methods during the execution of test scenarios. We experimented these approaches on several industrial, closed source, cases to understand the strengths and weaknesses of each solution.**

I. Introduction

In industry, the need to test every piece of code becomes compulsory. But developers conceive so many tests that some hours may be necessary to run them all. Developers have to wait to know whether the functionality just implemented impacts the tests. In an industrial environment where each line of code has to be written as fast as possible and where the code has to be flawless, this waiting is not bearable. As a consequence, the developer often bypasses the tests during the day and an automatic testing job is launched during the night to run all the tests. Developers need feedbacks on the behaviour of their implementation as soon as possible to avoid spending time on potential future debugging. A solution consists in reducing the number of tests to exercise. In the test suite, some tests can not fail because they are not impacted by the changes made in the source code. So, running only a subset of the tests can be suitable.

According to literature, two main approaches, the dynamic and the static one, are relevant to select the suitable subset of tests. These two approaches seem opposite but are actually complementary [Ernst, 2003]. The *static approach* consists in creating a model of the source code This model is then navigated from a changed method back to the tests that exercise it, going up the chain of method calls. The *dynamic approach* involves executing the tests and recording the methods invoked by each test. The test subset supplied by this approach is trivially composed by the tests executing a line of the changed method. Both approaches have their strengths and weaknesses By suggesting a static approach, Frechette et al. [2013] see several advantages: no instrumentation nor an execution of the source code is involved. Only the source code is required. Engström et al. [2008] found no superior test selection approach. From this postulate, we decided to compare both common approaches to find the more adapted to an industrial context. On several industrial projects, we experimented with both approaches to understand the pros and cons of each.

In this paper, we experiment with the dynamic approach, based on code coverage, and the static approach, analysing the source code and the executable code. Both were studied at class and method levels. Several large industrial systems are used. We discovered several problems related to object oriented conception of applications.

In Section II, we present the test suite selection problem and define the existing approaches. Section III, we detail the problems static and dynamic approaches raise. Then, in Section IV, we characterise the tools and the data used to conduct our experiment. Section V analyses the results of the experiment on the closed source projects, discusses and compares them. Finally, we conclude in Section VI.

II. Problem Description

As launching all tests after each change is a costly operation. Reducing the number of tests to run avoids this cost. A theoretical flawless approach selects only the tests ensuring the after-change behaviour of the application, *i.e.,* the tests failing after a change. But a real approach only approximates this selection. The hope is that this real approach selects suitable tests to detect a change in the application behaviour. The result of these tests can emphasize a misconception in the new version of the application or in the tests.

A. Test Selection

The test selection consists in choosing tests that are related to a change in the source code. Especially, it is associated to

the concept of test coverage. A test covers a method if the test executes one of the method instructions.

Figure 1 illustrates this principle. `testMethodT1` covers `methodC1` and `methodC2`, while `testMethodT2` covers only `methodC2`. If a change happens in `methodC1`, an ideal test selection approach selects `testMethodT1` whereas `testMethodC2` is not selected.

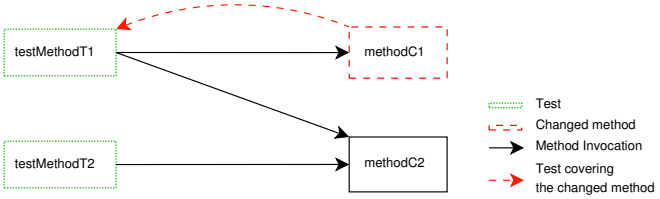

Fig. 1. Test selection simple case

The challenge is to establish an approach selecting only the tests covering the changed source code, without choosing any other irrelevant.

B. Dynamic Approach

The dynamic approach consists in executing the tests and recording the methods invoked during each test. A mapping between the tests and the methods covered is then built. From this mapping, the tests covering a changed line of code are easily identified.

Dynamic approach accurately selects the tests initially covering the line of code, *i.e.,* before the change. This accuracy results from the recording of the real invocations of methods. However this approach dismisses any modification resulting from a pure code addition or from modification. Indeed, the new methods invocations were not recorded during the test execution. This information is obtained by a new execution of the whole test suite.

C. Static Approach

The static approach does not require to execute the tests to perform the selection. However, a code representation is needed to discover the link between the methods changed and the tests. The source code itself is either abstracted in a model or in the compiled class code.

In the first case, the representation contains the code entities with the links between them. The links can be accesses, references, inheritances, or invocations. In case of large systems, representing the whole code can be huge. Actually, several approaches to select the tests are existing. They can be performed at different levels of abstraction in the source code: lines, methods, classes, or packages. They can give different results. To know whether a method is covered by a test, the links in the model are crossed back from the changes to all theirs potential callers. Among these callers, tests are selected.

In the second case, each compilation unit defines in a header all the classes it references. The approach based on this representation analyses theses references. Then, they are handled in a graph where the nodes are code entities and the edges are references between them. From this graph and given a modified method, a compiled class approach traces back the references until the test classes.

III. Approaches Limitations

In contrast to dynamic approach, the static one is not flawless. This section details the risen problems and brings solutions to solve them.

As we experiment on Object Oriented Language, and particularly Java, we encounter specific problems due to its features, like the interfaces and the anonymous classes.

A. Interfaces

Context. In Java, developers frequently use interfaces to define a contract between the implementation and the client.

Issue. The client of the interface invokes all the methods on the interface and not on the implementation. So, a static approach cannot identify a calling link between the client of the interface and the implementation. The interface blocks the resolution of the invocation.

Proposed solution. When an interface implementation is encountered, the method is looked for in the interface and the navigation continues from it.

Other similar context. This problem is also encountered in an inheritance context. The subclass can, like the interface implementation, override a superclass method. In this case, the static approach cannot link tests to the changed method.

B. Anonymous classes

Context. Java allows to use anonymous classes to define specific behaviour. An anonymous class is a class defined in a method. As any other class, the anonymous class may contain methods. These methods are callbacks, *i.e.,* their code is not executed immediately but triggered at a convenient moment. In this case, anonymous class methods define behaviours that are not called directly but through external implementations or frameworks.

Issue. The static approach does not know the implementation of the external code. A approach based only on methods invocations does not create a link between the container method and the anonymous method. It is a containment link, not an invocation one. So when one traces back the invocation calls, one stops at anonymous class methods.

Proposed solution. To set the callback, the container method should be called. Therefore, the test that covers the container method can also exercise the methods of the anonymous class. To retrieve the tests when this kind of method is changed, a solution is to jump from the changed anonymous class method to the container method. Both call graphs can be linked, and the test found.

Other similar context. This anonymous class problem is a subset of the external code problem. If the external code is not available, the invocation links cannot be resolved and the static approach is not efficient.

C. Attribute Direct Access

Context. In some cases, an instance variable can be called without using setter and be initialized by a method call in its declaration.

Issue. If the method called for the initialization is changed, the change impacts the instance variable. As it is a variable, there is no invocation but accesses. Whereas a dynamic approach resolves this problem directly, a static one have to recognize this concept.

Proposed solution. These accesses can be bypassed by retrieving the methods accessing these instance variables. From these methods, the approach continues navigating on the invocations until tests are encountered.

IV. Experimental Setup

In this section, we present the tools and software projects that we use to carry out our experiments. To compare static and dynamic test selection approaches on these projects, we use existing open source tools. They already implement the approaches.

A. Material

1) Projects: We worked with a major IT company. This company has projects exercising tests for hours. We took some of them to perform our experiments. We call them P1 and P2. Written in Java, they are complex applications with diversified tools & frameworks (EJB, Hibernate, Spring, Tomcat, ...). P1 and P2 are composed respectively of 631 and 764 KLOC.

The project P1 is an application where few libraries are used. On the other hand, the project P2 uses frameworks. This usage of external code can have an impact on test selection. In these projects, the tests are written with JUnit [JUnit].

2) Dynamic approach tooling: For the dynamic part, we used a coverage tool named Jacoco[1] [Lingampally et al., 2007]. This tool aims to record invocations of methods. During the tests execution, Jacoco is recording for each test the invoked methods. A mapping between the tests and the methods covered is obtained. This approach is not invasive and no recompilation nor modification of the source code is needed.

3) Static approach with bytecode tooling: For the static approaches, we used two tools. The first, Infinitest [2] analyses the bytecode. It runs automatically the tests which covers the changes. But, the algorithm is slow and approximate on complex projects. Infinitest uses the header of the compiled classes. It contains all the references made to external classes, *i.e.,* the names of classes of the invoked methods, the classes used, the annotations,... From this analysis, it is possible to construct a graph of the classes dependencies. This graph is crossed recursively to know which classes are referenced, and especially the tests classes. The test selection consists in fetching all the tests that are included in these test classes.

4) Static approach with source code analysis tooling: The second, Moose [3] [Ducasse et al., 2000], allows to make analysis on source code. It is based on the FAMIX meta-model [Ducasse et al., 2011]. Moose allows any kind of software analysis. For our purpose, this tools model call graphs of the source code from a changed method to the test ones. We modified the algorithm based on call graphs to take into account limitations described in Section III.

B. Metrics

To compare the approaches, we defined metrics that are representative of the objectives we want to achieve. An optimal approach selects less tests as possible but with a high rate of coverage.

The first metric is the *Number of tests executed*, it measures the number of test cases selected by the approach.

The next metrics are the *Precision* and the *Recall*. The *Precision* is the fraction of retrieved tests that are relevant, while *Recall* is the fraction of relevant tests that are retrieved.

These metrics are computed for every covered method of the application, and then agglomerated in one value, the average.

Our objective is to have the less tests selected and a good recall. We choose between the static approaches the one which optimize these both criteria.

C. Experiment Protocol

We considered one after the other each covered method of the application as changed. Actually, if several changes are made in the application, we acknowledge that the tests to select are the union of the tests selected for each method.

We decided to set the dynamic approach as an oracle. As the test coverage is recorded in this approach, any method of the application knows which tests can be impacted by its change. The dynamic approach is accurate in the tests selection.

We compared eight static approaches to the dynamic approach which is the reference. The first is the Infinitest approach, based on Java bytecode. The second is a Moose static approach using the links at classes level. The third is a Moose approach based only on the methods call graphs. The fourth is a Moose approach based on the method call graphs and where the interfaces have been bypassed. The fifth is based in the method call graphs and resolve the accesses problem. The sixth is based in the method call graphs and resolve the anonymous class problem. Finally, the two latest combine the resolution of accesses and anonymous classes on one hand, and add the interfaces on the other hand.

V. Results & Discussion

Table I gives the metrics of the comparison of each static approach to the dynamic approach for the two projects.

About the number of tests to relaunch, it seems that all approaches select few methods. However, a difference remains between the approaches at methods level and the ones at class level. The former selects up to 3% of the tests where the latter

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

[2] http://infinitest.github.io/

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

TABLE I
COMPARISON OF THE STATIC APPROACHES TO THE DYNAMIC ONE TO SELECT THE TESTS AFTER A METHOD CHANGE

	# Tests selected				Precision		Recall	
	P1		P2		P1	P2	P1	P2
Jacoco (dynamic)	45	(0.8%)	2.4	(1%)	-	-	-	-
Infinitest	1231	(23%)	7.6	(5%)	11%	47%	71%	66%
Moose (classes)	401	(8%)	1.9	(1%)	24%	56%	51%	37%
Moose (methods)	19	(0.4%)	0.2	(0.1%)	81%	99%	36%	11%
Moose w/ interface (methods)	107	(2%)	0.6	(0.4%)	52%	90%	85%	26%
Moose w/ accesses (methods)	19	(0.4%)	0.4	(0.2%)	81%	94%	36%	17%
Moose w/ anonymous classes (methods)	19	(0.4%)	0.2	(0.1%)	81%	99%	36%	11%
Moose w/ accesses and anonymous classes (methods)	19	(0.4%)	0.6	(0.4%)	81%	96%	36%	35%
Moose w/ accesses & anonymous classes & interfaces (methods)	150	(3%)	1	(0.6%)	45%	93%	91%	50%

selects up to 23% of the tests. Actually, at class level, not only one test is selected but all the tests contained in the test class.

For the approaches at class level, the recall is better than the others. It is likely that the tests in the same class cover the same method because the tests are grouped by features in the test cases. The approaches with a good recall have a bad precision and vice-versa. It is complex to increase both metrics at the same time.

The resolution of the problems acts differently between the projects. For P1, bypassing the interfaces in the static approach improves the results. The recall is going from 36% to 85%, but the precision has decreased from 81% to 52%. A lot of invocations are made through these interfaces. Bypassing them with the solution brought in Section III is a real improvement. However, there is no improvement in the application P2 by applying this solution. Other mechanisms exercise.

For P2, bypassing accesses or anonymous classes separately does not influence the result. However, resolving both improves the recall from 11% to 50%. This result can still be improved by looking for other limitation of the static approach.

Globally, these experiments suggest that each project has specifics blockers avoiding the test selection. However, combining all solutions brings a better performance of the approach on both projects. The recall for project P1 and P2 are respectively of 91% and 50%.

VI. CONCLUSION

As testing requires more time in the developer schedule, reducing this time is essential. A solution is to select the tests to execute, without executing all the test suite. This way, developers are focused on the resolution of test failures instead of spending time to wait for the test results.

From two projects of a major IT company, we experimented on different approaches to find the one selecting the most relevant tests. We performed both dynamic and static approaches and identified some problems. We solve them to see their impacts on the approaches performances.

The dynamic approach seems ideal to select the tests. However, it requires the tests execution and looses precision when methods are added or modified

The static approaches are faster. At class level, the recall is clearly higher for Infinitest. But the number of selected tests is higher. At methods level, the recall is strongly correlated and to the approach selected to the project. But the number of selected tests is lower. Resolving the problems has an impact on the performance of the selection.

By comparing the approaches on other projects, we expect to retrieve the same problems in a way that the approach resolving all the issues will have the best results.

REFERENCES

Stéphane Ducasse, Michele Lanza, and Sander 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.

Stéphane Ducasse, Nicolas Anquetil, Usman Bhatti, Andre Cavalcante Hora, Jannik Laval, and Tudor 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.lille.inria.fr/archives/reports/Duca11c-Cutter-deliverable22-MSE-FAMIX30.pdf.

Emelie Engström, Mats Skoglund, and Per 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. Ernst. Higher-order hierarchies. In *Proceedings European Conference on Object-Oriented Programming (ECOOP 2003)*, LNCS, pages 303–329, Heidelberg, July 2003. Springer Verlag.

Nicolas Frechette, Linda Badri, and Mourad Badri. Regression test reduction for object-oriented software: A control call graph based technique and associated tool. *ISRN Software Engineering*, 2013, 2013.

JUnit. JUnit. URL http://www.junit.org. http://www.junit.org.

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.

Don Roberts and Ralph E. Johnson. Evolving frameworks: A pattern language for developing object-oriented frameworks. In *Pattern Languages of Program Design 3*. Addison Wesley, 1997.