Source-Code Level Regression Test Selection: the Model-Driven Way

Thibault Béziers La Fosse, Jean-Marie Mottu, Massimo Tisi, Gerson Sunyé

Journal of Object Technology
Published by AITO — Association Internationale pour les Technologies Objets
http://www.jot.fm/

Source-Code Level Regression Test Selection: the Model-Driven Way

Thibault Béziers la Fosse[ac] Jean-Marie Mottu[b] Massimo Tisi[c]
Gerson Sunyé[b]

a. ICAM, Nantes, France
b. Naomod Team, Université de Nantes, LS2N (UMR CNRS 6004)
c. Naomod Team, IMT Atlantique, LS2N (UMR CNRS 6004)

Abstract In order to ensure that existing functionalities have not been impacted by recent program changes, test cases are regularly executed during regression testing (RT) phases. The RT time becomes problematic as the number of test cases is growing. Regression test selection (RTS) aims at running only the test cases that have been impacted by recent changes. RTS reduces the duration of regression testing and hence its cost.

In this paper, we present a model-driven approach for RTS. Execution traces are gathered at runtime, and injected in a static source-code model. We use this resulting model to identify and select all the test cases that have been impacted by changes between two revisions of the program. Our MDE approach allows modularity in the granularity of changes considered. In addition, it offers better reusability than existing RTS techniques: the trace model is persistent and standardised. Furthermore, it enables more interoperability with other model-driven tools, enabling further analysis at different levels of abstraction (e.g. energy consumption).

We evaluate our approach by applying it to four well-known open-source projects. Results show that our MDE proposal can effectively reduce the execution time of RT, by up to 32 % in our case studies. The overhead induced by the model building makes our approach slower than dedicated RTS tools, but the reuse of trace models for further analysis is overtaking this time difference.

Keywords Regression Test Selection, Regression Testing, Model Driven Engineering, Execution Trace

1 Introduction

Regression testing (RT) is an important step in the software development lifecycle. It ensures that code updates do not break the functionalities that have already been suc-

Thibault Béziers la Fosse, Jean-Marie Mottu, Massimo Tisi, Gerson Sunyé. *Source-Code Level Regression Test Selection: the Model-Driven Way*. Licensed under . In Journal of Object Technology, pages 1–20. Available at http://www.jot.fm/

cessfully tested. However, the size of regression test suites tends to grow fast [YH12] when an application is evolving; thus considerably increasing the testing cost.

According to various studies, up to 80 % of testing cost is related to regression testing, and more than 50 % of software maintenance cost is dedicated to testing [ER10]. The process of source code compilation, load, and test execution is commonly called a *build*. When a build is over, a result, successful or not, is returned to the developer. According to this result, the developer will either go to another task or correct the code that has regressed. This is especially true in the context of Continuous Integration (CI), where the regression testing takes place on a separate server [GPD14]. During the time of the build, the developer does not know yet if she needs to correct the code (the wait could be long: e.g., 25:33 min including 06:04+18:11 min testing time for the build `372209560` of the Google Guava project[1]). Thus, reducing this duration would improve the development productivity.

Reducing the cost of regression testing by only running a specific subset of test cases is the purpose of Regression Test Selection (RTS). Most of the large variety of RTS techniques are based on change impact analysis [LR03, LHS+16]. When an application under development is being modified, it might be unnecessary to run all the test cases, especially the ones that are not impacted by changes in the source code. For instance, existing RTS techniques such as Ekstazi [GEM15] are able to reduce the regression test time for the Google Guava project [2] to an average of 45 %. A standard RTS approach usually involves three phases:

- (C) Collection of the dependencies between code and test cases.
- (A) Analysis of the changes to select impacted test cases.
- (E) Execution of the selected test cases.

The benefit of running less test cases during phase (E) could be counterbalanced by the overhead introduced by the phases (C) and (A). Some existing approaches such as FaultTracer [ZKK12] have shown a RTS time that is longer than the execution of all tests. For that reason, several RTS approaches named *offline* calculate the phase (C) beforehand. Hence, when the user starts a build, the dependencies are already available for performing phases (A) and (E) with a positive time gain [Zha18].

The overhead introduced by the phase (A) depends on how the dependencies are computed during the phase (A) and analysed during the phase (C). It mainly depends on the *precision*. A RTS technique is said to be *precise* if all selected test cases are affected by the changed code [CH08] (i.e., no useless test case is ran on unchanged code). Being *precise* is not mandatory and computing the dependencies at a really fine grain induces a significant overhead [GEM15]. The balance between the precision and its induced overhead should be considered when developing and using RTS techniques.

The precision of RTS techniques depends on the granularity when considering *source-code updates*. It could distinguish *file*, *class*, *method*, or *statement* updates. For instance, a class-level update summarises all the modifications inside a class whereas a statement-level update considers modifications of a single statement. The usage of statement-level updates is more precise but induces more overhead than file-level updates. This is the reason why existing RTS approaches [GEM15] have shown faster end-to-end[3] results using file-level changes, despite selecting more tests. Nevertheless,

[1] `https://travis-ci.org/google/guava/jobs/372209560`
[2] `https://github.com/google/guava`
[3] time necessary to compute the set of test cases to run, plus the test execution time.

L. Zhang shows that depending on the file updated and the case-study, it could be worth to consider finer method granularity [Zha18]. Therefore, by designing a model-driven RTS approach, we can provide **modularity** and allow the tester to manage the granularity and optimise the precision.

Independently from the precision, RTS techniques must be *safe*, meaning that they should select every test case that is impacted by changes in the code [RH96].

While most of the related work on RTS is dedicated to ensuring precision or performance of language-specific and RTS dedicated tools, in this paper we focus on providing a solution that leverages Model-Driven Engineering (MDE) for improving RTS in terms of its **reusability** and **interoperability**. In the past decade, MDE has shown that the use of models improves the development in terms of productivity, quality, and reuse [WHR14]. The accuracy of model-driven program analysis techniques has proven this approach to be suitable for tracing down the execution of programs [BCJM10, WP10] and can benefit to RTS. First, the modularity of MDE allows for a configurable approach where the RTS precision can be set as required. Second, regression testing is one step among all the software development steps, MDE aims to prevent each step to be independent by sharing the models. Therefore, we propose a model-driven RTS approach that collects the execution traces in an *impact analysis model*. That model is used all along the RTS and then it can be exchanged, completed, and reused for different purposes (e.g., debugging, performance analysis, optimisation). Such model can be connected to the other models of the MDE environment, allowing for instance to trace regressions from the requirements model or UML models, etc. Finally, regressions often happen because of changes in different artefacts around the code, such as resources, configuration files, or data files. The holistic view fostered by MDE would enable to address all these problems in a uniform way based on the impact analysis model.

The risk of applying MDE in RTS is its potential performance overhead, in particular for building the model [Sel03]. In this paper we show that this cost can be lightened by developing an offline model-driven RTS technique: costly impact analysis model creation is anticipated. Our resulting tool significantly reduces the execution time of regression testing in our case studies, by up to 32 %. While state-of-the-art dedicated RTS tools may have better raw performance, we argue that the reusability of our computed models makes our solution especially valuable in MDE environments.

In this paper, we propose a model-driven approach for a highly *precise* and *safe* RTS, using statement-level impact analysis, in order to select test methods impacted by code changes. An impact analysis model is built offline. When a user requests a build, this model is queried to select all the tests impacted by changes. Finally, this reduced set of test cases is executed, thus accelerating the regression testing time. The approach is based on a customisable model storing the execution traces of the tests. If the fine-grain accuracy of this model may be slowing down the RTS, it is a powerful asset for other software engineering tasks, hence improving its reusability.

To evaluate this approach we aim at answering three research questions:

- **RQ1**: Does our MDE RTS approach allow for a *safe* and *precise* RTS?
- **RQ2**: Is our MDE RTS approach efficient enough to significantly reduce the regression testing time?
- **RQ3**: Does our MDE RTS approach offers a better modularity and reusability than existing approaches?

The remainder of this paper is organised as follows: Section 2 presents the related work, Section 3 describes a motivation example, then Section 4 introduces our approach, followed by an evaluation driven by several experimentations in Section 5. Finally, Section 6 discusses our proposal and Section 7 concludes this paper and presents future work.

2 Related Work

RTS has been extensively studied, leading to multiple systematic studies and surveys [ERS10, YH12, GHK+01, LHS+16]. In the following we describe a few representative examples of state-of-the-art tools. The approaches for RTS can be classified according to their granularity.

Coarse-grained regression test selection. Several popular RTS frameworks are coarse-grained, considering file or class granularity. They result in less overhead than a finer-grained one, and tends to be more scalable for bigger systems. Ekstazi has a 3-phases approach: test selection using file-dependencies [GEM15], test execution, and new dependencies collection, using bytecode instrumentation. Ekstazi is safe for both code changes and file-system changes. It is among the most efficient state-of-the-art RTS solutions, with an average time reduction of 47 %.

Fine-grained regression test selection. Orso et al. present a RTS technique that uses two levels of impact analysis [OSH04]. The first level is coarse-grained, and accounts for relationships between classes, by generating a graph representing inheritance and references. This phase allows for change impact computation at declaration level. The second level generates control-flow graphs at statement level, for each revision of the code. Coverage information is gathered during the execution of tests, and a *coverage matrix* is generated. Such matrix highlights the code statements that are executed by a specific test. Finally, by comparing control-flow graphs of two revisions of the same source code, changed statements are gathered, and the test cases impacted are fetched using the coverage matrix. The empirical evaluation shows that running the fine-grained analysis produces a high overhead.

Zhang et al. present FaultTracer, a RTS tool using Chianti's change impact analysis [ZKK12] for a method-accurate test selection. It generates a set of extended control-flow graphs, and collects the dependencies at the edge-level by running test methods. Tests impacted by changes are then computed using those graphs. Despite showing accurate results, FaultTracer produces a high overhead, and thus computing the impact analysis is sometimes longer than executing all the test cases.

Lingming Zhang proposes HyRTS, an hybrid RTS combining both method granularity and file granularity [Zha18]. Using both the cost-effectiveness of file granularity, and the accuracy of the method-level RTS, HyRTS outperforms Ekstazi and other state-of-the-art RTS techniques.

Our model-driven approach is **modular** and can be set to consider a fine-grained impact analysis as accurate as the state-of-the-art fine-grained RTS techniques or a coarser-grained granularity. This results in the possibility offers to the tester to manage the precision of RTS to get a faster test execution, at the expense of a much slower impact analysis, including the model-management overhead. However, differently from related work, we propose a model of execution trace of the code. This model is persisted, customisable, and available to be **reused** by other model-driven tools for other types of analysis. Finally, we compute this model beforehand, in order to effectively accelerate the regression testing duration when triggered by the user.

```
1  public class CA {
2    void mA(int i){
3      if (i == 0)
4        doSomething1();
5      else
6         doSomething2();
7    }
8     void mB() {
9      doSomething3();
10   }
11   void mC() {
12     doSomething4();
13   }
14 }
15
16 public class TestCA {
17   @Test
18   void test_mA_0() {
19     new CA().mA(0);
20   }
21   @Test
22   void test_mA_1() {
23     new CA().mA(1);
24   }
25   @Test
26   void test_mB() {
27      new CA().mB();
28   }
29   @Test
30   void test_mC() {
31     new CA().mC();
32   }
33 }
```

(a) Revision R_1

```
1  public class CA {
2    void mA(int i) {
3      if (i == 0)
4        doSomething1();
5      else
6        doSomething5();
7    }
8    void mD() {
9      doSomething3();
10   }
11   void mC() {
12     doSomething4();
13   }
14 }
15
16 public class TestCA {
17   @Test
18   void test_mA_0() {
19     new CA().mA(0);
20   }
21   @Test
22   void test_mA_1() {
23     new CA().mA(1);
24   }
25   @Test
26   void test_mB() {
27     new CA().mD();
28   }
29   @Test
30   void test_mC() {
31     new CA().mC();
32   }
33 }
```

(b) Revision R_2

Figure 1 – Two revisions of a program

While we are not aware of any MDE-based tool for RTS of general-purpose languages, there is some recent work on MDE-based management of execution traces, especially for DSLs [BMCB17, HBRV]. We differ in addressing by similar techniques the traces of general-purpose programs. Analysis of execution traces is a very active research area in software engineering (e.g., recent works are [AMP18, GRS17]) and process engineering (e.g., [BvdA12]). We believe that a model-driven management of traces can enhance the **interoperability** of such solutions.

3 Running Example

Figure 1 presents an example that is used to define the context of our model-driven approach for RTS. Figure 1a and Figure 1b define a first and a second revisions of a Java program, respectively. They are verified using the JUnit testing framework.

		Granularity		
		Class	**Method**	**Statement**
Methods	`test_mA_0`	x	x	x
	`test_mA_1`	x	x	
	`test_mB`	x	x	x
	`test_mC`	x		

Table 1 – Selection of test methods depending on the granularity of source-code updates

In the first revision, R_1, the test class `TestCA` verifies that the class `CA` works as expected. Four test cases are implemented: `test_mA_0`, `test_mA_1`, `test_mB`, and `test_mC`. The first two tests cover the two branches of the *if* condition in `mA`. The third one tests the method `mB`, and the last one tests the method `mC`.

The source code modifications of revision R_2 update several lines. The first change occurs at line 6, where the statement inside the *else* branch is updated. The second change occurs at line 8, by renaming method `mB` to `mD`. Finally, the last modification considers the new method name `mD` inside the corresponding test case.

Orso et al. define two different kinds of program changes, *statement-level* changes and *declaration-level* changes [OSH04]. A *statement-level* change can be either a modification, deletion, or addition of executable statements, such as lines 6 and 27 changes. A *declaration-level* change is a modification of a signature, such as line 8 change. It could be method name modifications, method additions or deletions, variable type changes, or any kind of signature changes.

Depending on the granularity of the change impact analysis, different test sets might be selected for a re-run, as illustrates Table 1. Applying RTS with Class granularity selects four test cases, with Method granularity three, and with Statement granularity only two, thus improving the precision. For instance, a Method granularity would consider the entire method `mA()` changed, and would select `test_mA_1()` even if it is not impacted. To improve the precision, Statement granularity analyses the impact of each statement: `test_mA_1()` is not impacted since it doesn't run line 6. Therefore, the test execution time is reduced but the overhead is increased.

Next Section, we present a model-driven approach that is able to select impacted test cases at the most precise Statement granularity. Thereafter, since our approach provides a persistent impact analysis model, it will be up to the tester to choose the granularity to balance overhead/precision w.r.t. its case study.

4 Approach

Our model-driven RTS approach performs the three RTS phases relying on impact analysis model:

- (C) Computation of the impact analysis model.
- (A) Analysis of the impact analysis model to select the impacted tests.
- (E) Execution of the impacted tests.

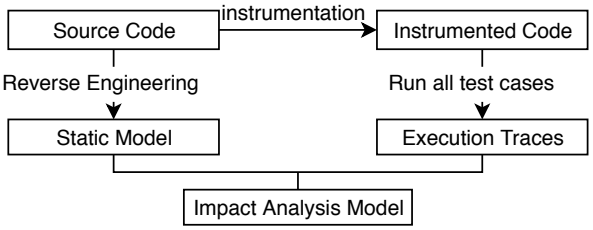


Figure 2 – Steps to compute the impact analysis model

4.1 Computation of the impact analysis model

Several properties of a program (correctness, robustness, safety, dependencies, etc.) can be analysed either statically, i.e., by examining the source code of a program, or dynamically, i.e., by analysing the execution of this program. Combining static and dynamic analysis is often interesting, since the data gathered dynamically can be used to add more precision to the static part. Indeed, considering all the possible execution behaviours of a program is not reasonable for several kinds of analysis [Ern03].

The MoDisco framework is designed to enable static program analysis in MDE [BCJM10]. MoDisco creates a model of the source code, usable for further processing, by *model-driven reverse engineering*. The model is easily accessible to any kind of external modelling tool, providing support for many possible scenarios, such as static measurement, modernisation, understanding. The MoDisco source-code model contains a full model-driven representation of the source code, and a bidirectional transformation between model and code.

RTS requires to add dynamic aspects to the static structure and dependency information already retrieved by MoDisco. Adding dynamic traces turns the static model of MoDisco into an *impact analysis model*, serialising how test cases execute elements of the source code, and helping RTS identifying test cases impacted by source code changes.

Building the impact analysis model is a costly operation. Nonetheless, considering the building of this model as the dependency computation in a standard RTS approach (phase (C)), this model can be computed offline. Hence, when a build is triggered by the user, after a pull request for instance, this model can be immediately analysed to select the test cases impacted by code changes.

The approach we propose to build the impact analysis model consists in a succession of several steps as shown in Figure 2 and as detailed in the next sections. The first step consists in building a static model from the source code, by reverse engineering using MoDisco. The second step involves the instrumentation of the code. In the third step, executing this instrumented code on all the tests produces execution traces. Finally, the fourth step requires parsing those traces, creating references in the model, that show which elements of the source code are executed by a specific test.

4.1.1 Reverse Engineering

To generate the static model of the source code, MoDisco uses a visitor-based approach, navigating through the abstract syntax tree (AST) of the source code. In the case of Java, this model conforms to the MoDisco Java metamodel, is persisted in XML files, and loaded on-demand in the other steps of our process.

Figure 3 shows an excerpt of the model MoDisco would generate using the source code of Figure 1a as an input. The MoDisco model contains most of the information

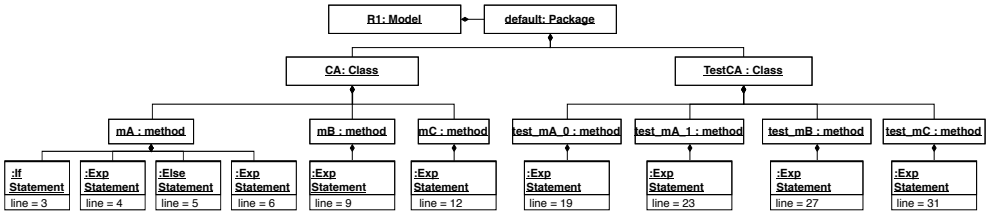


Figure 3 – Excerpt of the MoDisco model with Figure 1a as input

available in the AST, organised in a containment structure, from packages down to expressions. For instance, Statement objects contain line and column numbers, expressions, references to declarations, and so on.

In the next steps we complete this model with dynamic data, gathered when executing an instrumented code.

4.1.2 Instrumentation

The most common approach for tracing the execution of a program is by instrumenting it. In the case of Java test cases this can be done for instance using bytecode instrumentation (BCI) or source code instrumentation (SCI). Both approaches have their advantages and drawbacks. BCI is usually faster, because SCI needs to recompile the instrumented code before being able to use it. However BCI may not be as accurate as SCI [HGB+17].

For instance, the Java Code Coverage Framework JaCoCo[4] uses BCI [SM08] through the ASM library[5]. It iterates over lines of code using the `LineNumberTable` attribute available in Java compiled classes. Probes are added between each line of code, and hence, when a probe is executed, the line of code is considered as covered. Indeed, BCI can be used to know whether a specific instrumented line of code is executed, but not exactly when a statement is executed. Usually, there is only one statement per line of code, but specific cases might happen. For instance, considering a Java ternary operator which contains a if-true expression and an if-false expression at the same line: independently of the boolean condition, a standard BCI would consider both as executed, whereas a SCI is able to distinguish which one is executed.

In our approach we use SCI to precisely track the execution of every element on the MoDisco metamodel. MoDisco builds a model at the source code level, and thus, matching the elements of this source code with the model requires less calculation, and is more precise by using an instrumentation at the same abstraction level. Since this step is done offline, duration of the recompilation required by SCI is not an issue.

The source code is not instrumented in the same way considering test classes or System Under Test (SUT) classes. In the SUT classes, all the statements are instrumented, in order to determine if they have been called by a specific test method. Before each statement, a probe is added. Hence, if this probe is executed, then the statement following the probe is considered as executed too. In the test classes, the source code is instrumented at the method level. A probe is added at the beginning of each test method, monitoring the test case currently executed. This allows our approach to trace down all the SUT statements executed by each test method. We implement the instrumentation by leveraging the Spoon [PMP+16] framework.

[4]https://www.eclemma.org/jacoco/
[5]`http://asm.ow2.org/`

```
test: TestCA.test_mA_1()
    sutMethod: CA.mA(int)
        statement: line 3,col 27 -> line 3,col 38
        statement: line 5,col 55 -> line 5,col 59
        statement: line 6,col 60 -> line 6,col 75

test: TestA.test_mB()
        ...
```

Figure 4 – Execution trace of test_mA_1() after instrumentation

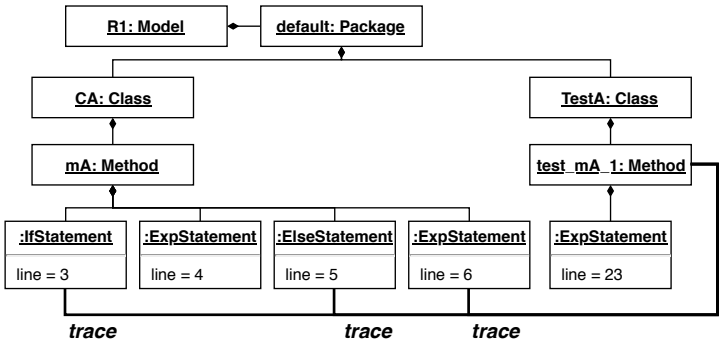


Figure 5 – Excerpt of the impact analysis model

4.1.3 Full test set execution

To produce the initial set of execution traces, all the test cases must be executed once, at the beginning of the process (e.g., when the code is first imported). After the reverse engineering step we query the MoDisco model to gather the list of test cases (using Eclipse OCL queries[6]). Methods having the JUnit 4 `@Test` annotation are considered as test cases, such as methods with the `Test` prefix, in classes extending the `TestCase` class of JUnit 3.

Once all the test cases are gathered, the execution of the injected probes will produce traces, indicating the exact statements, methods, and classes that have been executed for each test case. These execution traces (Figure 2) are serialised in a simple text file. For instance, executing the instrumented test method `test_mA_1()` of Figure 1a, would produce the execution trace of Figure 4.

4.1.4 Injecting traces in the model

Once all the test cases have been executed, the trace file can be analysed, in order to complete the MoDisco model of code structure with dynamic information: The classes and methods are queried in the model using their qualified names, and the statements are queried using their positions in the compilation unit. Once those elements have been found, a bi-directional reference is added between the test case, and the executed program elements. In Figure 5, this reference is the one labeled as *impacts*. For the remainder of this paper, we call this model *Impact Analysis Model*.

Finally, the Impact Analysis Model is persisted in order to be used as soon as a new version of the code needs to be analyzed, to select the test cases impacted.

[6] `https://projects.eclipse.org/projects/modeling.mdt.ocl`

4.2 Analysis of the impact analysis model to select the impacted tests

This second RTS phase selects the test cases to run, based on the impact analysis model in our model-driven approach. This phase is online, starting when the user launches the regression test activity, e.g., after pushing a new version of the code on a VCS.

Once the build starts and before anything else, the new revision is compiled. If the compilation fails, no tests are executed and an error report is given to the developer. If the code compiles successfully, then the RTS can be performed. The two revisions of the source code are compared, in order to locate the changes. To do that, a VCS comparison algorithm is used (we use the one implemented in JGit[7]).

Running such algorithm on two different revisions of the code produces a set of *DiffEntries*. They are not only produced from Java files, but from any kind of files contained in the project, such as configuration files, and compiled sources. When a *DiffEntry* concerns a Java file, the impact of source code changes on test cases is computed using the Impact Analysis Model. Considering non-Java files is part of our future work. The changed statements are gathered in the model and the impacted test cases are selected, using the *trace* reference. For instance, if we focus on the Figure 5, revision R_2 in Figure 1b modified the `ExpStatement` line 6. Using the *trace* reference in the model, `test_mA_1` is considered as impacted, and is selected to be run.

As shown in Figure 5, the *trace* references are precise, linking statements and test methods. Moreover, by querying the model we can derive the impact at coarser levels of granularity. For instance in Figure 5, the `ExpStatement` line 6 is contained inside the method `mA`, itself contained inside the class `CA`. Hence, a coarser-grained analysis (i.e. method or class granularity) can determine that modifying the method `mA` or the class `CA` would impact `test_mA_1`, and so on. Implementing a faster hybrid visitor for computing a coarser-grained impact analysis, such as the one proposed by Lingming Zhang [Zha18] is possible, thanks to the reusability of our model.

DiffEntry considers a block of a few lines of text, and is classified according to the type of update (Modification, Insertion, Deletion). In the following we detail the behaviour of the test selection by type of update.

4.2.1 Modification

First, modifications introduced by developer can be either statement-level changes or declaration-level changes. If the update is a statement-level change, then the impacts can be computed with a statement-granularity. If it is a declaration-level change, then the impacts are computed at a coarser granularity (Method, or Class), depending on the change.

Second, those changes are either in the SUT, or in the test cases. When a test case is modified, it is immediately selected to be re-executed. However, the approach is more specific for SUT modifications: In case of statement-level changes, the modified lines are parsed, in order to get a set of modified statements. Those statements are queried in the model, and using the *trace* reference, all the test cases that executed this specific statement are selected for a new run.

The approach differs in case of declaration-level changes: If a method declaration is modified, by either renaming it, or changing the signature, this method is queried in the model, using its old-version signature. Then, all the test cases that executed this method are gathered, using the *trace* reference, and selected for a re-run.

[7] `https://www.eclipse.org/jgit/`

```
1  abstract class SuperC{
2      void m(){
3          doSomething();
4      }
5  }
6  class C extends SuperC{
7
8
9
10 }
11 class TestC{
12     @Test
13     void test(){
14         new C().m();
15     }
16 }
```

(a) Revision R_1

```
1  abstract class SuperC{
2      void m(){
3          doSomething();
4      }
5  }
6  class C extends SuperC{
7      void m(){
8          doSomething2();
9      }
10 }
11 class TestC{
12     @Test
13     void test(){
14         new C().m();
15     }
16 }
```

(b) Revision R_2

Figure 6 – Adding a new method impacts an existing test case

Furthermore, the same reasoning is applied with declaration-level changes at class level. Thus, when a class signature is modified, the old version of the class is queried in the model, in order to get the test cases impacted, using the *trace* relation.

Selecting the tests impacted by the modification at different granularities (e.g., Package) is also possible, by applying the same approach, thanks to the modularity provided by that model-driven approach.

4.2.2 Insertion

When the change is an insertion, several cases are possible:

First: Any test-related insertion results in the selection of the test case. Those can be either a new test, or an existing test with new lines of code.

Second: When new lines are added in an existing SUT method (i.e. statement-level changes), the modified method is fetched inside the Impact Analysis Model, using an OCL query. And thus, impacted test cases are obtained using the *impacts* reference between this method and the test cases.

Third: When new classes or methods are added. If the method was not present in the previous revision, then it is not present in the model either, and hence, no impacts can be directly computed from it. However, adding such method can have impacts on the existing SUT. Figure 6 describes a situation where the insertion of a new method has impacts on the test cases. This case can be solved by checking by OCL if such method overrides a method of a superclass. For instance, if a method $m()$ is added in class C, this approach would check if $m()$ exists in superclass $SuperC$. If it does, then all the test cases executing `C.m()` would be selected for an execution, using the *trace* reference of `SuperC.m()` in the Impact Analysis Model. This scenario adds a minor overhead, but is necessary to ensure the safety of our RTS approach.

4.2.3 Deletion

In the same way as in the other changes, deletions can be at the *statement-level*, or *declaration-level*, which implies two different behaviours.

First, when a line of code is removed at the statement level (i.e.), inside a method, this method is fetched by querying the Impact Analysis Model. Finally, its impacts on the test cases are obtained using the *trace* reference. Test cases impacted are then selected to be run again.

Second, when an entire method, which does not override any method from a parent class, is removed, then the test cases calling it have probably been modified too. Indeed, removing a non-overridden method from a class, but not the calls to this method would produce compilation issues. The same reasoning works at the class-level. If a deleted parent class is used in the test cases, then those tests have to be updated too.

Otherwise, if the deleted method overrides a method from a superclass, then the old version of this method is fetched from the model, and all the tests executing it are selected, using the *trace* references.

To conclude the Section 4, we notice that we successfully design and implement a model-driven RTS approach that can select impacted test cases based on a model. The current version of our prototype is available on GitHub[8]. Thanks to that model, the granularity of the selection could be easily adapted according to the needs. Existing RTS techniques using variable granularities show excellent results [Zha18]. As shown in Section 4.2, it is possible to get the impacted test cases at several granularities (statement, method, class), depending on the change type (statement-level, declaration-level). Thus, in order to accelerate the RTS, it is possible to limit the impacts to a certain granularity.

5 Evaluation

This section presents an experimental evaluation of our approach. First, we present the environment setup and then the experimental workflow, followed by a presentation of our results.

5.1 Setup

All the experiments are executed on an Intel Core i5-7200U CPU (2.50GHz), 8GB of RAM, running with Ubuntu-16.04, and using Java 1.8.0_51. The four projects presented in Table 2 are used for our evaluation. The column *First commit* corresponds to the first Git revision used, *LOC* approximates the lines of Java code available in each project, and the last two columns list the number of system under test classes and number of test classes, during the first commit. Those projects are available on GitHub[9].

Those four frameworks have been chosen because they answer to several criteria that were mandatory for the current state of our prototype: using the git VCS, Java, the Maven dependency manager, and having test suites running with the JUnit testing

[8] `https://github.com/atlanmod/MDE4RTS`

[9] `https://github.com/jhy/jsoup`, `https://github.com/joel-costigliola/assertj-core` `https://github.com/junit-team/junit4`, `https://github.com/tipsy/javalin`

Framework	First commit	LOC	#Classes Tested	#Test Classes
Javalin	caae71e	2500	13	23
JSoup	7f8010d	25 000	61	31
JUnit4	64155f8	100 000	126	174
AssertJ	cf4d367	250 000	368	1903

Table 2 – Projects used for evaluation

framework. Since all those tools are popular in the development community, restricting the usage of our prototype to those does not significantly hamper its practical usefulness.

5.2 Workflow

We used the following workflow to run the experiments. First, a specific starting revision is determined. This can be a previous tag in a Git history, for instance. This revision is called R_1. Then we define a variable *Step*, representing the number of commits between each computation of the RTS. For instance, with $Step = 2$, the Regression Test Selection would be computed between R_1 and R_3. Instead of applying the RTS at each commit, being able to merge multiple commits at once offers a more accurate representation of a development lifecycle. In fact, developers either push multiple commits at once, or use pull requests, to merge several commits from other branches on the VCS. We then define another variable, Max representing the maximum amount of commits to analyse. For instance, with $Max = 50$, the last revision that could be analysed would be R_{50}.

The experimentation starts with the revision R_1. Since no model can be found, all tests are executed. Then the impact analysis model is built offline, thus computing the dependencies at the statement-level. Once this is done, the revision R_{1+Step} is cloned, and the RTS is applied between R_1 and R_{1+Step}, using the impact analysis model previously generated. Hence, the subset of tests is executed. The impact analysis model is built, from the revision R_{1+Step}, and the next revision $R_{1+2Step}$ is cloned, and this goes on until reaching R_{Max}. Each time, both the subset of tests and all tests are executed with the same technique, in order to compare the results.

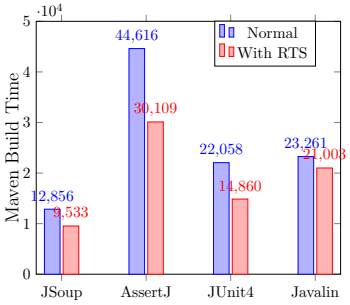
For the experimentations, the Max value is set to 100, and *Step* is set to 5. Those values have been arbitrary chosen and we assume that the quantity of revisions covered is relevant enough for our evaluation.

5.3 Results

Figure 7a presents the results of applying our model-driven approach to RTS to a pool of 100 commits, from the four different projects. The first section of the table presents the quantity of test methods selected by our approach. As the number of tests varies along the commits analysed, average values are used. The second section reports the average compilation and execution times for each project, without RTS. Hence, summing those two values for each project shows the standard build time. The third section presents the average times of the three model-driven RTS activities. The sum of the compilation time with the identification and execution of impacted tests results in the duration of a full build, when using our approach. This model-driven RTS build durations and the standard build durations are compared in the last section of the table, as well as in Figure 7b.

Frameworks	**JSoup**	**AssertJ**	**JUnit4**	**Javalin**
avg Selected test methods	200	888	35	6
avg Test methods	588	4868	998	95
% **Test methods selected**	34.01%	18.24%	3.51%	6.31%
(A) Compilation Time (ms)	4805	9854	10 230	18 368
(B) Executing all tests	8051	34 762	11 828	4893
(1) Computing impact analysis model (ms)	78 307	375 747	82 478	50 555
(2) Identifying impacted tests (ms)	649	4464	1081	331
(3) Executing impacted tests (ms)	4079	15 791	3549	2635
Standard build: (A) + (B)	12 856	44 616	22 058	23 261
Model-driven RTS build: (A) + (2) + (3)	9533	30 109	14 860	21 003
((A) + (2) + (3)) / ((A) + (B))	74.15%	67.5%	67.4%	90.3%

(a) Average evaluation results



(b) Build times with and without RTS

Figure 7 – Evaluation results

Results show that applying model-driven RTS shortens the build time, in all cases, when the impact analysis model is built beforehand. On a bigger project such as AssertJ, the results are more significant, with a RTS build time lasting 67.5 % of a standard build, when only 18.24 % of the tests cases are executed. Comparing model-driven RTS build times with standard build times on a small project like Javalin shows that 6.31 % of the tests are selected, on an average. However, for a project of small size the impact on the build time is less significant, since selecting and executing the tests takes 90.3 % of the execution time of all tests.

As explained earlier, running the tests, either all tests, or a test selection, includes a compilation time. This is performed using Maven and hence several steps of the Maven build lifecycle have to be executed: validation, compilation, and finally testing. Comparing only the actual test execution times would show a bigger gap between RTS and the execution of all tests, but would not reflect real-life builds.

In our experimentation, selecting the tests cases and running them takes always less time than running all the tests cases. As predicted, a loss of time may occur when the number of tests is low since the run of a test subset will not compensate the overhead of the RTS phase (A). However, testers will consider RTS only when their test suite length starts to be problematic and in that case, our approach is beneficial.

6 Discussion

In that section, we discussed the approach and the experiments to answer the three research questions.

6.1 RQ1: Precision

Precision has been discussed previously in Section 4. We can answer the second part of **RQ1** positively since our model-driven approach allows precise RTS. Thanks to the impact analysis model, the precision can be as much finer as Statement granularity. Moreover, the modularity of the approach allows to adapt the precision by analysing differently the impact analysis model.

6.2 RQ1: Safety

To answer the first part of **RQ1**, we consider the *safety* of our model-driven RTS approach. We should prevent an impacted test not to be selected. Therefore, we consider four risks: miss a source code change, analyse a non up-to-date impact analysis model, consider all the software artefacts, and flaky tests.

All the tests are first executed to generate the impact analysis model and their execution is traced at the finest statement granularity. Hence, if any statement is modified in a future revision, then all the tests that previously executed it can be selected thanks to the *trace* references.

The test selection requires an impact analysis model, built offline. If the user launches regression testing too frequently, the model of the last revision may not have been completely built. If that happens, the last entirely computed model is used instead. It would be less *precise* since more tests would be selected (the ones of several iterations), but the *safety* is ensured. If no model is available, then the entire test suite is executed.

The current state of our tool does not compute the impact of external files changes. Nonetheless, such dependencies could be added in the impact analysis model: e.g., Gregoric et al. describe an efficient approach to collect file dependencies, using BCI and monitoring [GEM15]. All standard library methods able to access files are monitored, and dependencies towards those files are added when accessed at runtime. Enhancing our current SCI to monitor external files accesses is part of our future work. It is another advantage of our model-driven RTS to be able to extend the impact analysis model with more information. Considering that risk, we ensure the safety in the current implementation, by detecting that external files modification (using a `diff` algorithm) and warning the developer. She can then run all the regression test cases, in order to ensure that nothing has been broken.

Finally, non-deterministic tests, also known as *Flaky Tests* [LHEM14], are not covered by our approach. These tests can behave differently between several executions, (depending on multiple parameters: asynchronous calls, concurrency, network, test order dependencies, etc.) and therefore cannot be traced deterministically. The safety risk of Flaky Tests is not only affected RTS but all the testing process and it is up to the tester to annotate them for being processed carefully (e.g., with several execution).

6.3 RQ2: Performance

As presented in the Introduction with **RQ2**, our model-driven approach has to be efficient enough to reduce the regression testing time. Poorly performed RTS steps might produce an overhead so important that selecting and executing a reduced set of tests would take longer than running the entire test suite.

First, our RTS approach is offline. Therefore, we do not take into account the model generation time (RTS phase (C)). This phase is costly when generating an impact analysis model, but it is already performed before the tester launches the regression testing. Second, the test cases selection (RTS phase (A)) is critical, and must be computed in a fast way to prevent a counter-productive overhead. Two expensive activities are processed during the test cases selection:

1. Comparing two revisions of a project, more specifically at a fine granularity (Statement-level).

2. Querying the model for specific elements.

Being efficient on those two points is a major concern. Performance when manipulating models are known issues in model-driven engineering. Experiments presented Section 5 shows that we positively answer **RQ2** since selecting the tests cases and running them takes always less time than running all the tests cases. In addition, our model-driven approach allows to configure the granularity and then to balance the overhead depending on the case study (e.g. a case study with few test cases would benefit from a coarser-granularity, while a case study with slow test cases would benefit from a fine-granularity to reject slow and useless test cases).

Our performance (gain of up to 32 %) are less important than the ones of dedicated tools such as the ones of [Zha18][GEM15] (they can gain more than 40 %). However, they will be overtaken by our model-driven approach for further analysis that can reuse the trace models, as discussed next section.

6.4 RQ3: Complementary use of the Model

In a MDE environment, our impact analysis model becomes a part of the holistic modelling of all software aspects. Thus, it can be reused to perform heterogeneous analysis on the project.

For example, tracing down the execution of test cases inside a model would provide a better understanding of a test suite, useful while reverse-engineering old code bases. This model could also be used to get an accurate overview of the testing code coverage, and thus significantly improve the quality of test suites [MvDZB08].

Concretely, in our tool, we use the Structured Metrics Meta-model[10] (SMM) for gathering measurements performed during the test execution [BlFMTS18]. SMM enables the modelling of several metrics: lines of code, test execution durations, method complexities, energy consumptions, and so on. By weaving the SMM model along with the model produced during RTS, we are able to know the energy saved by not running the test cases that have not been impacted. Figure 8 shows an example of a SMM usage along with the model produced by our RTS approach. The `Measurement` elements contain energy consumptions, and refer to the `Methods` available in the model. `ObservedMeasure` contains the timestamp of the measures. Finally `Observation` and `SmmModel` are root elements in the model.

[10] https://www.omg.org/spec/SMM/

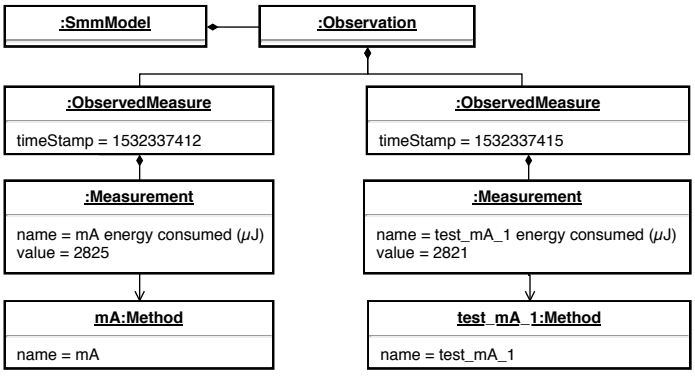


Figure 8 – Excerpt of a SMM Energy consumption model

We successfully answer the **RQ3** since the impact analysis model can be completed and analysed for other software engineering analysis, such as energy consumption. On the contrary of dedicated tools that are not compatible and require each one to run and to analyse the program, we collect and serialise all the information in one impact analysis model. Created once, we save time, counter-balancing the overhead induced by model manipulations.

6.5 Threat to validity

Several threats to the validity of our experimentation have to be considered:
Commits analysed: First of all, the amount of selected tests highly depends of the changes made by developers. For instance, the modification added between commits `fce43e6` and `4fe13cb` on the project AssertJ impacted 3312 existing test methods. Therefore, the results could be radically different using other commits. Nonetheless, by applying our RTS approach on a set of 100 successive commits, we assume that it covers usual development scenarios, such as adding of new methods, or modifying of the core methods of the project.

Single machine: All those results have been computed on a single computer dedicated to this task. Even if a particular attention has been taken to only dedicate the machine to the experimentations, background operations performed on the system may have impacted the execution times reported.

State of the prototype: Finally, our research prototype's implementation may contain undetected bugs, and thus the results produced may be subject to variations with the upcoming improvements and corrections. Even if this prototype is well tested and its source code has been thoroughly examined (and is open to inspection), not all the results obtained during the experimentation can be manually verified. Especially the safety of the approach, while theoretically valid, has been manually checked only for the small projects. Finally, this prototype is still under development, and so far its performances have been improving at each revision.

Scalability: Models are well known for suffering of scalability issues [PCM11]. In fact, the standard EMF implementation persists models using XML data structure. Those files cannot be partially loaded, and if the model is too large, it might not fit in memory anymore. When the source code is massive (thousands of classes), the model generated by MoDisco can be huge (GBs), which may produce scalability issues. This limitation can be resolved by using scalable persistence layers for models, such as NeoEMF[11], or CDO[12].

7 Conclusion and Future Work

In this paper, we presented an approach that applies MDE techniques to RTS. We build an impact analysis model, starting from a source-code structural model and enhanced with dynamic information, gathered during test case execution. On top of this model we implement a fine-grained impact analysis, allowing for effective RTS. The benefits of the impact analysis model are twofold: it allows the separation of the impact analysis from the test selection and opens different perspectives for future work.

[11] `https://www.neoemf.com/`
[12] `https://www.eclipse.org/cdo/`

Computing the long-running impact analysis offline allows us to rapidly apply the RTS when the user needs it, minimising the waiting time, and hence improving the productivity. Indeed, experiments on four different open-source projects showed that our approach significantly shortens the regression test execution time. Moreover, the proposed impact analysis model can serve as a basis for different software engineering activities: fault localisation, precise code coverage, energy consumption analysis, etc.

While the current version of our prototype provides positive performance results, several points can still be improved. Incrementality should be considered to reuse and complete the model during the RTS phase. Indeed, after each build, our prototype creates the whole model from scratch, using MoDisco. Using *diffs* to update an existing model according to the new source code version would shorten the impact analysis model building time. We also intend to connect our prototype with a persistence layer, to support larger projects. More experimentations will be performed, especially on projects larger than `AssertJ`, and accuracy will be compared to state-of-the-art RTS tools, such as HyRTS, Ekstazi and FaultTracer.

Finally we will also focus on completing the implementation of our research prototype, in order to have more precise results. The impact analysis should be improved, in order to also analyse external files other than Java source files. Indeed, modifying a configuration file may impact the test cases execution. This kind of problem threatens the safety of traditional RTS tools, and a MDE-based holistic approach to RTS could provide a natural solution.

References

[AMP18] Saba Alimadadi, Ali Mesbah, and Karthik Pattabiraman. Inferring hierarchical motifs from execution traces. In *International Conference on Software Engineering (ICSE), 2018 (to appear)*, 2018.

[BCJM10] Hugo Bruneliere, Jordi Cabot, Frédéric Jouault, and Frédéric Madiot. Modisco: a generic and extensible framework for model driven reverse engineering. In *Proceedings of the IEEE/ACM international conference on Automated software engineering*, pages 173–174. ACM, 2010.

[BlFMTS18] Thibault Béziers la Fosse, Jean-Marie Mottu, Massimo Tisi, and Gerson Sunyé. Characterizing a Source Code Model with Energy Measurements. In *Workshop on Measurement and Metrics for Green and Sustainable Software Systems (MeGSuS)*, Oulu, Finland, October 2018. URL: `https://hal.inria.fr/hal-01952724`.

[BMCB17] Erwan Bousse, Tanja Mayerhofer, Benoit Combemale, and Benoit Baudry. Advanced and efficient execution trace management for executable domain-specific modeling languages. *Software & Systems Modeling*, pages 1–37, 2017.

[BvdA12] RP Jagadeesh Chandra Bose and Wil MP van der Aalst. Process diagnostics using trace alignment: opportunities, issues, and challenges. *Information Systems*, 37(2):117–141, 2012.

[CH08] Pavan Kumar Chittimalli and Mary Jean Harrold. Regression test selection on system requirements. In *Proceedings of the 1st India software engineering conference*, pages 87–96. ACM, 2008.

[ER10] Emelie Engström and Per Runeson. A qualitative survey of regression testing practices. *Product-Focused Software Process Improvement*, pages 3–16, 2010.

[Ern03] Michael D Ernst. Static and dynamic analysis: Synergy and duality. In *WODA 2003: ICSE Workshop on Dynamic Analysis*, pages 24–27, 2003.

[ERS10] Emelie Engström, Per Runeson, and Mats Skoglund. A systematic review on regression test selection techniques. *Information and Software Technology*, 52(1):14–30, 2010.

[GEM15] Milos Gligoric, Lamyaa Eloussi, and Darko Marinov. Practical regression test selection with dynamic file dependencies. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, pages 211–222. ACM, 2015.

[GHK+01] Todd L Graves, Mary Jean Harrold, Jung-Min Kim, Adam Porter, and Gregg Rothermel. An empirical study of regression test selection techniques. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 2001.

[GPD14] Georgios Gousios, Martin Pinzger, and Arie van Deursen. An exploratory study of the pull-based software development model. In *Proceedings of the 36th International Conference on Software Engineering*, pages 345–355. ACM, 2014.

[GRS17] Maayan Goldstein, Danny Raz, and Itai Segall. Experience report: Log-based behavioral differencing. In *Software Reliability Engineering (ISSRE), 2017 IEEE 28th International Symposium on*, pages 282–293. IEEE, 2017.

[HBRV] Abel Hegedus, Gábor Bergmann, István Ráth, and Dániel Varró. Back-annotation of simulation traces with change-driven model transformations. In *Software Engineering and Formal Methods (SEFM), 2010*.

[HGB+17] Ferenc Horváth, Tamás Gergely, Árpád Beszédes, Dávid Tengeri, Gergő Balogh, and Tibor Gyimóthy. Code coverage differences of java bytecode and source code instrumentation tools. *Software Quality Journal*, pages 1–45, 2017.

[LHEM14] Qingzhou Luo, Farah Hariri, Lamyaa Eloussi, and Darko Marinov. An empirical analysis of flaky tests. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 643–653. ACM, 2014.

[LHS+16] Owolabi Legunsen, Farah Hariri, August Shi, Yafeng Lu, Lingming Zhang, and Darko Marinov. An extensive study of static regression test selection in modern software evolution. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 583–594. ACM, 2016.

[LR03] James Law and Gregg Rothermel. Whole program path-based dynamic impact analysis. In *Proceedings of the 25th International Conference on Software Engineering*, pages 308–318. IEEE Computer Society, 2003.

[MvDZB08] Leon Moonen, Arie van Deursen, Andy Zaidman, and Magiel Bruntink. On the interplay between software testing and evolution and its effect on program comprehension. In *Software evolution*, pages 173–202. Springer, 2008.

[OSH04] Alessandro Orso, Nanjuan Shi, and Mary Jean Harrold. Scaling regression testing to large software systems. In *ACM SIGSOFT Software Engineering Notes*, volume 29, pages 241–251. ACM, 2004.

[PCM11] Javier Espinazo Pagán, Jesúss Sánchez Cuadrado, and Jesús Garcıa Molina. Morsa: A scalable approach for persisting and accessing large models. *Model Driven Engineering Languages and Systems*, 6981:77–92, 2011.

[PMP+16] Renaud Pawlak, Martin Monperrus, Nicolas Petitprez, Carlos Noguera, and Lionel Seinturier. Spoon: A library for implementing analyses and transformations of java source code. *Software: Practice and Experience*, 46(9):1155–1179, 2016.

[RH96] Gregg Rothermel and Mary Jean Harrold. Analyzing regression test selection techniques. *IEEE Transactions on software engineering*, 22(8):529–551, 1996.

[Sel03] Bran Selic. The pragmatics of model-driven development. *IEEE software*, 20(5):19–25, 2003.

[SM08] Kaitlin Duck Sherwood and Gail C Murphy. Reducing code navigation effort with differential code coverage. *Department of Computer Science, University of British Columbia, Tech. Rep.*, 2008.

[WHR14] Jon Whittle, John Hutchinson, and Mark Rouncefield. The state of practice in model-driven engineering. *IEEE Software*, 31(3):79–85, 2014.

[WP10] Stefan Winkler and Jens Pilgrim. A survey of traceability in requirements engineering and model-driven development. *Software and Systems Modeling (SoSyM)*, 9(4):529–565, 2010.

[YH12] Shin Yoo and Mark Harman. Regression testing minimization, selection and prioritization: a survey. *Software Testing, Verification and Reliability*, 22(2):67–120, 2012.

[Zha18] Lingming Zhang. Hybrid regression test selection. In *Proceedings of the 40th International Conference on Software Engineering*, pages 199–209. ACM, 2018.

[ZKK12] Lingming Zhang, Miryung Kim, and Sarfraz Khurshid. Faulttracer: a change impact and regression fault analysis tool for evolving java programs. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*. ACM, 2012.