Empirical Software Engineering manuscript No.
(will be inserted by the editor)

Change-Based Test Selection: An Empirical Evaluation

Quinten David Soetens · Serge Demeyer · Andy Zaidman · Javier Pérez

Received: date / Accepted: date

Abstract Regression test selection (*i.e.*, selecting a subset of a given regression test suite) is a problem that has been studied intensely over the last decade. However, with the increasing popularity of developer tests as the driver of the test process, more fine-grained solutions that work well within the context of the Integrated Development Environment (IDE) are in order. Consequently, we created two variants of a test selection heuristic which exploit fine-grained changes recorded during actual development inside the IDE. One variant only considers static binding of method invocations while the other variant takes dynamic binding into account. This paper investigates the tradeoffs between these two variants in terms of the reduction (*i.e.*, How many tests could we omit from the test suite, and how much did we gain in runtime execution?) as well as the fault detection ability of the reduced test suite (*i.e.*, Were tests omitted erroneously?). We used our approach on three distinct cases, two open source cases —Cruisecontrol and PMD— and one industrial case — Historia. Our results show that only considering static binding reduces the test suite significantly but occasionally omits a relevant test; considering dynamic binding rarely misses a test yet often boils down to running the complete test suite. Nevertheless, our analysis provides indications on when a given variant is more appropriate.

Keywords Fine Grained Changes · Test Selection · Developer Tests · Dynamic Binding

Q. D. Soetens
University of Antwerp — Middelheimlaan 1 — 2020 Antwerp, Belgium — +32 3 265 38 71
E-mail: quinten.soetens@uantwerpen.be

S. Demeyer
University of Antwerp — Antwerp, Belgium
E-mail: serge.demeyer@uantwerpen.be

A. Zaidman
Delft University of Technology — Delft, The Netherlands
E-mail: A.E.Zaidman@tudelft.nl

J. Pérez
University of Antwerp — Antwerp, Belgium
E-mail: javier.perez@uantwerpen.be

1 Introduction

With the advent of agile processes and their emphasis on continuous integration, developer testing has been gaining popularity (Garousi and Varma, 2010). Developer tests are codified unit or integration tests written by developers, aiming to quickly verify whether changes to the system break previous functionality (Meszaros, 2006). From a software engineering point of view, this is certainly beneficial as it is likely to lead to higher quality software. Yet, the very popularity of developer testing is also what could signal its downfall. In particular, if we look at recent observations, we see that in 2005 Microsoft reported that 79% of developers use unit tests (Venolia et al, 2005). Others reported that the code for developer tests is sometimes larger than the application code under test (Daniel et al, 2009; Zaidman et al, 2011; Tillmann and Schulte, 2006). Knowing about the popularity of developer testing and the sheer size of resulting test suites, we are worried about the effectiveness of developer testing. When we look at the best practice of testing both early *and* often (McGregor, 2007), we are particularly concerned with the observation of Runeson, who reported that some unit test suites take hours to run (Runeson, 2006). And when developer tests take too long to run, developers are less inclined to run them after each and every change, hence offload the test execution to a "retest-all" on the continuous integration server (Beller et al, 2015a,b). This impedes the rapid feedback cycles required by continuous integration (Dösinger et al, 2012).

Consequently, we set out to reexamine so-called *test selection techniques*; *i.e.*, techniques that "*determine which test-cases need to be re-executed* [...] *in order to verify the behaviour of modified software*" (Engström et al, 2010). While test selection was studied intensely in the context of regression testing, it has seldom been studied in the area of developer tests (Yoo and Harman, 2012; Yoo et al, 2011). The presence of modern development tools in particular presents a nice opportunity since it is possible to obtain fine-grained access to the modifications that have been made. Integrated development environments (such as Eclipse), for instance, allow to record all changes that have been made in the editor. Similarly, version control systems (such as Git) keep track of the intermediate stages which allows to recover the changes.

In previous work, we designed a heuristic which, given a set of fine-grained (*i.e.*, method-level) changes, selected a subset of unit tests that were affected by iterating over the static dependency graph (Soetens et al, 2013). The heuristic proved to be very effective as long as the object-oriented design did not rely on polymorphism. Consequently, we designed a variant of the heuristic which took dynamic binding into account (Parsai et al, 2014). To compare the two variants of the heuristic —one with and one without dynamic binding— we follow the Goal-Question-Metric paradigm (Basili et al, 1994). The overall *goal* of our investigation is:

> ***GOAL*** – *To investigate the tradeoffs between two variants of a test selection heuristic, determining the minimal set of tests a developer needs to re-execute in order to verify whether a software system still behaves as expected.*

The *objects of study* are two variants of a test selection heuristic, one based on the static dependency graph only (hence will select only a few test cases, however is likely to miss some) and another which takes dynamic binding into account (hence will rarely miss at the expense of selecting several irrelevant test cases). The *purpose* is an in-depth investigation of these tradeoffs: which variant is most appropriate for which situation. The *focus* is the impact of dynamic binding on the performance of the test selection heuristics. The *viewpoint* is that of a developer who wants to minimise the number of test cases to re-execute, while maximising the likelihood of exposing regression faults. The *context* is one of software development teams adopting agile practices. In particular, fully automated unit tests combined with continuous integration. The continuous integration is needed to serve as a safety net, since relevant tests that are missed by the test selection are still executed in the nightly build.

Given that test selection has been an active field of research in the past, we address our *goal* via previously established criteria (*e.g.*, Mansour et al (2001); Graves et al (2001)) and derive the following *research questions*.

RQ1 – *What is the fault detection ability of the reduced test suites?* Given that we reduce the test set, we are interested in knowing the number of tests in the subset that were included unnecessarily, and the ones that were omitted erroneously. This question is addressed by two *metrics*:
- *Mutation Coverage*: A mutant is a transformation of the original base code that introduces a fault; a test which reveals the corresponding fault is said to "kill" the mutant. If the number of mutants killed by a subset is comparable to the "retest all" approach, it increases the confidence in the fault detection ability of the reduced test suite.
- *Failing Test Coverage*: Both variants of the test-selection heuristic are designed to prevent failing tests from reaching the continuous integration server; these tests should fail when they are executed in, for example, the IDE. Hence, as validation we propose to retroactively investigate the entire history of a software system in order to identify in each revision which tests failed on the continuous integration server. We subsequently evaluate whether our test selection heuristic generates a subset of tests that include the failing test cases.

RQ2 – *How much can we reduce the unit test suite in the face of a particular change operation?* Our reasoning is that when a "retest-all" takes too long it is off-loaded to the continuous integration server (Beller et al, 2015b,a). However, when the reduced test suite is small enough (and thus runs quickly enough), we expect developers will execute them before committing to the repository. This question is addressed by two *metrics*:
- *Size reduction*: How many tests could we omit from the test suite? This is a relative number compared to the total number of tests in the test suite.
- *Time reduction*: How much did we gain in run-time execution? We also take into account the overhead imposed by running the heuristic.

To answer these research questions, we implemented a proof of concept prototype named ChEOPSJ, an Eclipse plugin which extracts the changes from a version control system or captures them in the main editor while the developer is programming (Soetens and Demeyer, 2012). We applied the prototype on three distinct cases: two open source cases —Cruisecontrol and PMD— and one industrial case —Historia.

This paper is an extension of the work reported at CSMR 2013 (Soetens et al, 2013) and RefTest 2014 (Parsai et al, 2014). In particular, the following extensions have been made:
(i) A review and replication of the original case study on both of the original open source cases —Cruisecontrol and PMD.
(ii) A new *industrial* case —Historia— where continuous integration was used, that serves as a representative case.
(iii) An extra measurement, where we compare the run-time execution of the full test-suite and the reduced test suite, to see how much time is saved.
(iv) An additional analysis to assess whether the test selection heuristic would have prevented failing tests on the continuous integration server.
The remainder of this paper is structured as follows. First, we describe both the change model and the test selection heuristic, both with and without taking dynamic binding into account. We also detail how all this is implemented in ChEOPSJ (Section 2). The next sections are modelled following the GQM paradigm. The *Metrics* are detailed in the case study design (Section 3), in which we also motivate the cases under investigation. We proceed by analysing and discussing the results for each of the *Questions* in Sections 4 (**RQ1**) and 5 (**RQ2**). In Section 6 we achieve our *Goal* by discussing the tradeoff between the costs of selecting and executing test cases versus the need to achieve sufficient detection ability. We end the paper with an overview of the threats to validity (Section 7), a summary of related work (Section 8), and wrap up with the conclusions and future directions (Section 9).

2 Change Reification

To obtain more accurate information about the evolution of a program, changes should be considered as first-class entities, *i.e.*, entities that can be referenced, queried and passed along in a program (Robbes and Lanza, 2007). First-class change entities, modeled as objects in Robbes and Lanza's approach, represent the behaviour of the different kinds of changes required for a program (for example, to add, remove, or modify classes) (Ebraert et al, 2007).

In recent years several researchers have built upon that idea and have created tools that analyse change objects. The approaches by Robbes *et al.* in the Spyware tool (Robbes and Lanza, 2008) and later by Hattori *et al.* in the Syde tool (Hattori and Lanza, 2010) model changes as operations on the Abstract Syntax Tree (AST). These changes act upon program entities, such as packages, classes, methods and attributes. The approach made by Ebraert *et al.* also includes dependencies between changes (Ebraert et al, 2007). We chose to expand upon the approach by Ebreart *et al.* because we are particularly interested in analysing dependencies in order to determine which tests are relevant for a set of applied changes. Where Ebraert *et al.* made creative use of Smalltalk's internal change list, we have implemented a Java version of their change model in Eclipse (Soetens and Demeyer, 2012).

The change model is shown in Figure 1. We define a *Change* as an object representing an action that changes a software system. In our model we define three kinds of *Atomic Changes*: `Add`, `Modify` and `Remove`. These changes act upon a *Subject* and respectively represent three actions: adding a subject, modifying a subject or removing a subject. For the subjects we can use any model of the software system. We chose to use the FAMIX model as defined in (Demeyer et al, 1999). This is a model to which most class-based object oriented programming languages adhere to; as such, our approach can be translated to any object-oriented language. FAMIX contains entities representing packages, classes, methods and attributes; as well as more fine grained entities such as method invocations, variable accesses and inheritance relationships.

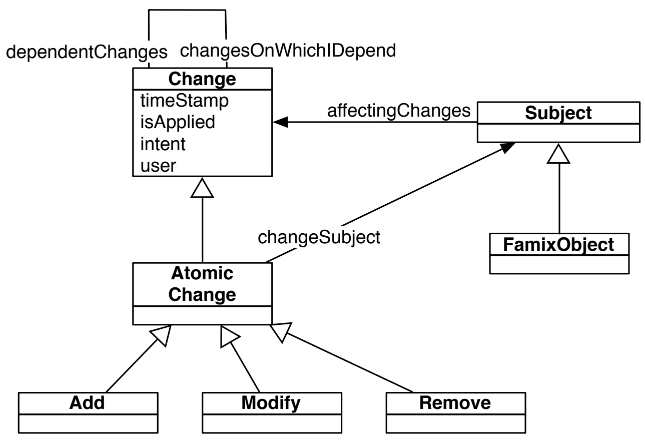

Fig. 1: Model of the changes.

In our model we also define the change dependencies. For this we rely again upon the FAMIX model, which imposes a number of invariants to which each FAMIX model must adhere. For instance, there is an invariant that states that each method needs to be added to a class. This means that there is a precondition for the change that adds a method m to a class c: there should exist a change that adds the class c and there is no change that removes the class c. Using these preconditions we can define dependencies between changes.

In general we can say that a change object c_1 depends on another change object c_2 if the application of c_1 without c_2 would violate the system invariants.

2.1 Test Selection Heuristic

We aim to use this dependency information for test selection purposes. In particular, in our test selection heuristic, we make use of two particular kinds of dependencies:

– A change c_1 is said to *Hierarchically depend* on a change c_2 if the subject of c_1 is in a `belongsTo` relation with the subject of c_2. This `belongsTo` relation is defined by the FAMIX model (*e.g.*, a method belongs to a class, a class belongs to a package, *etc.*).
– A change c_1 is said to *Invocationally depend* on a change c_2 if c_2 is the change that adds the behavioural entity that is invoked by the subject of c_1. For

instance, consider a change c_1 that adds an invocation to a method, which was added by change c_2, then we say that c_1 invocationally depends on c_2.

The tests written in an XUnit framework are also written in the same programming language as the base code. As such, both the changes that act upon the test code, as well as the changes that act upon the program code, adhere to the same meta-model. Therefore there exist dependencies between the changes of the test code and the changes of the program code. These dependencies are used to retrieve the tests that are relevant for a particular change (or set of changes).

To calculate a reduced test suit we execute Algorithm 1. In essence this is a variation of the so-called "Retest within Firewall" method (Binder, 1999), using control flow dependencies on the methods that have been changed to deduce the affected tests.

Algorithm 1: selectRelevantTests

```
Input: A $ChangeModel$, A set $SelectedChanges$
Output: A Map that maps each selected change to a set of relevant tests.
foreach $c$ in $SelectedChanges$ do
    $relevantTests$ = new empty list;
    $calledMethod$ = findMethodAddition(hierarchicalDependencies(c));
    $invocations$ = invocationalDependees(calledMethod);
    foreach $i$ in $invocations$ do
        invokedBy = findMethodAddition(hierarchicalDependencies(i));
        foreach $m$ in $invokedBy$ do
            if $m$ is a test then
                add $m$ to $relevantTests$;
            else
                if $m$ was not previously analysed then
                    $tests$ = selectRelevantTests($m$);
                    add $tests$ to $relevantTests$;
    map $c$ to $relevantTests$;
```

In this algorithm, we iterate all selected changes and map each change to their set of relevant tests. We start by finding the change that adds the method in which the change was performed. We can find this change, by following the chain of hierarchical dependencies and stop at the change that adds a method. In Algorithm 1 this is presented abstractly by a call to the procedure `findMethodAdditions`. After this call `calledMethod` will be the change that adds the method in which the change `c` took place. Next we need to find all changes that invocationally depend on this methodAddition. These are the additions of invocations to the method in which the selected change occurred. For each of these changes we again look for the change that adds the method in which these invocations were added. And thus we find the set of all changes adding a method that invokes the method containing our selected change. We then iterate these method additions and check whether these changes added a test method (*i.e.*, for jUnit 3.X, we look for methods that have an identifier starting with "test"; for jUnit 4.X we tag the methods in our change model whenever a method has the annotation "@Test"). If this was the case we consider this test method as a relevant test for the originally selected change. If on the other hand the added method was not a test method, then we need to find the relevant tests of this method and that set of tests needs to be added to the set of relevant tests for the selected change.

2.2 Dynamic Binding During Test Selection

In our original approach (Soetens et al, 2013), the change model assumed that invocations were a one to one relationship between the caller and the callee. As such, the addition of an invocation was hierarchically dependent on the addition of the caller method and invocationally dependent on the addition of the callee method. We could statically determine the latter based on the type of the variable on which the method was invoked. However with dynamic binding this is not necessarily the case, as a method invocation might invoke any of a number of possible methods.

Take for instance, the code in Figure 2. This is a simplified presentation of one of the tests in the PMD case, where we have a class `Renderer` that declares a method `renderFileReport` and a subclass `HTMLRenderer` which overrides that method. The test code also makes use of dynamic binding, by using an `AbstractRendererTest`, that declares three methods: the abstract methods `getRenderer` and `getExpected` and the actual test method `testRenderer`. The subclass `HTMLRendererTest` then provides overridden versions (*i.e.*, actual implementations) of the methods `getRenderer` and `getExpected`. The actual test `testRenderer` invokes the method `renderFileReport` on a variable `renderer` of type `Renderer`. Therefore the heuristic would state that this test is relevant for all changes in the method `Renderer.renderFileReport`. However invoking `getRenderer` will at runtime actually generate an instance of type `HTMLRenderer` so this test is in fact also relevant for the method `HTMLRendererTest.renderFIleReport`, which is a link that the first variant of our test selection heuristic missed. This is due to the fact that our change model did not account for dynamic binding. Indeed, when we look at the change model for this code in Figure 3, we see that the three invocations in the `testRenderer` method are only linked to the methods declared in superclasses.

In Figures 3 and 4 hierarchical dependencies are shown with a full arrow, invocational dependencies are indicated by a dashed arrow. Note that for readability the links between changes and the FAMIX entities they act on are omitted and instead this relationship is indicated by proximity.

In order to deal with dynamic binding, we changed our change model to include the following heuristic: for each addition of a method invocation using method identifier i, we take it to be invocationally dependent on all additions of methods declared using i as method identifier. This simple heuristic is likely to lead to overestimations, which is why we did an analysis of the method identifiers used in our three cases. The results show that on average 70% of the method identifiers are unique within a project, *i.e.*, they are used for a single method declaration. In 30% of the cases a method name is reused within the same class, *i.e.*, method overloading, and in 4.5% of the cases a method name is overriding a method in a superclass. There is also a small percentage of methods that carry the same name, but are not part of the same inheritance hierarchy. It is actually this last category as well as the overloaded methods that possibly make our test reduction approach less accurate, in the sense that this combined set of methods can still be reduced.

Therefore to take dynamic binding into account we redefine *Invocational dependency* as follows:

```
public abstract class Renderer {
  public void renderFileReport(Report report){...};
}

public class HTMLRenderer extends Renderer {
  public void renderFileReport(Report report){...}
}
```

```
public abstract class AbstractRendererTest{
  public abstract Renderer getRenderer();
  public abstract String getExpected();

  @Test
  public void testRenderer() throws Throwable {
    Report report = new Report();
    Renderer renderer = getRenderer();
    actual = renderer.renderFileReport(report);
    assertEquals(getExpected(), actual);
  }
}

public class HTMLRendererTest extends AbstractRendererTest {
  public Renderer getRenderer() {
    return new HTMLRenderer();
  }
  public String getExpected() {...}
}
```

Fig. 2: Example of code with dynamic binding (adapted from PMD code).

– A change c_1 is said to *invocationally depend* on a change c_2 if c_2 is the change that adds a behavioural entity with identifier i and the identifier i is used as the invocation in the subject of c_1.

This would change the model of the changes in Figure 3 to the model represented in Figure 4. In the new model there are added dependencies from the additions of the invocations (indicated in blue arrows). Most importantly there is an added dependency from the invocation addition in the test method to the addition of the method `HTMLRenderer.renderFileReport`. So now our same test selection heuristic will say that the test `AbstractRendererTest.testRenderer` is relevant for changes in both the methods `Renderer.renderFileReport` and `HTML-Renderer.renderFileReport`.

2.3 Tool Support: ChEOPSJ

In a previous paper we presented our tool, ChEOPSJ[1] (Change and Evolution Oriented Programming Support for Java), which is implemented as a series of Eclipse plugins (Soetens and Demeyer, 2012). The general design is depicted in Figure 5.

At the centre of the tool we have a plugin that contains and maintains the change *Model*. To create instances of the change model we have two plugins: the *Logger* and *Distiller*. These two plugins are responsible for populating the change

[1] http://win.ua.ac.be/~qsoeten/other/cheopsj/

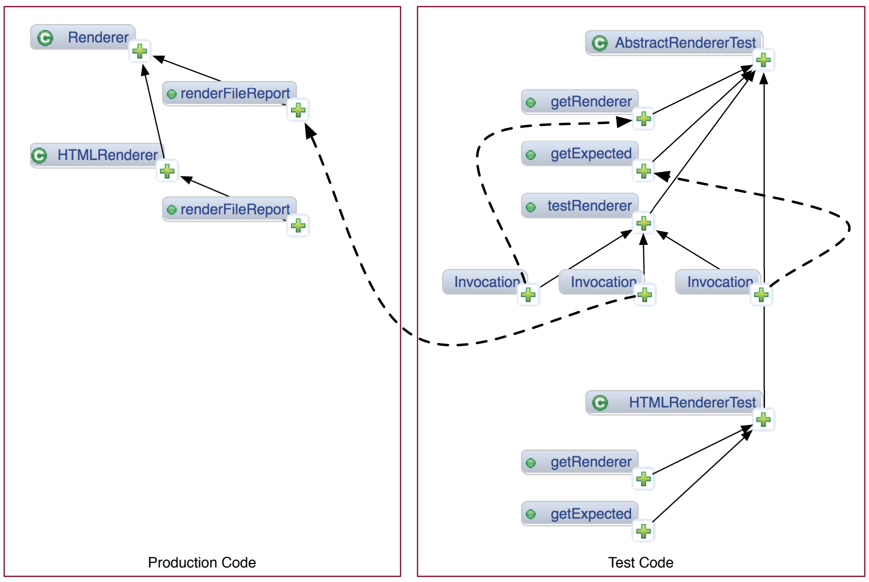

Fig. 3: Change Model for code in Figure 2 without dynamic binding.

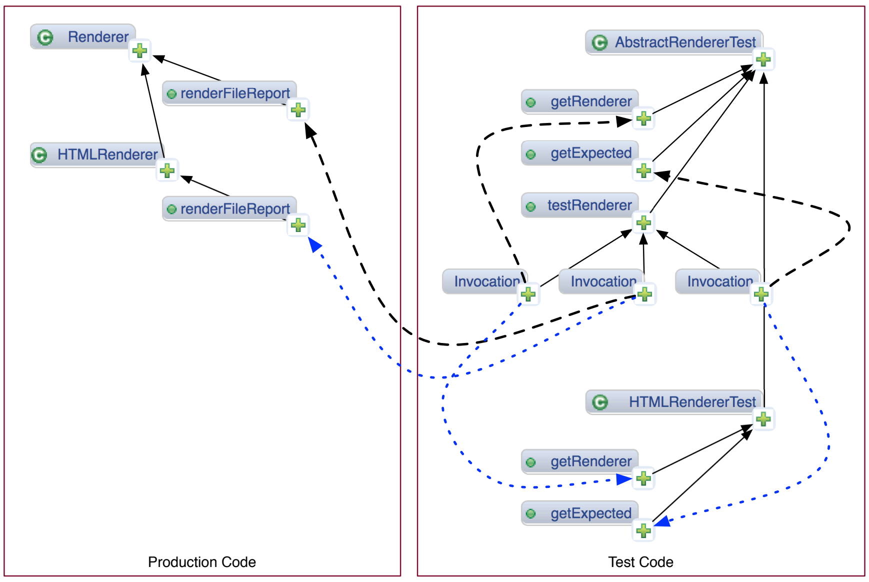

Fig. 4: Change Model for code in Figure 2 with dynamic binding.

model, respectively by *recording* changes that happen during the current development session and by *recovering* previous changes with an analysis of a Subversion repository.

The Logger sits in the background of Eclipse and listens to changes made in the main editor during a development session. We have used Eclipse's `IEle-`

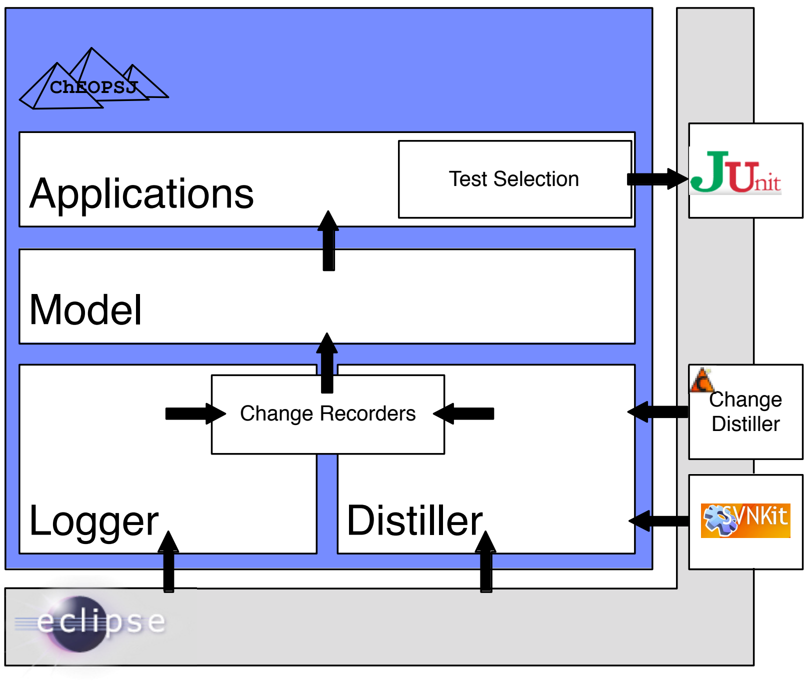

Fig. 5: The layered design of ChEOPSJ.

`mentChangedListener` interface to receive notifications from Eclipse whenever a change is made to Eclipse's internal Java Model (either by changes in the textual Java editor or by changes in other views, like the package explorer). Upon this notification ChEOPSJ will leap into action to record the change; the *ChangeRecorders* will then see what was actually changed. Information about changes up to the level of methods (*i.e.*, additions, removals or modifications of packages, classes, attributes or methods) is contained in the notification. For changes up to statement level (*e.g.*, adding or removing method invocations, local variables or variable accesses) we diff the source code of the changed class before and after the change. To this end ChEOPSJ stores a local copy of the source code before the change.

As the logger does not allow us to analyse existing systems for which a change model has not been recorded, we also provide a Distiller which implements a change recovery technique. Using SVNKit[2] it iterates through all versions stored in the Subversion repository and then looks at the commit message to see what was changed. If a java file was added, an Addition change needs to be instantiated for everything in that file (class, attributes, methods, *etc.*). If a file was removed a Remove change needs to be instantiated for everything in the file. For a modified file we use ChangeDistiller by Fluri et al (2007) to find the difference between the unmodified and the modified versions of the file and then translate a subset of the changes from ChangeDistiller to the changes in our model. This includes

[2] http://svnkit.com

linking the changes with dependencies, which are not present in the model of ChangeDistiller, but which we can derive from the model of the source code.

Limitations: As ChEOPSJ is an academic tool prototype, there are some limitations to its implementation which we need to keep in mind when doing analyses using this tool. As we are using the FAMIX model to model source code entities, we can not define changes on language specific constructs. In the case of Java for instance, this means we can not express changes dealing with generics, annotations, reflexion, *etc.*. We also currently do not maintain all of the FAMIX changes. For instance, below method level we only record changes on method invocations, but not on local variables, statements, expressions, *etc.*. Additionally these method invocations do not include invocations of constructors. This implies that we will not be able to find tests relevant for changes made in a constructor. Another problem that we encountered while doing the analyses is that the current implementation is unable to deal with inheritance in the test code. An example of this and how this can be a problem is explained in Section 4.

3 Case Study Design

To address the research questions in Section 1 we perform a case study and follow the guidelines for case study research by Runeson and Höst (2009). We first precisely describe the procedures for the analyses. Next, we also provide the necessary motivation for and the characteristics of the cases under investigation. This should provide sufficient details so that other researchers can replicate our investigation.

Each of the following subsections describes in detail the procedure for an analysis that we performed and is divided in three parts: a short introduction, where we explain the context of the analysis; the *Setup* in which we explain all the details that are needed for replication purposes (which tools were used and which measurements were collected, *etc.*); and the *Exceptions* that explains why certain data points were omitted or why certain steps were taken to ensure we could actually run the analysis.

We start by explaining the two analyses dealing with **RQ1** to investigate the fault detection ability of the reduced test suites. The setup for the *Mutation Coverage Analysis* is detailed in Section 3.1 and the setup for the *Failing Test Coverage Analysis* is explained in Section 3.2. The next section (Section 3.3) explains how we measured the size and time reduction of the test suites which addresses **RQ2**.

Finally we wrap up this section with an in depth motivation for the cases under investigation in Section 3.4.

3.1 Mutation Coverage Analysis Procedure

A recurring issue with testing experiments is the lack of realistic cases containing documented faults. As a substitute, researchers often plant faults into a correct program by applying a so-called *mutation operator* (Andrews et al, 2005; Rothermel et al, 2001). These mutation operators are chosen based on a fault model and as such, are a close approximation of typical faults occurring in realistic software systems. If a mutation causes a test to fail, the mutation is *killed*, if a mutation can be introduced without breaking any of the tests, then the mutation *survived.* The fault detection ability of the test suite can now be gauged by the percentage of mutations that were killed. This is called the mutation coverage, which provides a reliable metric to measure the quality of a test suite. A higher mutation coverage means that more of the introduced mutants were killed and consequently that your test suite is of better quality.

From this point onwards we will refer to this analysis as the "*Mutation Coverage Analysis*".

Setup: PIT[3] is a tool that does mutation testing based on Java byte code. It supports both Ant[4] and Maven[5] and can thus easily be integrated into the build process of many open source systems. We used the default PIT configuration, in which seven kinds of mutation operators are activated. These are briefly explained in Table 1. Additionally we set PIT to run on four threads, which allows us to speed up the process as PIT can then calculate mutation coverage on four different cores in parallel.

Mutation operator	Description
Conditionals Boundary	Replaces relational operators with their boundary counterpart (*e.g.*, $<$ becomes $<=$, $>=$ becomes $>$, *etc.*).
Negate Conditionals	Replaces all conditionals with their negated counterpart (*e.g.*, $==$ becomes $!=$, $<$ becomes $>=$, *etc.*).
Math	Replaces binary arithmetic operations from either integer or floating-point arithmetic with another operation (*e.g.*, $+$ becomes $-$, $*$ becomes $/$, *etc.*).
Increments	Replaces increments of local variables with decrements and vice versa.
Invert Negatives	Inverts the negation of integer and floating point numbers.
Return Values	Changes the return value of a method depending on the return type (*e.g.*, `non-null` return values are replaced with `null`, integer return values are replaced with 0, *etc.*).
Void Method Call	Removes method calls to void methods.

Table 1: Default mutation operators activated in PIT.

To get a base measurement of the quality of the test suites, PIT is first run considering all classes of the production code and the full test suite.

For each case we distilled changes from the source code repository up to the head revision in the repository. At that point our change model contains all changes ever performed in the history of the case. We then proceed to select all method level changes (Method Additions and Method Removals) and ran the test selection algorithm for each of these changes. The subsets of tests calculated for each method level change are then aggregated on a per class basis, so that we get a subset of tests relevant for each class. It is important to note that since we use all changes in a particular class, we should also find all relevant tests for that class. As such, we can compare the mutation coverage of that class using the reduced test suite with the mutation coverage using the full test suite.

[3] `http://pitest.org`
[4] `http://ant.apache.org`
[5] `http://maven.apache.org`

The configuration for PIT in a project's build file allows us to only target a specific subset of classes and a specific subset of tests to use in the mutation analysis.

In that sense we set up the test selection process as follows: for each class c, we will use all method level changes inside that class to calculate a set of test classes T that our heuristic deemed relevant. We can then use this data to generate separate build files for the project, in which we configure PIT with the target class c and the target tests T.

To calculate the mutation coverage of class c for a given set of tests T, PIT uses the following formula:

$$MutCov(c, T) = \frac{M_{Killed}(c, T)}{M_{Total}(c)}$$

With $M_{Total}(c)$ the total number of mutants introduced in class c and $M_{Killed}(c, T)$ the number of mutants in c killed by test suite T.

To assess the fault detection ability of the reduced test suite, we compare the number of mutants killed with the ones that are killed by the complete test suite. For each class c we can calculate the difference between the mutation coverage of class c when using the complete test suite ($MutCov(c, \mathit{Full})$) with the mutation coverage of class c using only the reduced set of tests for class c ($MutCov(c, \mathit{Reduced})$):

$$MutCovDiff(c) = MutCov(c, \mathit{Full}) - MutCov(c, \mathit{Reduced})$$

With *Full* the set of all tests, *Reduced* the reduced set of tests calculated for class c.

Ideally, the mutation coverage of the reduced test suite should be equal to the mutation coverage of the full test suite ($MutCovDiff(c) = 0$). In that case the reduced test set kills the same number of mutants as the full test set and is therefore as good at exposing faults. When mutation coverage is lower ($MutCovDiff(c) > 0$), it means we have missed some relevant tests in our selection, showing that the fault detection ability of the reduced test suite is worse, however the confidence one might have in the results depends on the percentage of mutants that survive. Since the number of mutants introduced in a class will always be the same, regardless of the test suite used to calculate the mutation coverage, $MutCovDiff(c)$ should never be negative.

For those reduced test suites that achieve a lower mutation coverage than the full test suite, we do an additional investigation to see how much worse it actually is. To do so we look at the average $MutCovDiff$. However this will provide a warped image, as the total number of mutants introduced in a class varies with the size of the class. Missing a high percentage of a low amount of mutants might drastically increase the average. Suppose there is a class with only 2 mutants, running the full test suite kills both (achieving 100% mutation coverage) and running the reduced test suite kills none (resulting in 0% mutation coverage). The difference here would be that the reduced test suite misses 100% of the mutants. To counter this we look at a weighted average of the percentage of fewer mutants killed, using the total number of mutants introduced as a weight. The formula for this calculation is then:

$$Avg_{MutCovDiff} = \frac{\sum\limits_{\forall\, c\, MutCovDiff(c)>0} MutCovDiff(c)}{\sum\limits_{\forall\, c \in MutCovDiff(c)>0} M_{Total}(c)}$$

In which $MutCovDiff(c) > 0$ is a way of representing classes with a reduced test suite with lower mutation coverage than the full test suite.

We run the mutation analysis on the subset of tests using the technique, both with and without taking dynamic binding into account.

Exceptions: In order for PIT to work properly it needs a "green" test suite (*i.e.*, a test suite in which all tests pass). That means that in the base setup for Cruisecontrol we had to exclude the following ten testclasses, since they contained failing tests to begin with:

- `net.sourceforge.cruisecontrol.BuildLoopInformationBuilderTest`
- `net.sourceforge.cruisecontrol.bootstrappers.ExecBootstrapperTest`
- `net.sourceforge.cruisecontrol.builders.ExecBuilderTest`
- `net.sourceforge.cruisecontrol.builders.PipedExecBuilderTest`
- `net.sourceforge.cruisecontrol.util.GZippedStdoutBufferTest`
- `net.sourceforge.cruisecontrol.util.StdoutBufferTest`
- `net.sourceforge.cruisecontrol.functionaltest.BuildLoopMonitorTest`
- `net.sourceforge.cruisecontrol.jmx.CruiseControlControllerJMXAdaptorGendocTest`
- `net.sourceforge.cruisecontrol.jmx.DashboardControllerTest`
- `net.sourceforge.cruisecontrol.util.BuildInformationHelperTest`

For PMD we had to exclude the following two test classes:

- `net.sourceforge.pmd.cpd.XMLRendererTest`
- `net.sourceforge.pmd.RuleSetFactoryTest`

For Historia, we had no tests to exclude as all tests passed successfully.

3.2 Failing Test Coverage Analysis Procedure

When the execution of the developer tests takes too long, developers off-load the testing to the continuous integration server (Beller et al, 2015a,b). This impedes the rapid feedback cycles required by continuous integration (Dösinger et al, 2012), as the test results will only become available after the so-called "nightly build". Both variants of the test-selection heuristic are designed to prevent failing tests on the continuous integration server. Therefore, the ultimate validation is to retroactively investigate whether the heuristic would have selected those failing tests.

From this point onwards we will refer to this analysis as the "*Failing Test Coverage Analysis*".

Setup: In this analysis we looked for cases of failing tests that are stored in the version control system of each case system. More precisely we are looking for tests that failed in the past due to changes in the source code. To find these we wrote a simple python script that iterates all revisions stored in a version control system and builds the system (using either Ant or Maven). If the test suite fails during the build, the test reports are copied and stored for later analysis. In this way we were capable of identifying revisions that had failing tests. Moreover we needed to compare the failing tests in each revision with the failing tests in the previous revision in order to find only those revisions that introduced at least one new failing test.

For each of the revisions that introduced failing tests we used Eclipse's built-in compare capabilities to determine which methods were changed in that revision. We distilled changes for all entities in that revision and then ran the test selection heuristic for the changed methods to determine a reduced set of tests. We then checked if the failing test(s) introduced in that revision are in the reduced set of tests. Note that in contrast to the *Mutation Coverage Analysis* where we dealt with test classes, here we are doing a more fine grained analysis, by creating a reduced set of test methods.

Again we run this analysis for both the technique with dynamic binding taken into account, as the one without.

Exceptions: We basically recreate a local build server for the three projects. By doing so, we run into some problems, since the build environment sometimes changes during the evolution of a software system. In the case of PMD for instance, we find that they only started using Maven in revision 5590 (before that they used Ant). Therefore for PMD we only looked at revisions after 5590 (analysing a total of 2116 revisions). For Cruisecontrol we ran into build failures not related to the testing in revision 4208, so we only analysed the revisions from that point onwards (totalling 418 revisions). For Historia we also ran into build problems and therefore only analysed half of the repository (for a total of 2168 revisions).

In order to successfully build the projects we needed to slightly modify the build files before running the build commands. Both PMD and Historia are built using Maven, which meant sometimes the build file referred to a "SNAPSHOT" dependency. However the snapshot dependencies are not stored in the main Maven repositories, so we removed any occurrence of "-SNAPSHOT" from the build files and built using the closest release dependency. Additionally, the Historia build files were modified to remove one of the modules that used JavaScript code and required additional dependencies.

For both the Cruisecontrol and PMD cases there were instances of failing tests that occurred in several subsequent revisions. We are only interested in the first occurrence of a failing test, thus we ignored all failing tests that already failed in the previous (successful[6]) build. In the case of Cruisecontrol we remained with 22 revisions in which a test failed for the first time, which means that something changed in that revision to make that test fail. Similarly for the PMD case we find 23 revisions that introduce a failing test.

A manual inspection of the changes made in the revisions with failing tests showed that sometimes those revisions had either only changes in the test code itself or no changes to any code at all. For instance, in one revision the commit message stated *"added test for bug 3484404 (NPathComplexity and return statements)"* which means that in this revision they introduced a test to reproduce a bug, which should indeed fail when no changes to the source code were done. Also in the case of PMD, they sometimes made changes in the xml files used by the tests but nothing code-related was changed, which could make the tests fail. As we are only interested in tests that failed due to changes in the production code,

[6] We consider a build successful when no build failures occurred, other than failing tests.

we omitted these kinds of revisions. When we eliminate these instances we remain with 9 revisions for Cruisecontrol and 7 revisions for PMD, where we have actual changes in the production code that lead to at least one new failing test.

In the case of Historia, we had no such problems since failing tests are almost immediately fixed. We found failing tests in 22 revisions. Most of the time it was only a handful of tests that failed. In only three cases we found more than ten tests that failed.

3.3 Size and Time Reduction

As we are reducing the full test suite to smaller test suites relevant for particular classes or methods, we need to evaluate, how much it was reduced (in terms of size) and also whether reducing the test suite was worth the effort (*i.e.*, how long did it take to calculate the reduced set? and what is the return on investment?).

Setup: To evaluate the size reduction, we looked at all the reduced test suites we obtained for both the *Mutation Coverage Analysis* and the *Failing Test Coverage Analysis*. In the case of the *Mutation Coverage Analysis*, due to the technical limitations of using PIT, we are dealing with reductions in terms of test classes. Hence the size reduction is also evaluated on the number of test classes in the reduced set. We measure the test size reduction as the percentage of test classes in the selected subset against the number of test classes in the entire test suite. Thus for each reduced set of tests the reduction can be calculated as follows:

$$Reduction_{Size} = \frac{Nr.\ of\ Test\ Classes\ in\ Reduced\ Suite}{Nr.\ of\ Test\ Classes\ in\ Full\ Suite} \times 100\%$$

In the *Failing Test Coverage Analysis* we are capable of doing a more fine grained reduction (*i.e.*, reducing to a set of test methods rather than test classes). The reduction can in that case be calculated as:

$$Reduction_{Size} = \frac{Nr.\ of\ Test\ Methods\ in\ Reduced\ Suite}{Nr.\ of\ Test\ Methods\ in\ Full\ Suite} \times 100\%$$

The $Reduction_{Size}$ is expressed in terms of a percentage of the full test suite. When this percentage is high, it means that a large portion of tests are included in the reduced test suite. A low percentage means that we reached a good reduction. The lowest $Reduction_{Size}$ we can achieve is dependent on the total number of tests in a system. For instance, in Cruisecontrol the total number of test classes is 183, so when we reduce to 1 single test class we achieve the minimal $Reduction_{Size}$ of 0.55%. For PMD the minimal $Reduction_{Size}$ is 0.62% (or 1 test class out of 162). In Historia the minimal $Reduction_{Size}$ is 0.96% (or 1 out of 104 test classes).

To evaluate the time reduction, we look at (i) how long it takes to calculate a reduced set of tests for a selection of changes ($CalculationTime$) and (ii) how long it takes to run the reduced test suite ($Runtime(Reduced)$) in comparison to the time it takes to run the full test suite ($Runtime(Full)$). Using these measures we can calculate the *Reduction* in terms of runtime as follows.

$$Reduction_{Time} = \frac{CalculationTime + Runtime(Reduced)}{Runtime(Full)} \times 100\%$$

As with $Reduction_{Size}$ the $Reduction_{Time}$ is expressed in terms of percentage of the runtime of the full test suite. The higher this percentage, the worse the reduction. If it is above 100% it even means that calculating and running a reduced set of tests takes longer than running the full set of tests. A low percentage therefore means that we reached a good reduction in terms of time.

The time reduction measurements were taken on a Macbook Pro running a 2.4 Ghz Intel Core i7 (with 4 cores) and 8GB of RAM.

Exceptions: As we are performing these measurements in parallel to both the *Mutation Coverage Analysis* and the *Failing Test Coverage Analysis*, the same exceptions apply.

In the *Mutation Coverage Analysis* all reduced test suites are calculated together, therefore it is difficult to measure the time it took to calculate the individual reduced test suites. That is why the time reduction is only evaluated on the reduced test suites in the *Failing Test Coverage Analysis.*

3.4 Case Selection

We selected three distinct cases — Cruisecontrol, PMD and Historia — on which to run our evaluations:

- Cruisecontrol[7] is both a continuous integration tool and an extensible framework for creating a custom continuous build process.
- PMD[8] is a source code analyser to find a variety of common mistakes like unused variables, empty catch blocks, unnecessary object creations, *etc.*. Additionally, it includes CPD, the copy-paste-detector, which finds duplicated code in several programming languages.
- Historia is an industrial codebase owned by the Flemish Department for Roads and Traffic. It is an administrative tool that is used by the Department for Roads and Traffic to monitor the budgets of its projects. Division managers can link cost estimations to open projects and Historia will then use these estimations to draft a proposal that can then be approved by the necessary authority.

The sizes of these projects in terms of number of lines of code (in KLOC) and number of classes for both the source code and the test code are shown in Table 2. Additionally we also show the count of the total number of test methods implemented inside these Test Classes.

The test coverage of the three cases in terms of Class, Method, Block and Line Coverage is shown in Table 3. This shows that the test suites that come with the cases under investigation are of good quality and we can use these test suites for the evaluation of our heuristics.

[7] `http://cruisecontrol.sourceforge.net`
[8] `http://pmd.sourceforge.net`
[9] `http://metrics2.sourceforge.net`

Project	Nr. of Revisions Analysed	Src KLOC	Nr. of Src Classes	Test KLOC	Nr. of Test Classes	Nr. of Test Methods
Cruisecontrol	4627	31	397	25	183	1157
PMD	7706	60	890	14	162	831
Historia	1944	28	438	28	104	1365

Table 2: Size metrics for both source code and test code (measured with Metrics plugin for Eclipse 1.3.8[9]).

Project	Class Coverage	Method Coverage	Block Coverage	Line Coverage
Cruisecontrol	87% (284/325)	68% (2066/3048)	60% (40175/66550)	61% (9386.4/15331)
PMD	79% (695/877)	68% (4253/6263)	65% (94782/145567)	65% (19596.5/30332)
Historia	93% (399/430)	67% (1955/2909)	60% (36900/61671)	61% (7718.9/12739)

Table 3: Test coverage of all three cases (measured with emma[10]).

The precise selection criteria are summarised in table 4. Each of the three cases adheres to the a common set of criteria: they are well designed object-oriented systems, written in the Java programming language and have gone through evolution (according to the staged model of Bennett and Rajlich (Bennett and Rajlich, 2000)). Both PMD and Cruisecontrol have their evolution stored in Subversion repositories and Historia comes in a Mercurial repository. By mining the log of these software repositories we find that Cruisecontrol has 13 contributors, with 6 *major contributors* (*i.e.*, more than 200 commits); PMD has 24 contributors in its history, with 5 major contributors; and Historia has 8 contributors, with half of them major contributors.

Additionally we can find evidence of agile practices. In particular, these projects use fully automated developer tests as a first defence against regression (all cases use JUnit), and an automatic build system to easily test and deploy a new version (ant for Cruisecontrol and Maven for PMD and Historia).

Technical limitations of our approach imply that they should be written in Java and they should be easily integrated into the eclipse IDE, either by already being Eclipse projects (*i.e.*, coming with a ".project" file in the repository, like in the case of Cruisecontrol) or by easily transforming it to an Eclipse project (for instance, by running the "mvn eclipse:eclipse" command that comes with Maven in the cases of PMD and Historia). Another limitation of our tool is that the Distiller can currenlty only distill changes from Subversion repositories. So in order to distill changes for the Historia case a bash script was run to i) trace all ancestor revisions of the HEAD revision (recursively tracing the "parent revision") ii) then update (using Mercurial) to each of these revisions and committing (using Subversion) that revision to a newly made Subversion repository. In doing so we lose some of the meta-data from the Mercurial repository (*e.g.*, author, timestamp, *etc.*) however we are only interested in the change information which is implicitly maintained in the difference between subsequent revisions.

These three cases show a varied degree in the usage of dynamic binding. We can estimate this degree by looking at how many classes contain overridden methods (*i.e.*, methods that override a method declaration in a superclass) and by looking

[10] `http://emma.sourceforge.net`

Case Selected	Case Properties
All Cases	– Small team of developers. – Evolution stored in source code repository (*e.g.*, Subversion, Git, Mercurial, *etc.*) – Fully automated developer tests (using JUnit). – Automatic build system (*e.g.*, Maven, Ant, *etc.*). – Written in Java programming language.
Cruisecontrol	– Open source case. – Developer tests consist mainly of unit tests. – Small to moderate usage of dynamic binding.
PMD	– Open source case. – Developer tests consist of many integration tests, running several scenarios to exercise a particular rule. – Heavy use of dynamic binding.
Historia	– Industrial case. – Use of continuous integration; failing tests must be repaired before other changes can be made. – Developer tests consist mainly of unit tests, with a few smoke tests exercise a happy-day scenario. – Test coverage is monitored (branch coverage, *etc.*)

Table 4: Criteria for case selection.

at the average ratio between the number of overridden methods (NORM) and the total number of methods (NOM) per class. These numbers are summarised in the box plots in Figure 6.

We can give an estimation for the degree of dynamic binding usage per class:

$$DB_c = \frac{NORM_c}{NOM_c}$$

With DB_c the degree of dynamic binding usage for class c; $NORM_c$ the number of overridden methods in class c; and NOM_c the total number of methods in class c.

We can then give an estimation for the degree of dynamic binding usage for an entire project ($DB_{project}$), by looking at the average DB_c of all classes.

$$DB_{project} = \frac{\sum\limits_{\forall\, c \,\in\, Classes} DB_c}{N}$$

With $Classes$ the set of all classes in the project and N the number of classes in the project.

Cruisecontrol: This is a case in which we encounter a small to moderate usage of dynamic binding and should have unit tests covering most of the code base (see coverage metrics in Table 3). Cruisecontrol has 47 classes out of 397 (or 12%) that have overridden methods. The fact that 88% of the classes have no overriden methods, is also visible in Figure 6, as the minimum, median and both first and third quartile of the DB_c are at 0. The $DB_{project}$ for Cruisecontrol lies at 4.3. Showing that it does indeed show a small usage of dynamic binding.

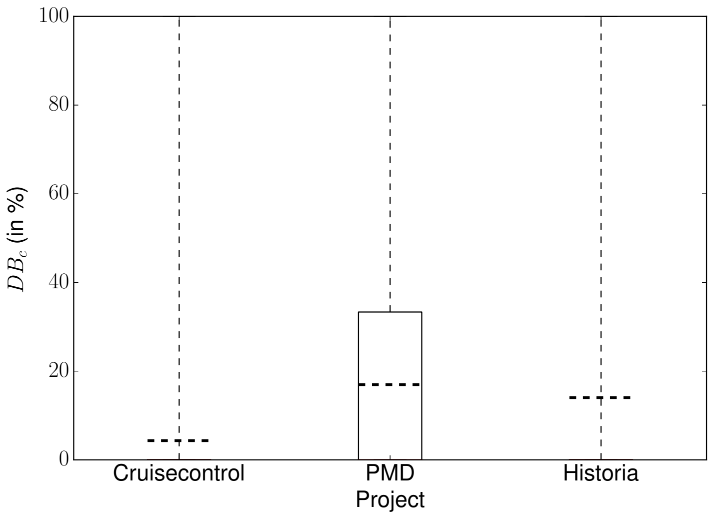

Fig. 6: Degree of dynamic binding usage in the three cases. The horizontal dashed lines show the $DB_{project}$.

PMD: This is a case where we encounter a heavy use of dynamic binding. The developer tests consist of unit tests, but also many integration tests that run several scenarios to exercise a particular rule. PMD has 401 out of 890 classes (or 45%) that have overridden methods and the $DB_{project}$ lies at 17. This shows that this project has a high degree of dynamic binding usage.

Historia: This is a case in which there is a known use of continuous integration and a corresponding quality assurance. In particular, when a unit test fails, it must be repaired before new features are added. Apart from that the quality of the test suite is guarded by monitoring test coverage through the means of branch coverage. With regard to the usage of dynamic binding, Historia lies somewhere between PMD and Cruisecontrol. 98 out of 438 classes (or 22%) have overridden methods making it similar to Cruisecontrol, but those 98 classes display a high number of overridden methods resulting in a $DB_{project}$ of 14. Similar to the Cruisecontrol case the minimum, median and both first and third quartiles of the DB_c lie at 0, which is again explained by the low number of classes with overridden methods.

By talking to one of the main developers we know that they use a continuous integration framework and failing tests in the repository should be avoided as much as possible.

4 Results for RQ1 – Fault Detection Ability

In this section we address **RQ1** (*What is the fault detection ability of the reduced test suites?*) by analysing the results of both the *Mutation Coverage Analysis* (Section 4.1) and the *Failing Test Coverage Analysis* (Section 4.2).

4.1 Mutation Coverage Analysis Results

We performed the *Mutation Coverage Analysis* on the three cases starting from the following revisions: rev. 4627 for Cruisecontrol; rev. 7706 for PMD and rev. 4336 for Historia. We first ran PIT to calculate the mutation coverage on these revisions using the full test suite. We show the results of the mutation coverages for each case using the full test suite in Table 5. We then proceeded to use our tool to distill changes and dependencies from the first revision of each case up to these revisions. In Table 6 we show the number of changes that were distilled for each case broken down into additions (indicated by the ✚ icon) and removals (indicated by the ✖ icon) for each of the structured program entities we support. We then used these changes to calculate reduced sets of tests for each class in each case. We then calculated the mutation coverage for each class individually using these reduced sets of tests, which is then compared to the mutation coverage of each class using the full test suite.

Case	Total Mutants Killed	Total Mutants Introduced	Mutation Coverage
Cruisecontrol	3952	7128	55%
PMD	10361	19831	52%
Historia	3915	7899	50%

Table 5: Mutation coverages with the full test suites.

Entity Type	Change Type	Cruisecontrol	PMD	Historia
Packages	✚	36	258	312
	✖	0	0	0
Classes	✚	863	3653	1650
	✖	164	2375	5
Methods	✚	7102	22694	11565
	✖	1965	16087	1167
Fields	✚	3331	8970	5340
	✖	1133	6415	601
Invocations	✚	17493	47797	29953
	✖	7279	38387	6940
Total	✚	28825	83372	48820
	✖	10541	63264	8723
	Total	39366	146636	57543

Table 6: Number of changes distilled from the source code repositories.

The results of these comparisons are summarised in Figure 7. We detail the results for each case separately in the following paragraphs:

Cruisecontrol: In this case we could generate mutants in 249 classes (on average 29 mutants per class). When taking dynamic binding into account we could calculate

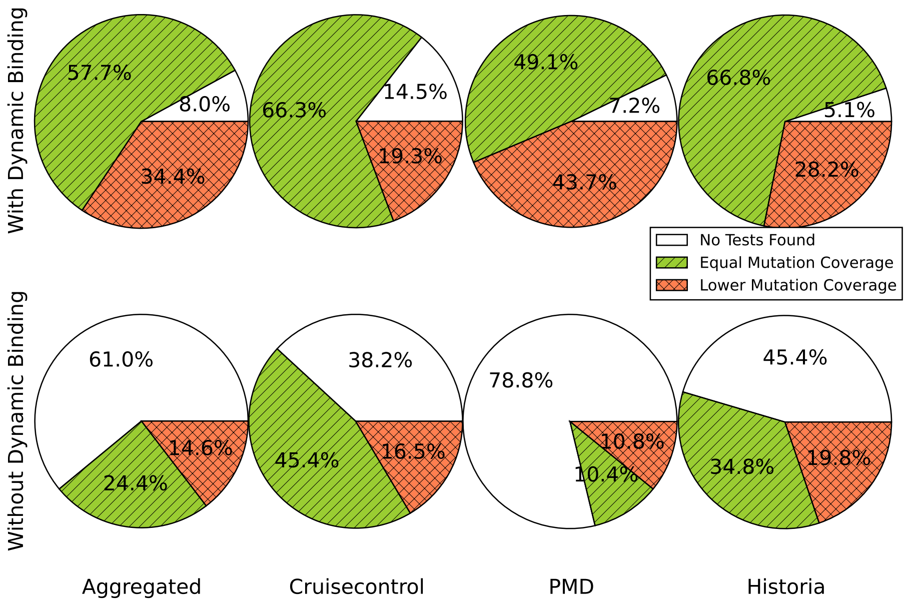

Fig. 7: Lower and equal mutation coverages for the reduced test suites.

a reduced test suite for all of these 249 classes. Without it however we could only generate a reduced test suite for 154 of these. If we ignore the classes for which we could not generate a reduced test suite we find that with dynamic binding we have 81.1% (202 out of 249) classes that have an equal mutation coverage when compared to the mutation coverage of the full test suite. Without dynamic binding there is 61% (94 out of 154) where there is an equal mutation coverage.

Consequently, there are also a number of reduced test suites, for which fewer mutants were killed: 26 (10.4%) when we take dynamic binding into account and 44 (28.6%) when we do not take dynamic binding into account.

When we apply our formula to calculate the weighted average of how much lower the mutation coverage is in the classes with a lower mutation coverage ($Avg_{MutCovDiff}$), we find that both with and without dynamic binding this results in on average 14% less mutants killed.

PMD: In this case we could generate mutants in 665 classes (on average 30 mutants per class). With dynamic binding we can calculate reduced test suites for most of these (616) which is a lot better than without dynamic binding, where for only 141 classes a reduced test suite could be calculated. This can be explained by the fact that PMD is a highly polymorphic project (see Section 3.4). The reason we are unable to calculate a reduced test suite for all classes (even with dynamic binding) is that not all classes are covered by the test suite. So those are classes where even with the full test suite we get a mutation coverage of 0%. When we only look at the classes for which we could generate a reduced test suite we see that in both cases with and without dynamic binding, that about half (315 out of 616 (51.5%) and 64 out of 141 (45.4%)) have an equal mutation coverage and the other half (291 out of 616 (47.2%) and 70 out of 141 (49.6%)) have a lower mutation coverage.

For the classes with a reduced test suite that achieves a lower mutation coverage, we can calculate the $Avg_{MutCovDiff}$. We find that with dynamic binding on average 24% less mutants were killed and without dynamic binding 11% less mutants were killed.

Historia: In this case we could generate mutants in 394 classes (on average 20 mutants per class). With dynamic binding taken into account we can calculate reduced test suites for 374 of these. Which is substantially higher than the 215 classes we could generate a reduced test suite for without taking dynamic binding into account. When we only look at the classes for which we generated a reduced test suite, we find that with dynamic binding taken into account; 92.5% (346 out of 374) of the reduced test suites have the same mutation coverage as when we use the full test suite. Without dynamic binding we also find a majority of reduced test suites (85.1% or 183 out of 215) that have the same mutation coverage as the full test suite.

There are few (19 out of 374 (5.1%) and 28 out of 215 (13.0%)) that have a lower mutation coverage, meaning that fewer mutations were killed using the reduced test suite when compared to the full test suite. Looking at these we calculate the $Avg_{MutCovDiff}$ and find that with dynamic binding, on average 50% less mutants were killed and without dynamic binding on average 21% less mutants were killed.

The fact that some reduced test sets kill fewer mutants means that we missed some relevant test cases. Although we can still put this in perspective: on average 14% less mutants were killed in Cruisecontrol; 24% and 11% less mutants (with and without dynamic binding respectively) were killed in PMD; and the reduced test suites for Historia killed 50% and 21% less mutants (with and without dynamic binding respectively). The reasons for missing some relevant tests (even with dynamic binding taken into account) are twofold: on the one hand one of the test classes in PMD (`PMDTaskTest`) tests PMD's functionality to work with Ant build files. To this end they use third party testing code (`org.apache.tools.ant.BuildFileTest`) which does not directly invoke the PMD code, but merely runs the necessary build targets and verifies that the output contains the expected warnings. On the other hand as we mentioned in Section 2.3 our current implementation of the approach is unable to deal with inheritance in the test code itself. Especially in PMD this is a problem where one particular test class (`SimpleAggregatorTst`) has 46 subclasses. The actual test method that is being run is implemented in the base class, yet the test's behaviour is driven by the `setUp` methods in the subclasses.

One of the examples this caused a lower mutation coverage was for the class `RuleSet`. This class had 87 mutants introduced, of which the full test suite killed 57 and the reduced test suite killed 46 (with dynamic binding taken into account). So 5 less mutants were killed by the reduced test suite. Upon closer inspection 1 of those mutants was killed by the `PMDTaskTest` which uses the third party testing code and the 4 other mutants were in the full test suite all killed by the test `SignatureDeclareThrowsExceptionTest.testAll()`. This test was missed by our implementation, as the test structure itself is built using derived classes (as shown in Listing 1). The problem then lies in the fact that the method `RuleSet.usesTypeResolution()` is invoked somewhere in the execution of the `testAll` in `SimpleAggregatorTst`. However as the test class `SignatureDeclareThrowsExceptionTest` is a subclass of `SimpleAggregatorTst` it inherits the `testAll` method. This method will have a changed behaviour due to configurations set in the `setUp()` method of `SignatureDeclareThrowsExceptionTest`. Our heuristic is perfectly capable of identifying `SimpleAggregatorTst.testAll()` as a relevant test, however since `SignatureDeclareThrowsExceptionTest` has no direct implementation of the `testAll` method it will not be included in the reduction.

```
public abstract class SimpleAggregatorTst extends RuleTst {
  [...]
   private List<Rule> rules = new ArrayList<Rule>();

  /**
   * Add new XML tests associated with the rule to the test suite.
   * This should be called from the setup method.
   */
  protected void addRule(String ruleSet, String ruleName) {
    rules.add(findRule(ruleSet, ruleName));
  }

  @Test
  public void testAll() {
    [...] // does something with the rules
  }
  [...]
}

public class SignatureDeclareThrowsExceptionTest extends SimpleAggregatorTst {

  @Before
  public void setUp() {
    addRule("java-typeresolution", "SignatureDeclareThrowsException");
  }


  [...]
}
```

Listing 1: SignatureDeclareThrowsExceptionTest

We have shown that (when we only look at the classes for which we could calculate a subset of tests) on average 62.6% of the reduced test suites kill the same amount of mutants as the complete test suite. The Historia and Cruisecontrol cases perform much better than PMD, where we saw a fifty-fifty distribution between subsets with equal and subsets with lower mutation coverage.

On average 62.6% of the reduced test suites have the same quality as the full test suite. Despite fewer mutants being killed in 37% of the cases, the confidence one might have in the reduced test suites is still acceptable as on average only 22% less mutants are killed.

4.2 Failing Test Coverage Analysis Results

To assess whether our approach might work in practice we look at test failures in the history of our three cases. As mentioned in Section 3.2 we found 9 revisions in Cruisecontrol that introduce at least one new failing test, 7 such revisions in PMD and 22 revisions with new failing tests in Historia. For each of these 38 revisions we distilled changes from the first revision of the case up to that revision. Only the methods changed in these revisions are used for the test selection heuristics and we checked if the failing tests in the revisions are in the calculated subset of tests. Table 7 shows the detailed results of this analysis. For each revision we analysed this table shows the total number of tests, the number of failing tests introduced, the number of changes distilled by our tool and the number of changed methods. Moreover for both the heuristic with and without dynamic binding it shows the number of tests in the reduced set of tests as well as the number of failing tests that are in the reduced set of tests. These results are color coded to map the results to each of the categories in Figure 8.

Figure 8 shows a summary of the results of this analysis in terms of four distinct categories:

- *All Failing Tests Found*: This is the optimal results, in which all of the failing tests are covered by the reduced test suite.
- *Some Failing Tests Found*: These are the results in which we found a reduced set of tests, yet only some of the failing tests are covered by the reduced set. This means that our heuristic is capable of finding some of the relevant tests, yet not all of them.
- *No Failing tests found*: For the results in this category we were capable of calculating a reduced set of test methods, however this reduced set did not contain any of the failing tests.
- *No Tests Found*: In this category our heuristic was unable to calculate a reduced set of tests for the selected changes.

In the following paragraphs the results are detailed for each case separately:

Cruisecontrol: We calculated the reduced test sets for the changes in the nine revisions of Cruisecontrol that introduce a failing test. We find that when we take dynamic binding into account, in all but one case we find all failing tests. In the one revision where we only found some of the failing tests, we missed 3 out of 17. If we do not take dynamic binding into account, we only have about half the cases where we find all or some of the failing tests. In the other half we either find none of the failing tests or we could calculate no reduced test suite at all.

PMD: In this case, we calculated reduced test sets for the changes in seven revisions. We find that when we take dynamic binding into account, in all but one case, we could generate reduced test suites that contained all of the newly failing tests.

When we do not take dynamic binding into account we find that in all but one of the seven revisions we are in fact unable to generate a reduced test suite, which is again due to the highly polymorphic nature of some of PMD's core functionality (see Figure 6). Still we find one revision in which the reduced test suite was capable of finding the single new failing test.

						With Dynamic Binding		Without Dynamic Binding	
Case	Revision	# Tests total	# (New) Failed Tests	# Distilled Changes	# Changed Methods	# Tests in Reduced Set	# Failed Tests inReduced Set	# Tests in Reduced Set	# Failed Tests in Reduced Set
Cruisecontrol	4377	1004	1	36510	3	898	1/1	0	0/1
	4416	1028	1	36999	1	915	1/1	0	0/1
	4564	1118	17	38780	1	974	14/17	36	11/17
	4571	1124	7	39058	26	979	7/7	16	7/7
	4590	1130	6	39115	37	980	6/6	11	6/6
	4591	1138	1	39151	9	988	1/1	57	1/1
	4613	1148	1	39253	2	994	1/1	23	0/1
	4623	1152	1	39298	12	999	1/1	13	1/1
	4624	1157	1	39335	3	1003	1/1	27	0/1
PMD	6214	957	3	93607	1	439	3/3	0	0/3
	6374	982	1	94355	19	438	1/1	0	0/1
	6388	995	1	94499	45	447	1/1	7	1/1
	7275	821	2	101106	9	485	2/2	0	0/2
	7469	826	8	101920	8	496	0/8	0	0/8
	7547	826	3	145861	4	496	3/3	0	0/3
	7680	831	1	146545	1	518	1/1	0	0/1
Historia	2370	691	2	27757	1	69	2/2	1	1/2
	2415	692	8	27724	5	24	8/8	0	0/8
	2424	693	5	27813	9	592	5/5	71	4/5
	2449	692	64	27890	52	604	64/64	6	0/64
	2538	719	3	29251	11	72	3/3	3	0/3
	2542	719	1	29269	2	69	1/1	1	0/1
	2566	730	1	29432	51	615	1/1	53	0/1
	2598	730	1	29454	28	618	1/1	16	0/1
	2719	760	1	29733	15	139	1/1	111	1/1
	2789	763	2	30046	21	717	2/2	2	0/2
	2807	769	1	30049	4	647	1/1	113	1/1
	2857	769	18	30088	1	141	18/18	113	18/18
	2866	769	1	30100	2	4	1/1	2	1/1
	2919	769	3	30413	7	646	3/3	11	3/3
	2927	769	3	30434	5	69	3/3	1	1/3
	2931	769	1	30443	9	650	0/1	114	0/1
	2941	770	3	30544	1	12	3/3	0	0/3
	3897	1162	1	40491	10	385	1/1	151	1/1
	4215	1273	18	44075	68	1109	18/18	153	4/18
	4229	1289	1	44524	2	1155	1/1	0	0/1
	4230	1288	5	44528	9	1156	5/5	25	3/5
	4334	1365	2	45944	2	5	2/2	0	0/2

Table 7: Measurements taken on revisions with failing tests.

Historia: The changes for 22 revisions were used to calculate reduced sets of tests. Our approach with dynamic binding was capable of identifying all but one failing test in its reduced test sets. The one failing test that we missed, was due to a change in a constructor, that is invoked from the failing tests. We miss this instance, since our change model currently does not keep track of constructor invocations (see Section 2.3). Without dynamic binding we find that in only half of the reduced test suites, all or some of the failing tests are present. In the other half we either have no test found at all or none of the failing tests.

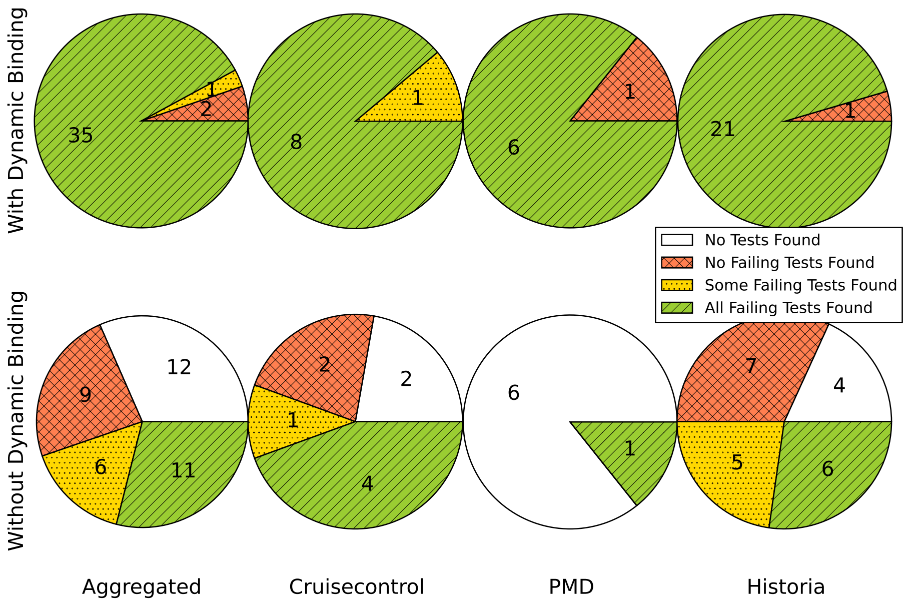

Fig. 8: Quality of reduced test suites for *Failing Test Coverage Analysis*.

When we look at the aggregated results of the *Failing Test Coverage Analysis*, we see that when we take dynamic binding into account we can find all failing tests in all but three cases. In one of those three cases we were still even capable of finding 14 out of 17 failing tests. The other two cases can be explained by (i) the fact that our change model currently is not capable of linking constructor invocations to the constructor being invoked (see Section 2.3) and (ii) third party testing code that was used (this is discussed in more detail in Section 6).

Looking at the data obtained when we do not take dynamic binding into account, we get that in 17 out of 38 (45%) cases we find all or some of the failing tests. In the other half we either find none of the failing tests or no test reduction at all. This last category is what happened most in the PMD case. Again we see that the Historia and Cruisecontrol cases performed much better.

Taking dynamic binding into account we find that almost all of the failing tests are in the reduced test suites. Without dynamic binding this only happens in half the cases.

5 Results for RQ2 – Test Reduction

In this section we address **RQ2** (*How much can we reduce the unit test suite in the face of a particular change operation?*) by analysing the $Reduction_{Size}$ in both the *Mutation Coverage* and the *Failing Test Coverage Analysis* (Section 5.1). We also analyse the $Reduction_{Time}$ in the *Failing Test Coverage Analysis* (Section 5.2).

For the *Failing Test Coverage Analysis* we provide a table with detailed measurements for each revision analysed (Table 8). Apart from the sizes and runtime of the full and reduced sets of tests we also provide the calculation time for the reduced set of tests and the number of changes distilled from the repository as well as the number of dependencies distilled with both variants of our heuristic. Note that the number of dependencies distilled is much higher when taking into account dynamic binding. Particularly in the case of PMD we see that the difference is of an order of magnitude.

						With Dynamic Binding				Without Dynamic Binding			
	Revision	# Tests total	# Runtime of Full Set (in seconds)	# Distilled Changes	# Changed Methods	# Distilled Dependencies	# Tests in Reduced Set	# Calculation Time of Reduced Set (in seconds)	# Runtime of Reduced Set (in seconds)	# Distilled Dependencies	# Tests in Reduced Set	# Calculation Time of Reduced Set (in seconds)	# Runtime of Reduced Set (in seconds)
Cruisecontrol	4377	1004	42.06	36510	3	100787	898	8.17×10^{-1}	29.33	33879	0	1.93×10^{-4}	0
	4416	1028	46.27	36999	1	102161	915	8.22×10^{-1}	33.11	34275	0	3.60×10^{-5}	0
	4564	1118	57.70	38780	1	111057	974	4.51×10^{-1}	44.34	36082	36	2.87×10^{-4}	12.27
	4571	1124	58.26	39058	26	112134	979	9.19×10^{0}	44.89	36306	16	2.15×10^{-4}	10.30
	4590	1130	58.26	39115	37	112234	980	1.22×10^{1}	44.89	36356	11	2.69×10^{-4}	10.22
	4591	1138	58.26	39151	9	112309	988	3.36×10^{0}	44.89	36387	57	1.41×10^{-3}	12.36
	4613	1148	58.54	39253	2	112548	994	9.88×10^{-1}	45.05	36457	23	2.95×10^{-4}	1.44
	4623	1152	58.81	39298	12	112627	999	4.86×10^{0}	45.32	36485	13	1.84×10^{-3}	0.41
	4624	1157	58.96	39335	3	112911	1003	1.44×10^{0}	45.47	36494	27	1.55×10^{-3}	10.32
Historia	2370	691	17.57	27757	1	152708	69	4.70×10^{-3}	0.81	29387	1	1.45×10^{-4}	0.01
	2415	692	17.72	27724	5	152819	24	3.21×10^{-4}	0.21	29486	0	7.70×10^{-5}	0
	2424	693	17.36	27813	9	152845	592	1.95×10^{-1}	14.42	29534	71	1.41×10^{-3}	2.21
	2449	692	17.04	27890	52	153385	604	2.61×10^{-1}	14.49	29650	6	2.37×10^{-4}	0.03
	2538	719	19.24	29251	11	162821	72	1.90×10^{-3}	0.82	31077	3	9.50×10^{-5}	0
	2542	719	19.22	29269	2	162852	69	1.09×10^{-3}	0.83	31089	1	2.30×10^{-5}	0.01
	2566	730	19.52	29432	51	163243	615	1.46×10^{0}	16.84	31266	53	6.93×10^{-4}	2.52
	2598	730	21.02	29454	28	163288	618	1.12×10^{0}	18.42	31288	16	2.42×10^{-3}	0.18
	2719	760	20.54	29733	15	163964	139	1.71×10^{-2}	5.10	31564	111	6.38×10^{-3}	4.91
	2789	763	20.53	30046	21	165336	717	3.17×10^{-1}	20.34	31901	2	1.95×10^{-4}	0.02
	2807	769	20.89	30049	4	165341	647	2.36×10^{-1}	11.00	31905	113	3.80×10^{-4}	4.93
	2857	769	21.52	30088	1	165515	141	1.43×10^{-3}	4.80	31934	113	4.25×10^{-4}	4.59
	2866	769	22.00	30100	2	165551	4	7.40×10^{-5}	0.03	31955	2	2.40×10^{-5}	0.02
	2919	769	20.91	30413	7	166464	646	8.13×10^{-2}	11.04	32270	11	8.60×10^{-5}	0.09
	2927	769	21.12	30434	5	166486	69	3.69×10^{-3}	0.85	32292	1	3.40×10^{-5}	0.01
	2931	769	20.99	30443	9	166553	650	3.90×10^{-1}	10.94	32301	114	5.56×10^{-4}	4.96
	2941	770	21.10	30544	1	166564	12	8.40×10^{-5}	0.11	32312	0	1.70×10^{-5}	0
	3897	1162	23.77	40491	10	231654	385	1.63×10^{-2}	5.80	43097	151	5.67×10^{-4}	4.73
	4215	1273	29.06	44075	68	260871	1109	5.20×10^{-1}	12.61	47037	153	7.49×10^{-4}	0.30
	4229	1289	25.29	44524	2	265057	1155	2.24×10^{-1}	22.91	47520	0	2.30×10^{-5}	0
	4230	1288	25.39	44528	9	265090	1156	3.82×10^{-1}	23.03	47523	25	1.01×10^{-4}	0.23
	4334	1365	25.08	45944	2	272786	5	6.50×10^{-5}	0.10	49035	0	6.00×10^{-6}	0
PMD	6214	957	22.56	93607	1	1655210	439	1.95×10^{0}	6.97	103008	0	1.30×10^{-5}	0
	6374	982	24.96	94355	19	1656796	438	8.32×10^{0}	6.11	103635	0	3.20×10^{-5}	0
	6388	995	24.52	94499	45	1659890	447	4.83×10^{1}	6.37	103741	7	1.54×10^{-3}	0.004
	7275	821	28.15	101106	9	1926090	485	3.30×10^{1}	16.76	110303	0	6.00×10^{-5}	0
	7469	826	23.32	101920	8	1940090	496	3.43×10^{1}	7.25	111001	0	1.16×10^{-4}	0
	7547	826	28.71	145861	4	3721435	496	1.50×10^{1}	11.86	184161	0	5.10×10^{-5}	0
	7680	831	34.72	146545	1	3724971	518	5.09×10^{0}	18.84	185190	0	1.03×10^{-3}	0

Table 8: Measurements taken on revisions with failing tests.

5.1 Test Size Reduction

We start by analysing the reduction in terms of size ($Reduction_{Size}$). We differentiate between test size reductions in the *Mutation Coverage Analysis* (Figure 9) and the test size reductions in the *Failing Test Coverage Analysis* (Figure 10). In both we make the distinction between the heuristic without taking dynamic binding into account and the one with the dynamic binding implementation.

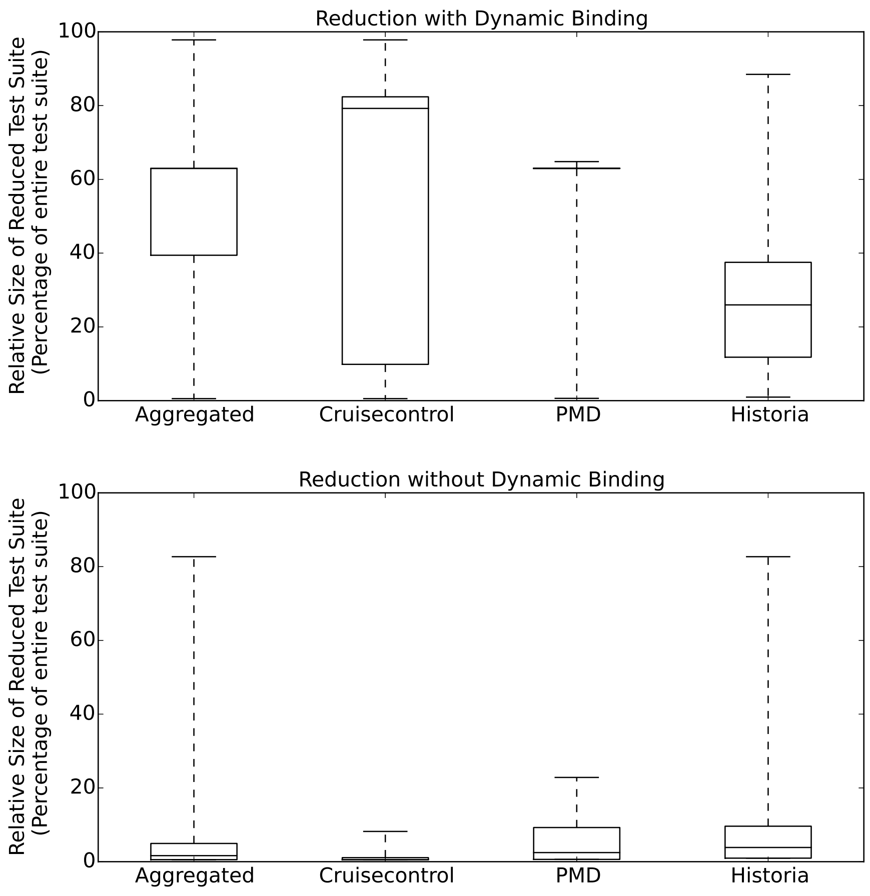

Fig. 9: Size reduction in the *Mutation Coverage Analysis*.

In the reductions for the *Mutation Coverage Analysis* (Figure 9) we can see that when we do not take dynamic binding into account, in most cases the reduced test suite consists of a single unit test. For Cruisecontrol this implies a reduction of 0.55% (1 out of 183 test classes); for PMD this corresponds to 1 out of 162 test classes or 0.62% and for Historia this comes down to 0.91% or 1 out of 104 test classes. When we run our heuristic that takes dynamic binding into account the

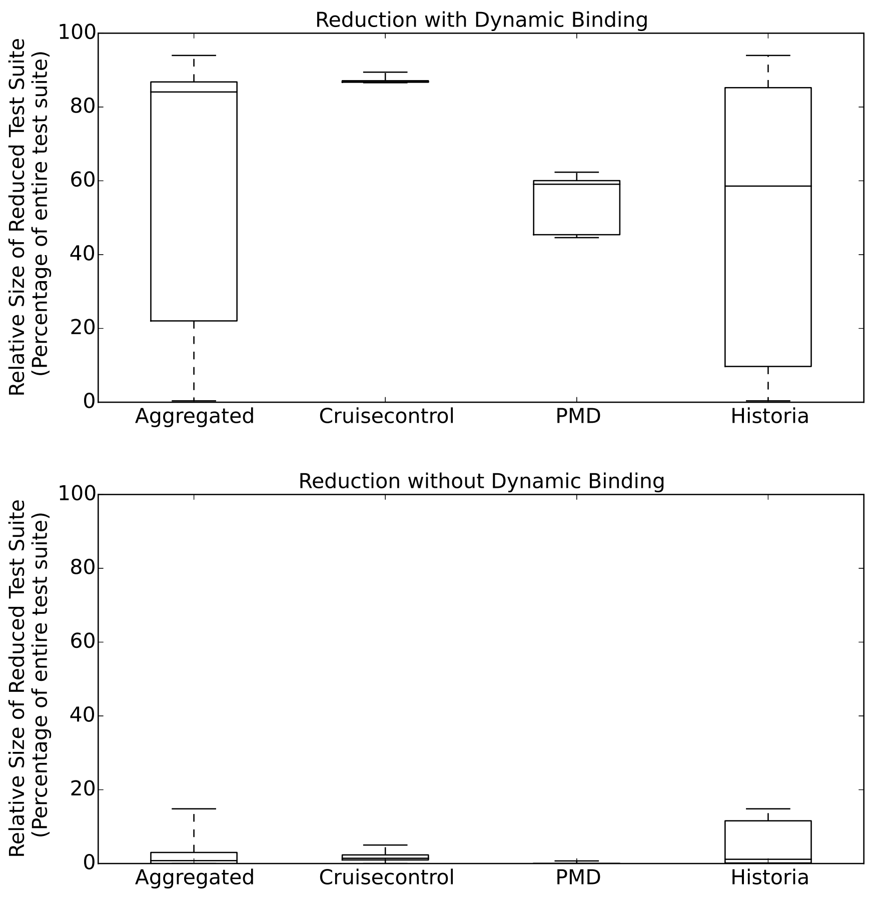

Fig. 10: Size reduction in the *Failing Test Coverage Analysis*.

reductions are on average much worse. In a lot of cases we even see hardly any reduction. In the case of Cruisecontrol we find that most cases are reduced to 98% or 179 out of 183 testclasses. For PMD all but 8 reductions are a reduction of 63% (or 102 out of 162 test classes). And in the case of Historia, what we see most is a reduction to 87.5% (or 91 out of 104 test classes). On the other hand we also see some classes with a reduced test suite of less than 20% of the full test suite. For Cruisecontrol there are 7 such classes, for PMD there are 8 and for Historia there are 112 which is 28.4% of all classes.

We can see similar results in the reductions for the *Failing Test Coverage Analysis* (Figure 10 and Table 8). When we do not take dynamic binding into account for all three cases all of the reduced test suites are less than 15% of the full test suite. When we do take dynamic binding into account, we see that for Cruisecontrol, we find all nine cases result in a reduction to around 86% of the full test suite. In the case of PMD all seven cases result in a reduction to 45% or 60% of the full test suite. Finally for Historia we find 10 cases where the test suite is reduced to less than 20% of the full test suite and 11 cases where the test suite was reduced to between 84% and 94%.

For Cruisecontrol we find that in all instances there is nearly no reduction when taking dynamic binding into account. This is a similar observation as the one made by Graves et al (2001). This could be caused by the way Cruisecontrol is tested (*e.g.*, more integration or system testing than unit testing) or by the fact that the dynamic binding has a more significant effect than we anticipated. Curiously, even though PMD has a higher $DB_{Project}$ than Cruisecontrol it still gets on average better reductions when taking dynamic binding into account. This is because we still miss some relevant tests with PMD, due to its use of third party testing code and even more so the number of inheritance relationships in the test code of PMD (see example in Section 4.1).

We can conclude that when we do not take dynamic binding into account we get much better reductions than when we do. Nevertheless even when taking dynamic binding into account we can sometimes still get a very nice reduction, but this happened mostly in the Historia case.

We have shown that in both the *Failing Test Coverage Analysis* as well as the *Mutation Coverage Analysis*, the solution that does not take dynamic binding into account has a much higher reduction. In most cases we can even say that more than 99% of the entire test suite is discarded. Which means that for many changes only a handful of tests need to be rerun. When dynamic binding is taken into account, we find that both Cruisecontrol and PMD in most cases have nearly no reduction. Only in the industrial case (Historia), we can sometimes have a good reduction even with dynamic binding taken into account. This could be explained by the way we deal with dynamic binding (*i.e.*, only looking at the identifier used in a method invocation). In the PMD case the use of the Visitor pattern in particular introduces a `visit` identifier in 998 method declarations, which dominates the way the system behaves at runtime (and during the tests).

Without dynamic binding, we can reduce the full test suite to only a handful of tests. When taking dynamic binding into account, this is true for only a few cases. In the other cases taking dynamic binding into account often boils down to selecting near to the complete set of tests.

5.2 Runtime Evaluation

In this section we present and analyse the reduction in terms of runtime of the reduced test suite as well as the time needed to calculate the reduced test suite.

We start by looking at the calculation time. Figure 11 shows two scatterplots in which we have plotted the time needed to calculate the reduced set of tests (Y-axis) in relation to the number of method level changes selected for the calculation (X-axis).

A general trend and also the most obvious is that the more changes are selected the longer it will take to perform the calculation. However the most interesting observation that we can make is that when we compare the calculation time for the approaches with and without dynamic binding included, we see that there is a difference in several orders of magnitude. Calculating the reduced test suite using the approach with dynamic binding takes milliseconds to seconds. While the calculation without dynamic binding takes mere microseconds, compared to the milliseconds that are necessary for the case with dynamic binding. This is because when we take dynamic binding into account the number of dependencies between additions of method invocations and additions of method declarations is much higher (see Table 8). In the PMD case for instance, the use of the *Visitor* pattern results in a `visit()` method, which is declared 998 times and invoked 602 times, therefore each of these 602 invocations has a dependency to each of the 998 declarations. So instead of there being 602 dependencies from an invocation to a method being called, there are now 600796 (602×998) dependencies.

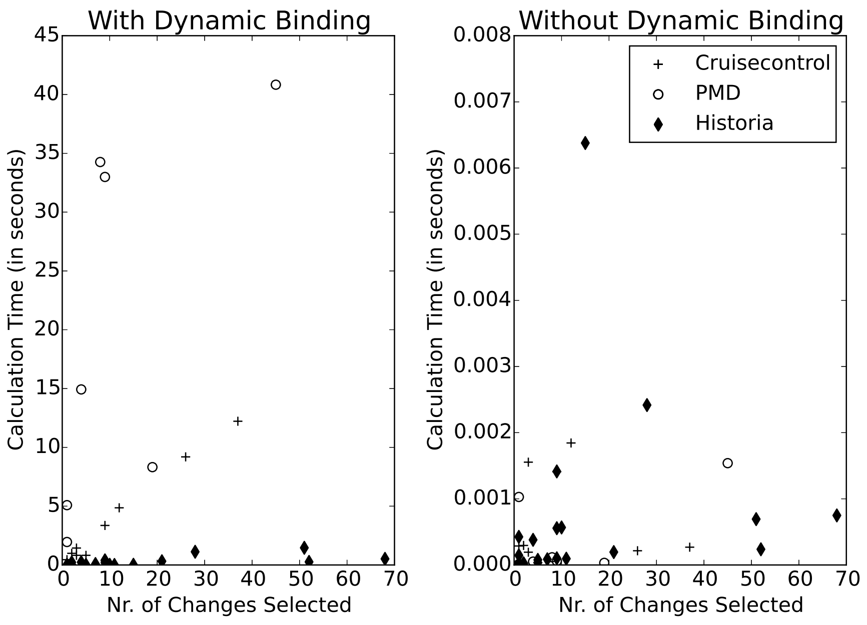

Fig. 11: Calculation time (in seconds) of reduced test suite in relation to number of changes selected for calculation.

Looking at the $Reduction_{Time}$ in Figure 12, what we see is that when we do not take dynamic binding into account the combined calculation time and runtime of the reduced sets are in all cases less than 25% of the runtime of the full test suite. When all data is aggregated the $Reduction_{Time}$ is on average 9%. Which means that calculating and running a reduced set of tests is on average 11 times faster than running the complete test suite. When we take dynamic binding into account the time won is much lower. The average $Reduction_{Time}$ on the aggregated data is 65% of the runtime of the full test suite, which means that on average calculating and running a reduced set of tests is 1.5 times faster than running the full test suite.

Cruisecontrol: Looking at the individual cases we see that for Cruisecontrol the average *ReductionTime* without dynamic binding is 14% (7 times faster) and with dynamic binding is 82% (1.2 times faster).

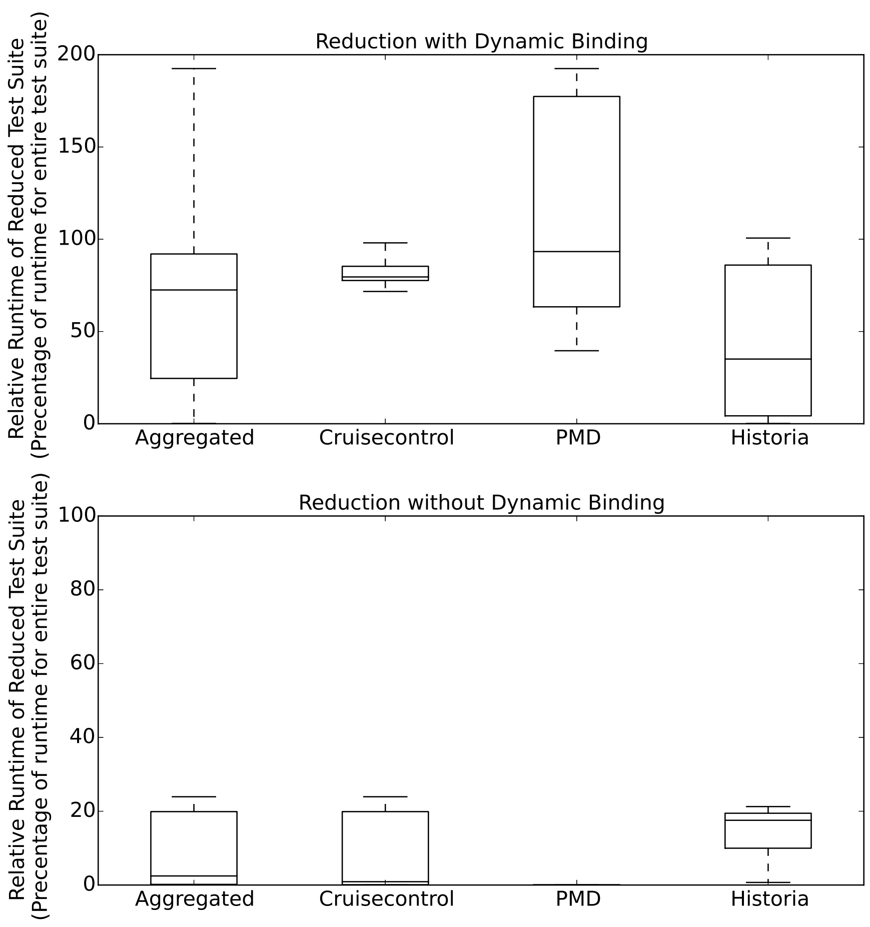

Fig. 12: Time reduction in the *Failing Test Coverage Analysis.*

PMD: Without dynamic binding, there was only one of the seven cases in which a reduced test suite was calculated, but for that case the $Reduction_{Time}$ was 0.02%. When taking dynamic binding into account, there were three out of the seven cases in which the $Reduction_{Time}$ was larger than 100% (177%, 178% and 192%) which means that in those three cases it actually took longer (almost twice as long) to calculate and run the reduced test suite as it would have been to run the full test suite.

Historia: Results of the reduction with dynamic binding are again very much spread with a maximum $Reduction_{Time}$ of 100.63% (a case where almost the full test suite was selected) to a minimum $Reduction_{Time}$ of 0.14%. On average the $Reduction_{Time}$ is 43% (2.3 times faster). In the case without dynamic binding the average $Reduction_{Time}$ is 7% (14 times faster).

What we can conclude from this is that when we do not take dynamic binding into account we achieve better reductions in time than when we do. Additionally the calculation of the reduced test suites is much faster.

6 GOAL - Discussion of Tradeoffs

In this section we summarise the results and look at the tradeoffs involved for taking dynamic binding into account, which was the goal of the paper:

> *GOAL – To investigate the tradeoffs between two variants of a test selection heuristic, determining the minimal suite of tests a developer needs to re-execute in order to verify whether a software system still behaves as expected.*

In the previous sections we have already highlighted results with and without dynamic binding, but here we will focus on highlighting the advantages and disadvantages of using the one over the other. Table 9 lists the advantages and disadvantages of each. We see that in fact both approaches are polar opposites in the criteria examined (what is good in the one, is bad in the other).

WITH Dynamic Binding		WITHOUT Dynamic Binding	
(-)	Slow calculation	(+)	Fast calculation
(+)	Can calculate a reduced set of tests for most classes	(-)	Unable to calculate reduced set of tests for many classes
(-)	Often bad reduction (sometimes nearly no reduction)	(+)	Very good reduction (mostly handful of tests)
(+)	Most failures covered by the reduced set of tests	(-)	Often failures missed

Table 9: Tradeoff when taking dynamic binding into account.

When we look at the calculation time we can see that this takes a lot longer when taking dynamic binding into account. This is of course due to the fact that there will be a lot more links between invocations and methods that an invocation could possibly be invoking. Whereas in the approach without dynamic binding, a one-to-one mapping between invocation and method is assumed. Table 8 showed that indeed the number of dependencies distilled from the repositories when taking dynamic binding into account was much higher. This was especially the case in PMD, where there was a difference of an order of magnitude due to its use of the Visitor pattern. As a result in some cases the time needed to calculate and run a reduced set of tests would exceed the runtime of the full test suite.

Looking at the reduction, we find that by not taking dynamic binding into account, we have many classes for which we are unable to calculate a reduced set of tests. Since the test selection heuristic is based on tracing the call graph from test cases to methods in the production code, we can not guarantee that we can find a subset of tests for each class. This is most likely the case with a high degree of dynamic binding. Since those are method invocations that can not be traced statically. Hence, when we take dynamic binding into account the number of classes for which we can calculate a reduced set of tests is much higher.

Yet for those classes that we are able to calculate a reduced set of tests without dynamic binding, we almost always have a very good reduction with in most cases just a handful of tests selected. Taking dynamic binding into account however will off course have its effect on the number of tests found for the reduced set of tests. Often it leads to very large sets of tests, sometimes even nearly no reduction.

Again the case of PMD provided some interesting observations, where some of the relevant tests were missed for other reasons than dynamic binding. On the one hand some tests in PMD used a third party library through which the production code was tested indirectly. On the other hand the test suit itself used a lot of inheritance (more so than the other two cases) which our heuristic currently does not support. Therefore a test class that indirectly tested production code through a test in its superclass would not be selected.

For the fault detection ability of the reduced sets of tests, we find that when we take dynamic binding into account most failures are covered. And without dynamic binding we do occasionally miss some relevant tests in our selection.

In order to make the trade-off we suggest that software teams should assess the degree of dynamic binding into the project. Most likely, the software engineers involved in the project will know this at the start. They can rely on the metrics DB_c and $DB_{project}$ to create a box-plot like in Figure 6. Moreover they should keep in mind that currently our heuristic only works for tests that directly invoke production code and not indirectly through third party libraries or through inheritance in the test class hierarchy.

From the point of view of the test suite reduction the introduction of dynamic binding increases the size of the reduced set of tests as well as the runtime for its calculation. This is a normal tradeoff we have to accept if we increase the number of potential relevant tests of our set. A software engineer may rely on the metrics DB_c and $DB_{project}$ to assess the use of dynamic binding in the project, and based on that decide which variant to use.

7 Threats to Validity

We now identify factors that may jeopardize the validity of our results and the actions we took to reduce or alleviate the risk. Consistent with the guidelines for case studies research (Runeson and Höst, 2009; Yin, 2002) we organise them into four categories.

Construct validity – do we measure what was intended We evaluated the reduced set of tests with three criteria, the size of the reduction, the runtime of the calculation and its return on investment, and the fault detection ability of the reduced set of tests measured through (i) the number of mutants killed and (ii) the number of actual failing tests found. There are however other criteria that can be used to evaluate a test selection heuristic, *e.g.*, test coverage, inclusion.

Mutation coverage is not an exact metric for the quality of a set of tests, it can only be used to gauge the quality. The results could be influenced by the use of other mutation operators (for instance, mutation operators specifically designed for Object-Oriented systems). However a higher number of mutation operators doesn't necessarily lead to a different mutation coverage. Indeed Offut*et al.* have determined that only a selection of few mutation operators is enough to produce the same coverage as all essential mutation operators (Offutt et al, 1996). Still it will be interesting to further investigate test selection using Object-Oriented mutation operators.

Another possible issue in the mutation analysis, is the possibility of equivalent mutants (*i.e.*, mutations that do not change the observable (testable) behaviour of the code). On the one hand PIT is designed to minimise the number of equivalent mutants generated. On the other hand if equivalent mutants are introduced our analysis would not suffer from this, as the same equivalent mutants would be introduced in both the mutation analysis with the reduced set of tests and the one with the full test suite. Therefore the mutation coverages can be reliably compared.

In the *Failing Test Coverage Analysis*, when looking for revisions with failing tests, we might miss certain revisions that contain failing tests. However it was never our intent to find all instances of failing tests that occurred in the system's evolution.

Internal validity – are there unknown factors which might affect outcome of the analyses The change model currently does not include constructor invocations, which leads to relevant tests being omitted erroneously. Additionally we were unable to deal with class hierarchies in the test code itself. In future work we will evaluate whether incorporating these constructs in the change model improves the results. Another problem that arose was, when the tests use third party libraries (*e.g.*, to test Ant tasks) the production code is not directly invoked from the tests but rather through the third party library. When doing the *Failing Test Coverage Analysis* we ignored tests that failed due to non-code changes (*e.g.*, changes to configurations in xml files). In future work it could be interesting to see whether we could incorporate changes to non source-code files in the change model and do test selection for those files as well.

Lastly our approach for dealing with dynamic binding is based on a simple approach of linking invocations to method declarations merely using the identifier. This approach can still be improved by taking into account the full method signature and the inheritance tree and by ignoring common java methods like `toString()`, `equals`, *etc.* It would be interesting to see in future work how these improvements influence the fault detection ability and the reduction of our heuristic.

External validity – to what extent is it possible to generalise the findings In this study we investigated three cases: Cruisecontrol and PMD as open source cases and Historia as an industrial case. We chose them to be sufficiently different, yet, with only three data-points, we cannot claim that our results generalise to other systems. Moreover all three cases come with a full test suite that achieves relatively good code coverage (see Table 3) yet we cannot claim that the use of a different test suite on the same program achieves the same results, which is an issue that is interesting to further investigate.

Reliability – is the result dependent on the tools In this paper we relied on tools of our own making as well as some external tools. Our ChEOPSJ tool is implemented as an Eclipse plugin and relies on Eclipse's internal java model; it also uses ChangeDistiller, both of which can be considered to be reliable external tools. The *Mutation Coverage Analysis* is performed using an external tool PIT, which is still actively being developed and improved, but which can be considered reliable.

8 Related Work

Regression Test Selection. This is a problem that has been investigated intensely over the last decade as demonstrated in the systematic literature review conducted by Engström et al (2010, 2008). It is an interesting problem from a practical point of view because it results in significant savings on the time to execute the regression test suite, hence lessens the pressure right before a software release (Binder, 1999). From a research point of view it is equally interesting as it results in interesting tradeoffs: the costs of selecting and executing test cases versus the need to achieve sufficient detection ability. Two studies in particular inspired the experimental set-up in this paper, as they compared different regression test selection techniques using a predefined set of criteria. Mansour *et al.* investigated algorithms such as simulated annealing, reduction, slicing, data-flow and firewall and compared them using criteria like number of selected test cases, execution time, precision, inclusiveness, preprocessing requirements, type of maintenance, level of testing, and type of approach (Mansour et al, 2001). Graves *et al.* compared a representative algorithm for five categories of techniques: minimisation, data-flow, safe, random and retest all using criteria like the test size reduction and fault detection effectiveness (Graves et al, 2001).

The variants of our test selection heuristic (with and without dynamic binding) can be classified in Graves' categories as follows. The variant without dynamic binding can be considered a minimisation technique, and the variant with dynamic binding can be considered a safe technique. Our approach differs from the ones used in their studies, as we are basing our selection technique on fine grained method level changes and we evaluate the approach in the context of developer tests. Moreover our heuristics are a static analysis of the source code, where many of the existing test selection (or test prioritisation) techniques rely on dynamic information. We do however come to the same conclusions in that the technique without dynamic binding (minimization) produces the smallest and least effective test suites whereas the approach with dynamic binding (safe) capture most test failures, but in several instances it cannot reduce the test suites at all.

Developer Tests. However, with the advent of agile processes and their emphasis on test-driven development (Beck, 2002) and continuous integration (Fowler, 2006), the nature of the test selection problem has changed significantly. In particular the line between unit/integration and regression testing is blurred; some authors explicitly use the term "developer tests" to refer to this grey zone (Meszaros, 2006). This has an impact on the following characteristics.

Process: Regression testing is traditionally a separate activity scheduled after unit and integration testing but before acceptance testing. With developer tests, it is an activity tightly interwoven with the build- and release process.

Automation: While automation has always been a key enabler for efficient regression testing some degree of manual scenario testing is often tolerated. With developer tests, "*self testing code*" is a necessary prerequisite.

Coverage: Regression tests are designed to maximise the chance of exposing regression faults; hence mainly use black-box coverage criteria. With developer tests, the coverage criteria depend on the focus of a particular test case, mixing black-box and white-box criteria.

These three dimensions together illustrate why it is worthwhile to revisit the test selection problem in the context of developer tests, yet that the criteria used to assess the solution should be interpreted differently. The process dimension implies that developer tests run more frequently, hence that the test size reduction (as an indicator for speed) is a highly relevant criterion. Nevertheless, the fully automated tests imply that a safe selection is not required: we can afford to miss a few relevant tests as long as the complete test suite is executed regularly serving as a safety net. Finally, the mixture of black- and white-box coverage criteria, implies that we should look beyond traditional coverage metrics (branch coverage, statement coverage, *etc.*) to assess the fault detection effectiveness, for instance, by comparing the number of mutants killed (Andrews et al, 2005).

Test Traceability. Given that the nature of the test selection problem has changed significantly, some authors have investigated heuristics to recover test-to-code traceability links. In earlier research, we exploited naming conventions, fixture element types, the static call graph, last call before assert, lexical analysis and co-evolution (Van Rompaey and Demeyer, 2009). Qusef *et al.* compared to these heuristics with their tool SCOTCH, which exploits dynamic slicing and conceptual coupling (Qusef et al, 2011). These heuristics work surprisingly well, however not in all cases. Weijers for instance, has shown that naming conventions are not reliable because some developer tests serve more like integration tests, testing multiple methods of multiple classes (Weijers, 2012).

Test Case Prioritisation. Some studies have expanded on the Regression Test Selection studies. Since using the safe regression test selection techniques often leeds to nearly no reduction at all these studies investigate the prioritisation of tests. On top of selecting a relevant subset of tests, they also put an ordering on them with the test most likely to reveal faults ranked highest. The way the tests are ranked can then be based on coverage criteria (*i.e.*, prioritization in terms of the number of statements, basic blocks, or methods test cases cover) (Catal and Mishra, 2013).

Integration with the IDE. The goal of developer testing is to provide rapid feedback to the individual developer, hence tight integration with the Integrated Development Environment (IDE) is critically important. Saff and Ernst have proposed *continuous testing* (Saff and Ernst, 2004), which, similarly to background compilation in the IDE, enables ultra-short feedback cycles. However, Saff and Ernst also report that in order to make continuous testing feasible, test selection techniques should be incorporated. Hurdugaci and Zaidman have developed TestNForce, a Visual Studio plug-in that links changes in production code to the tests that cover the changed pieces of production code (Hurdugaci and Zaidman, 2012). As the TestNForce approach relies on coverage, a coverage baseline is necessary, which is unavailable in the case of newly developed code as the tests have not been executed to cover that code. Our approach does not suffer from this weakness, yet it is potentially also less fine-grained. Ideas like these are making the transition from the state-of-the-art towards the state-of-the practice: Microsoft has incorporated the "Test Impact Analysis" feature in Visual Studio.

9 Conclusion

With the advent of agile processes and their emphasis on test-driven development and continuous integration, obtaining rapid feedback from executing a set of developer tests remains a challenge. Given the size and execution time of the complete test suite, it is often impractical to perform a "retest all" after each and every change. Hence, the problem is to select an appropriate subset of the complete test suite covering the most recent changes with sufficient detection ability. Inspired by previous research on *test selection*, we have investigated whether changes in the base-code can serve as a reliable indicator for identifying which developer tests need to be re-executed.

The goal of this paper was:

> *To investigate the tradeoffs between two variants of a test selection heuristic, determining the minimal set of tests a developer needs to re-execute in order to verify whether a software system still behaves as expected.*

This tradeoff can be expressed in terms of the results we obtained for the research questions:

RQ1 *What is the fault detection ability of the reduced test suites?* Looking at the mutation coverage we found that the results with and without dynamic binding were very similar, however this was only when looking at the classes for which a reduced test suite could be generated. The biggest influence of taking dynamic binding into account was that this resulted in the technique being able to generate a reduced test suite for more classes. Looking at the test failures in the history of the cases we find that with dynamic binding the fault detection ability was better than without dynamic binding.

RQ2 *What is the size reduction of the unit test suite in the face of a particular change operation?* When we do not take dynamic binding into account, the test selection heuristic reduces in 75% of the cases the subset to approximately 1% of the complete test suite; in most cases this corresponds to a single unit-test. With dynamic binding in the picture the reduction is worse, in 75% of the cases the subset was reduced to more than 60% of the full test suite. When we look at the return on investment in terms of runtime, we find that when not taking dynamic binding into account we make a significant improvement. However, taking dynamic binding into account sometimes results in the situation that the combination of calculating and running the reduced test suite takes longer than running the full test suite.

Our results show that, given a list of methods which changed since the latest commit, it is feasible to exploit control flow dependencies to select a subset of the entire test suite which is significantly smaller. The selected subset is not safe as it occasionally misses a few relevant tests, however it is *adequate*. Especially so, because we still expect that the complete test suite will be executed as part of the integration build anyway. When taking dynamic binding into account, we are capable of calculating a reduced test suite for more cases, and the reduced test suites will be safer. However this comes at the cost of a longer calculation time and larger reduced test suites.

As such, the adoption of dynamic binding in the test selection process should be provided as an optional feature for the developers to choose from depending on the degree of dynamic binding usage in the project.

Contributions. Over the course of this research, we have made the following contributions:

– We have implemented a tool prototype named ChEOPSJ serving as an experimental platform for conducting feasibility studies with first-class representation of changes in Java.
– We have demonstrated how this platform can be used to deduce which developer tests need to be re-executed for a given change.
– We applied the prototype on three cases —Cruisecontrol and PMD— as open source cases and Historia as an industrial case to assess the savings on reducing the test suite versus the ability to detect regression faults.
– We have demonstrated that it is feasible to use method-level changes to select a subset of a large test suite which is significantly smaller yet is adequate for identifying regression faults.
– We have compared two variants of our approach and analysed the tradeoffs involved.

Future work. There is a large body of knowledge on test selection techniques in the context of regression testing. Some of this work will have to be re-examined against the changing context of developer tests. In particular, we aim to address the following questions.

Are more elaborate test selection algorithms worthwhile? In literature more elaborate test selection techniques based on data-flow and slicing are documented. Hence it is worthwhile to see whether these techniques achieve better results. It is also worthwhile to further improve and evaluate the heuristics presented here. They can be improved by eliminating some of the limitations of the tool (*e.g.*, extend the change model to support constructor invocations; or adapt the heuristic to take into account inheritance in the test code). Moreover it would be interesting to see the effect (if any) of evaluating the heuristics with Object-Oriented mutation operators.

What are acceptable thresholds for less surviving mutations? In this paper, we used this measure as indicator for the fault detection ability of the reduced test suite. We observed that for many changes the reduced test set is perfectly safe. However, in some cases the reduced test set does not expose faults and then the question becomes when this reduced test set is adequate. Acceptable thresholds still need to be defined probably on the basis of field studies with realistic projects.

What is the real significance of test selection in the context of developer tests? Will developers be more inclined to run their developer tests more frequently with test selection enabled? Will this result in fewer (regression) faults later in the life-cycle? We see it as a challenge to perform field studies with real project teams to get insight.

Acknowledgements We express our gratitude to the SEAL team in the University of Zürich, Switzerland for releasing Changedistiller in the public domain; our ChEOPSJ tool is partly based on this release.

We would also like to express are gratitude to Yves Vandewoude at Qmino, for allowing us to conduct these analyses on the Historia codebase.

This work has been sponsored by (i) the Interuniversity Attraction Poles Programme - Belgian State Belgian Science Policy, project MoVES; (ii) the Institute for the Promotion of Innovation through Science and Technology in Flanders (IWT-Vlaanderen) under project number 120028 entitled "Change-centric Quality Assurance (CHAQ)"; (iii) the NWO TestRoots project, the Netherlands.

Figures 6, 7, 8, 9, 10, 11 and 12 were created with matplotlib (Hunter, 2007).

References

Andrews JH, Briand LC, Labiche Y (2005) Is mutation an appropriate tool for testing experiments? In: Proc. Int'l Conf. on Software Engineering (ICSE), ACM, pp 402–411, DOI 10.1145/1062455.1062530

Basili VR, Caldiera G, Rombach HD (1994) The goal question metric approach. Encyclopedia of Software Engineering

Beck K (2002) Test Driven Development: By Example. Addison-Wesley

Beller M, Gousios G, Panichella A, Zaidman A (2015a) When, how and why developers (do not) test in their ides. In: Proceedings of the 10th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE), ACM, pp 179–190

Beller M, Gousios G, Zaidman A (2015b) How (much) do developers test? In: Proceedings of the 37th International Conference on Software Engineering (ICSE – volume 2), IEEE, pp 559–562

Bennett KH, Rajlich VT (2000) Software maintenance and evolution: A roadmap. In: Proceedings of the Conference on The Future of Software Engineering, ACM, New York, NY, USA, ICSE '00, pp 73–87, DOI 10.1145/336512.336534, URL `http://doi.acm.org/10.1145/336512.336534`

Binder R (1999) Testing Object-Oriented Systems: Models, Patterns, and Tools. Addison-Wesley

Catal C, Mishra D (2013) Test case prioritization: A systematic mapping study. Software Quality Control 21(3):445–478, DOI 10.1007/s11219-012-9181-z, URL `http://dx.doi.org/10.1007/s11219-012-9181-z`

Daniel B, Jagannath V, Dig D, Marinov D (2009) Reassert: Suggesting repairs for broken unit tests. In: Proc. of the Int'l Conference on Automated Software Engineering (ASE), IEEE CS, pp 433–444

Demeyer S, Tichelaar S, Steyaert P (1999) FAMIX 2.0 - the FAMOOS information exchange model. Tech. rep., University of Berne

Dösinger S, Mordinyi R, Biffl S (2012) Communicating continuous integration servers for increasing effectiveness of automated testing. In: Proceedings of the International Conference on Automated Software Engineering (ASE), pp 374–377

Ebraert P, Vallejos J, Costanza P, Paesschen EV, D'Hondt T (2007) Change-oriented software engineering. In: Proc. of the Int'l Conference on Dynamic Languages (ICDL), ACM, pp 3–24, DOI http://doi.acm.org/10.1145/1352678.1352680

Engström E, Skoglund M, Runeson P (2008) Empirical evaluations of regression test selection techniques: a systematic review. In: Proc. Int'l Symp. Empirical Softw. Engineering and Measurement (ESEM), ACM, pp 22–31, DOI 10.1145/1414004.1414011

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

Fluri B, Wuersch M, PInzger M, Gall H (2007) Change distilling: Tree differencing for fine-grained source code change extraction. IEEE Transactions on Software Engineering 33(11):725–743, DOI http://dx.doi.org/10.1109/TSE.2007.70731

Fowler M (2006) Continuous integration. Tech. rep., `http://www.martinfowler.com/`, `http://www.martinfowler.com/articles/continuousIntegration.html`

Garousi V, Varma T (2010) A replicated survey of software testing practices in the canadian province of alberta: What has changed from 2004 to 2009? Journal of Systems and Software 83(11):2251–2262

Graves TL, Harrold MJ, Kim JM, Porter A, Rothermel G (2001) An empirical study of regression test selection techniques. ACM Transactions on Software Engineering and Methodology 10(2):184–208

Hattori L, Lanza M (2010) Syde: A tool for collaborative software development. In: Proc. of the Int'l Conference on Software Engineering (ICSE), ACM, pp 235–238

Hunter JD (2007) Matplotlib: A 2d graphics environment. Computing In Science & Engineering 9(3):90–95

Hurdugaci V, Zaidman A (2012) Aiding software developers to maintain developer tests. In: Proc. European Conf. on Softw. Maintenance and Reengineering (CSMR), IEEE CS, pp 11–20

Mansour N, Bahsoon R, Baradhi G (2001) Empirical comparison of regression test selection algorithms. Journal of Systems and Software 57(1):79—90

McGregor J (2007) Test early, test often. Journal of Object Technology 6(4)

Meszaros G (2006) XUnit Test Patterns: Refactoring Test Code. Prentice Hall PTR

Offutt AJ, Lee A, Rothermel G, Untch RH, Zapf C (1996) An experimental determination of sufficient mutant operators. ACM Trans Softw Eng Methodol 5(2):99–118, DOI 10.1145/227607.227610, URL `http://doi.acm.org/10.1145/227607.227610`

Parsai A, Soetens QD, Murgia A, Demeyer S (2014) Considering polymorphism in change-based test suite reduction. In: Agile Methods. Large-Scale Development, Refactoring, Testing, and Estimation - XP 2014 International Workshops, Rome, Italy, May 26-30, 2014, Revised Selected Papers, pp 166–181, DOI 10.1007/978-3-319-14358-3_14, URL `http://dx.doi.org/10.1007/978-3-319-14358-3_14`

Qusef A, Bavota G, Oliveto R, De Lucia A, Binkley D (2011) Scotch: Test-to-code traceability using slicing and conceptual coupling. In: Proc. of the Int'l Conference on Software Maintenance (ICSM), IEEE CS, pp 63–72, DOI 10.1109/ICSM.2011.6080773

Robbes R, Lanza M (2007) A change-based approach to software evolution. Electronic Notes in Theoretical Computer Science 166:93–109, DOI http://dx.doi.org/10.1016/j.entcs.2006.06.015

Robbes R, Lanza M (2008) Spyware: A change-aware development toolset. In: Proc. of the Int'l Conference in Software Engineering (ICSE), ACM Press, pp 847–850

Rothermel G, Untch R, Chu C, Harrold M (2001) Prioritizing test cases for regression testing. IEEE Transactions on Software Engineering 27(10):929–948, DOI 10.1109/32.962562

Runeson P (2006) A survey of unit testing practices. IEEE Software 23(4):22–29

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

Saff D, Ernst MD (2004) An experimental evaluation of continuous testing during development. In: Proc. Int'l Symp. Softw. Testing and Analysis (ISSTA), ACM, pp 76–85

Soetens QD, Demeyer S (2012) ChEOPSJ: Change-based test optimization. In: Proc. of the European Conference on Software Maintenance and Reengineering (CSMR), IEEE CS, pp 535–538, DOI http://doi.ieeecomputersociety.org/10.1109/CSMR.2012.70

Soetens QD, Demeyer S, Zaidman A (2013) Change-based test selection in the presence of developer tests. In: Proc. Conf. Softw. Maintenance and Reengineering (CSMR), pp 101–110

Tillmann N, Schulte W (2006) Unit tests reloaded: Parameterized unit testing with symbolic execution. IEEE Software 23(4)

Van Rompaey B, Demeyer S (2009) Establishing traceability links between unit test cases and units under test. In: Proc. of the Conference on Software Maintenance and Reengineering (CSMR), IEEE CS, pp 209–218, DOI http://doi.ieeecomputersociety.org/10.1109/CSMR.2009.39

Venolia G, DeLine R, LaToza T (2005) Software development at microsoft observed. Tech. rep., Microsoft Research, `http://research.microsoft.com/pubs/70227/tr-2005-140.pdf`

Weijers J (2012) Extending project lombok to improve junit tests. Master's thesis, Delft University of Technology, `http://resolver.tudelft.nl/uuid:1736d513-e69f-4101-8995-4597c2a4df50`

Yin RK (2002) Case Study Research: Design and Methods, 3 edition. Sage Publications

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

Yoo S, Nilsson R, Harman M (2011) Faster fault finding at Google using multi objective regression test optimisation. In: 8^{th} European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE '11), Szeged, Hungary

Zaidman A, Van Rompaey B, van Deursen A, Demeyer S (2011) Studying the co-evolution of production and test code in open source and industrial developer test processes through repository mining. Empirical Software Engineering 16(3):325–364