Change-Based Test Selection in the Presence of Developer Tests

Quinten David Soetens
University of Antwerp
Antwerp, Belgium
quinten.soetens@ua.ac.be

Serge Demeyer
University of Antwerp
Antwerp, Belgium
3 serge.demeyer@ua.ac.be

Andy Zaidman
Delft University of Technology
Delft, The Netherlands
a.e.zaidman@tudelft.nl

***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 are in order. In this paper we investigate how method-level changes in the base-code can serve as a reliable indicator for identifying which tests need to be rerun. We validate the approach on two cases — PMD and CruiseControl — using mutation testing as a means to compare the selected subset against a "*retest all*" approach. Our results show that we are able to reach a sizable reduction of the complete test suite, yet with a comparable number of mutants killed by the reduced test suite.**

I. Introduction

With the advent of agile processes and their emphasis on test-driven development, developer testing has been gaining popularity [1]. Developer tests are codified unit or integration tests written by developers, aiming to verify quickly whether changes to the system break previous functionality [2]. 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 reports that 79% of developers use unit tests [3]. Others report that the code for developer tests is sometimes larger than the application code under test [4], [5], [6]. 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 [7], we are particularly concerned with the observation of Runeson, who reports that some unit test suites take hours to run [8].

Our concern stems from the observation that when developer tests take too long to run, developers are less inclined to run them after each and every change. This is a genuine concern, as confirmed in previous case studies, where we witnessed that not all software projects uphold graceful co-evolution between production code and test code [5]. This effectively means that the software is vulnerable for extended periods of time as the production code evolves, but the test code does not follow (immediately). In this context, Moonen et al. have shown that even while refactorings are behaviour preserving they potentially invalidate tests [9]. In the same vein, Elbaum et al. concluded that even minor changes in production code can significantly affect test coverage [10].

We deduce that with today's agile development practices, it remains a challenge to capitalize on developer tests in rapid feedback cycles. 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 behavior of modified software*" [11]. While test selection was studied intensely in the context of regression tests, it has seldom been studied in the area of developer tests. This leads us to the overall goal of our investigation:

> *How can method-level changes in the base-code serve as a reliable indicator for identifying which developer tests need to be rerun?*

Given that test selection has been an active field of research in the past, we address the main research question via previously established criteria (e.g., [12], [13]) and derive the following subsidiary research questions.

RQ1 *What is the size reduction of the unit test suite in the face of a particular change operation?* Our reasoning is that a "retest-all" possibly takes several hours, so reducing the test set will likely lead to more testing and quicker feedback cycles.

RQ2 *What is the precision and recall of our approach?* Given that we reduce the test set, we are interested in knowing the number of tests in the subset that were added unnecessarily, and the ones that were omitted erroneously. A perfectly safe selection technique has 100% recall (i.e. all of the relevant tests in the complete test suite were included in the subset) perhaps at the cost of precision (i.e., some irrelevant tests were also included). In the context of developer tests however, a safe selection technique is excessive because the complete test suite will likely be executed as part of the build anyway. Hence, we strive for an *adequate* subset [14].

RQ3 *How many mutants does the reduced test suite kill?* A mutant is a transformation of the original base code that introduces a defect; a test which reveals the

corresponding defect 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.

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 and captures them made in the main editor while the developer is programming [15]. We applied the prototype on two distinct cases —PMD and CruiseControl— to verify the feasibility of the approach and assess against the aforementioned criteria.

The remainder of this paper is structured as follows. First, we describe both the change model and the test selection algorithm and how all this is implemented in ChEOPSJ (Section II). Next, we discuss the experimental set-up in Section III. We proceed with an analysis of the results in terms of the trade-off between the costs of selecting and executing test cases versus the need to achieve sufficient detection ability (Section IV). We end the paper with a discussion of the results and the threats to validity (Section V), a summary of related work (Section VI), and wrap up with the conclusions (Section VII).

II. Change Reification in ChEOPSJ

Robbes and Lanza argue that 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 [16]. First-class change entities, modeled as objects in Robbes' and Lanza's approach, represent the behavior of the different kinds of changes required for a program (for example to add, remove, and modify classes) [17].

In recent years several researchers have built upon that idea and have created tools that analyze change objects. The approaches by Robbes et al. in the Spyware tool [18] and later by Hattori et al. in the Syde tool [19] 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 [17]. We chose to expand upon the approach by Ebreart et al., because we are particularly interested in analyzing 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 his model in Eclipse.

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 [20]. This is a model to which most class-based object oriented programming languages adhere and it contains entities representing, packages, classes, methods and attributes, as well as more fine grained entities such as method invocations, variable accesses and inheritance relationships.

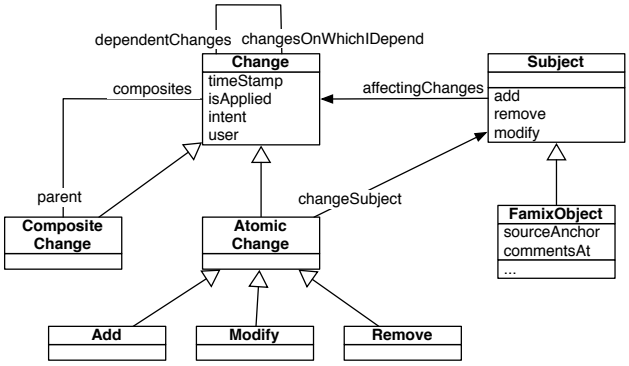

Figure 1: Model of the changes

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. A change object c_1 is said to depend on another change object c_2 if the application of c_1 without c_2 would violate the system invariants.

A. Test selection algorithm

We aim to use this dependency information for test selection purposes. In particular, in our test selection algorithm, 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 behavioral 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 [21], 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
    calledMethod = findMethodAddition(c.hierarchicalDependencies());
    invocations = calledMethod.invocationalDependees();
    foreach i in invocations do
        invokedBy = findMethodAddition(i.hierarchicalDependencies());
        foreach m in invokedBy do
            if m is a test then
                add m to relevantTests;
            else
                if m was not previously analyzed 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`. 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 that add a method that invokes the method that contains our selected change. We then iterate these method additions and check whether these changes added a test method. 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.

B. 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 plug-ins [15]. The general design is depicted in Figure 2.

At the center 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*.

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

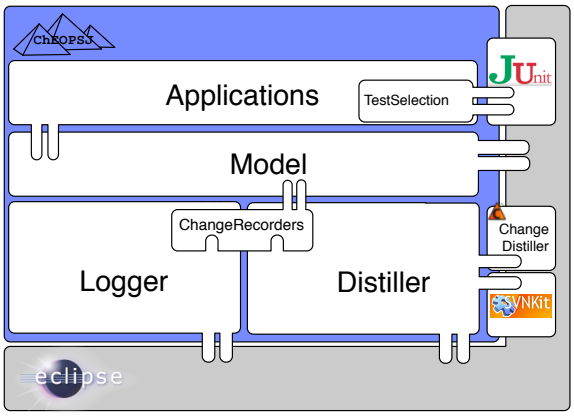

Figure 2: The layered design of ChEOPSJ

These two plugins are responsible for populating the change 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 `IElementChangedListener` 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 analyze real world cases, we also provide a Distiller which implements a change recovery technique. By using SVNKit it iterates through all versions stored in the SVN 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 Fluri et al.'s ChangeDistiller [22] to diff between the unmodified and the modified versions of the file and then translate the changes from ChangeDistiller to the changes in our model. This includes 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.

We can now build other applications on top of ChEOPSJ that use the information contained in the model. One of these

is our TestSelection plugin, that implements the test selection algorithm shown in Algorithm 1. The ChangeInspector view, provided by the ChEOPSJ Model plugin allows us to make a selection of changes for which we can then run the algorithm that finds tests that are relevant for those changes.

III. Experimental Setup

To address the subsidiary research questions in Section I (in particular RQ2 and RQ3) we first establish precise rules on how we measured the precision, recall and the mutants killed. 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 could replicate our investigation.

A. Dynamic Analysis with AspectJ

To measure the precision and recall of our test selection algorithm, we need to define a *baseline* of the relevant tests for a certain method-level change. To obtain this base-line we perform a dynamics analysis: we execute the original test suite and note for each class which tests actually invoke a method of this class. Via this baseline we can calculate precision and recall; in Figure 3 we show how the set of tests selected by ChEOPSJ relate to the tests selected by the dynamic analysis. The true positives (TP) are those tests that are in both the reduced set and the baseline. The tests that were only selected by ChEOPSJ are called the false positives (FP), whereas the tests only selected with the dynamic analysis are called the false negatives (FN). All tests that were neither selected by ChEOPSJ nor by the dynamic analysis are the true negatives (TN).

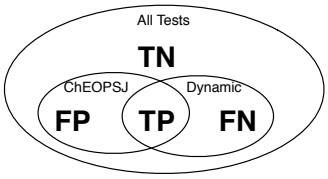

Figure 3: The basis for calculating precision and recall.

We can now define precision and recall as follows:

Precision: is the rate of true positives versus the number of tests in the reduced test suite. This says how many of the tests selected by ChEOPSJ were also selected by the Dynamic Analysis (i.e., how many of the selected tests are actually relevant?).

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

Recall: is the rate of true positives versus the number of tests in the set found with the dynamic analysis. This says how many of the tests selected with the dynamic analysis were also selected by ChEOPSJ (i.e., how many of the relevant tests were also selected?).

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

To do the dynamic analysis we wrote a simple aspect in AspectJ[2] More specifically we wrote an aspect that whenever a test method is executed, we note which test class this method belongs to. Additionally the aspect works upon each method execution within the control flow of the test method's execution. It will then keep track of the class in which this method was and add the corresponding test class to the relevant tests for this class.

B. Mutation Testing with PIT

A recurring issue with testing experiments is the lack of realistic cases containing documented faults. As a substitute, researchers often plant defects into a correct program by applying a so-called mutation operator or *mutator* [23]. These mutators are chosen based on a fault model and as such are a close approximation of typical defects 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.

PIT (http://pitest.org) is a tool that does byte code based mutation testing. It supports both Ant (http://ant.apache.org) and Maven (http://maven.apache.org) and can thus easily be integrated into the build process of many open source systems. We used the default PIT configuration, which uses seven kinds of mutators, which we briefly explain in Table I. We can run PIT using all tests or using only our reduced test suite and then compare the outcomes. In the one case we let PIT create mutations in the entire code base, in the other case we tell PIT to create mutations only on one particular class. For all changes in this class we can determine what the relevant tests are and can then see how many of the mutations are found when only running this subset of tests.

C. Case Selection

We selected two cases — PMD and Cruisecontrol — on which to run our evaluations. Cruisecontrol (http://

[2]http://www.eclipse.org/aspectj/

Mutator	Description
Conditionals Boundary	Replaces relational operators with their boundary counterpart (e.g. $<$ becomes $<=$, $>=$ becomes $>$, ...)
Negate Conditionals	Replaces all conditionals with their negated counterpart (e.g. $==$ becomes $!=$, $<$ becomes $>=$, ...)
Math	Replaces binary arithmetic operations from either integer or floating-point arithmetic with another operation (e.g. $+$ becomes $-$, $*$ becomes $/$, ...)
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, ...)
Void Method Call	Removes method calls to void methods.

Table I: Default mutators activated in PIT

cruisecontrol.sourceforge.net) is a framework that allows for creating a custom continuous integration process. PMD (http://pmd.sourceforge.net) is a source code analyzer to find a variety of common mistakes like unused variables, empty catch blocks unnecessary object creations, etc.

Our main research question is essentially a feasibility study, hence we chose the cases mainly on the basis of convenience. Both of these are open-source Java projects, which can be accessed through a Subversion repository. They both come with a `.project` file for Eclipse, making it easy to run our Eclipse plugin to instantiate the changes necessary for running our test selection algorithm. They can also be built via the command line through either Ant or Maven, which allows us to easily add the commands to the build files to run the PIT mutation tester.

The sizes of these projects in terms of number of lines of code and number of classes for both the source code and the test code as well as the revision that was analyzed are shown in table II.

IV. Analysis of Results

Given the description of the conceptual tool prototype (Section II) and the motivation for the selected cases (Section III-C), this section analyzes the results of applying the test selection algorithm (Algorithm 1) on the two cases. This analysis is performed on the basis of the criteria listed in the introduction: the test size reduction (Section IV-A), precision and recall (Section IV-B) and the mutants killed (Section IV-C).

A. Test Size Reduction

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. In most cases the reduced test suite consists of a single unit test. For Cruisecontrol this implies a reduction of 0.34% (1 out of 295 test classes); for PMD this corresponds to 1 out of 215 classes or 0.47%.

The relative reductions in size for the two cases under study are shown in the box plot in Figure 4. We can see that for Cruisecontrol the minimum and the median are the same, which means that 50% of the test suites are reduced to 0.34% — a single test class. The next 25% (between the median and the 3rd quartile) of reduced test suites were reduced to sizes between 0.34% and 1.0%. The largest reductions vary between 1.0% and 7.5%. For PMD we observe similar reductions. Half of the reduced test suites were reduced to

Project	Version analyzed	Src KLOC	Src NOC	Test KLOC	Test NOC	Build Process
Cruisecontrol	rev. 4601	26.5	376	24.5	295	ant
PMD	rev. 7706	46	804	9	215	maven

Table II: Number of 1000 Lines of Code (KLOC) and Number of Classes (NOC) for both source code and test code (measured with InFusion 7.2.7).

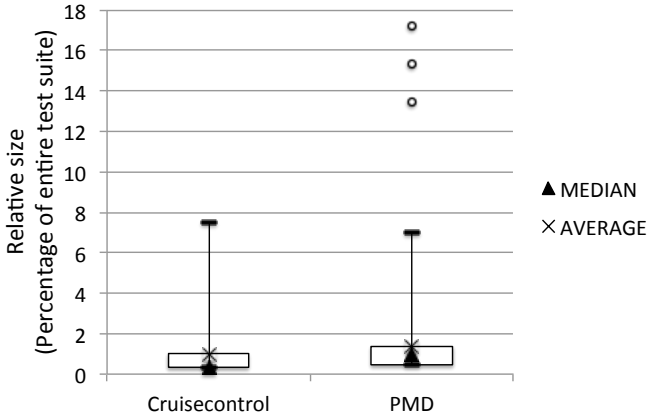

Figure 4: Relative Test Size Reduction.

between 0.47% (the minimum) and 0.93% (the median); a quarter was reduced to between 0.93% (the median) and 1.4% (the upper quartile). The most reduced test suites have relative sizes of between 1.4% and 7.0% of the total test suite. In PMD we also found three outliers where the test suite was reduced to 13.5% (29 tests out of 215), 15.3% (33 tests out of 215) and 17.2% (37 tests out of 215).

Since for both PMD and Cruisecontrol, the third quartile lies around 1%, we deduce that in 75% of the cases the test selection algorithm reduces the entire test suite by 99%.

B. Precision and Recall

To count the number of tests in the subset that were added unnecessarily, and the ones that were omitted erroneously we compared the subset identified with the test selection algorithm against a baseline test set identified with a dynamic analysis. As such we gathered results for 236 test sets for Cruisecontrol and 139 test sets for PMD. These results are shown in the box plots in Figure 5

Half of the test sets in Cruisecontrol get an ideal precision of 1, as both the median and the maximum are located at 1. The next 25% of the test sets have a precision between 0.84 (the lower quartile) and 1 (the median) and the final 25% have a precision that varies between 0 (the minimum) and 0.84 (the lower quartile). Overall we have an average precision of 0.87, which implies that on average, 13% of the test are added unnecessarily by our test selection algorithm.

The recall performs a bit worse. Again, the first half of the reduced test sets obtain a perfect recall of 1. However 25% of the reduced test sets score between 0.5 (the lower quartile) and 1 (the median); the rest of the test sets get a recall between 0 and 0.5. On average we have a recall value of 0.77, which implies that on average 23% of the relevant tests were omitted erroneously by our algorithm.

In the case of PMD, the precision and recall values are significantly lower. Again we see that in half of the test sets we get an ideal precision of 1. A quarter of the test sets have precision values between 0.67 (the lower quartile) and 1 (the median) and the rest of the test sets have a precision lower

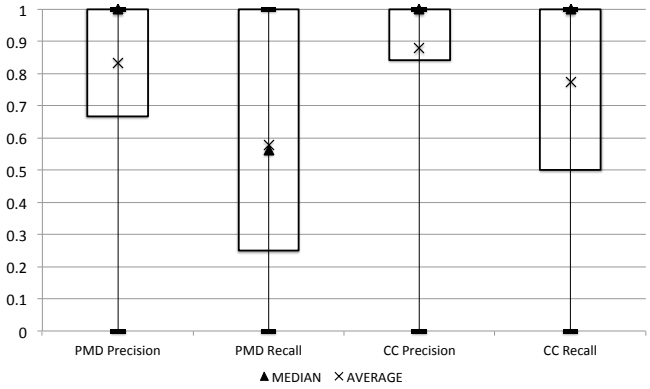

Figure 5: Precision and recall of the test selection.

than 0.67 with 0 being the smallest precision. On average the precision is 0.83, which is comparable to the precision of Cruisecontrol. The recall on the other hand is significantly worse. Only a quarter of the test sets have an ideal recall of 1; the second quarter of test sets have recall of between 0.56 (the median) and 1 (the upper quartile); the final half of the test sets have a recall lower than 0.56 (the median). On average there is a recall value of 0.58, which implies that almost half of the relevant tests were omitted erroneously by our test selection algorithm.

C. Mutants Killed

To assess the fault detection ability of the reduced test suite, we compare the number of mutants killed against the ones that are killed by the complete test suite. When the reduced test set kills the same number of mutants, it is as good at exposing defects as the retest-all approach. When more mutants survive, the detection ability of the reduced test suite is worse, however the confidence one might have in the results depends on the percentage of how many more mutants survive. Table III shows the initial results: for Cruisecontrol in 88% of the investigated test suites (155 out of 176) both test suites kill the same number of mutants. In the case of PMD, this number drops to 50% (71 out of 141). Consequently, for a number of reduced test sets, fewer mutants were killed: 12% (21 of 176) for CruiseControl; 50% (71 out of 141) for PMD.

Despite fewer mutants being killed, the confidence one might have in the reduced test set is still acceptable as may be deduced from Figures 6 and 7. The horizontal axis shows how many mutants were introduced in total in the inspected class; the vertical axis shows how many more mutants survived relative to the total number of mutants introduced. For instance point A in Figure 6, indicates that

Project	Equal Kills	Less Kills
Cruisecontrol	88% (155 out of 176)	12% (21 out of 176)
PMD	50% (71 out of 141)	50% (70 out of 141)

Table III: Quality of reduced test sets.

A's reduced test suite killed 38.8% (out of 36 mutants) less mutants than the entire test suite. Similary, point B shows that 31% of 138 mutants survived the reduced test suite, yet did not survive the retest-all.

On each graph we also show two lines: the bottom line is the *three-mutant line*, which indicates what the percentage is of three mutants out of the total number of mutants (e.g. 3 out of 3 is 100%, 3 out of 6 is 50% and 3 out of 180 is 1.667%). Any point below this line represents a case where the reduced test suite killed only one or two less mutants. In these cases we still have a comparable number of mutants.

The second line is the *ten-mutant line*, which indicates the percentages of ten mutants in the total number of mutants. All points above this line indicate that in those cases the reduced test suite killed more than ten mutants less. In the case of Cruisecontrol there are only two such points: A and B, which respectively have 14 out of 36 and 43 out of 138 more surviving mutants.

These lines also put the higher percentages in perspective. For instance point C in Figure 6 misses all (100%) of the mutants that were killed with the retest-all; however out of a total of only two introduced mutants.

In the case of PMD many more classes lie above the ten-mutant line. This indicates a lot more tests were missing from the reduced test suites, which was also evident from the lower recall values. Nonetheless many points also lie below the *three-mutant line* representing cases were the reduced test suites killed only one or two mutants less, which is still a comparable result to the number of mutants killed by the entire test suite.

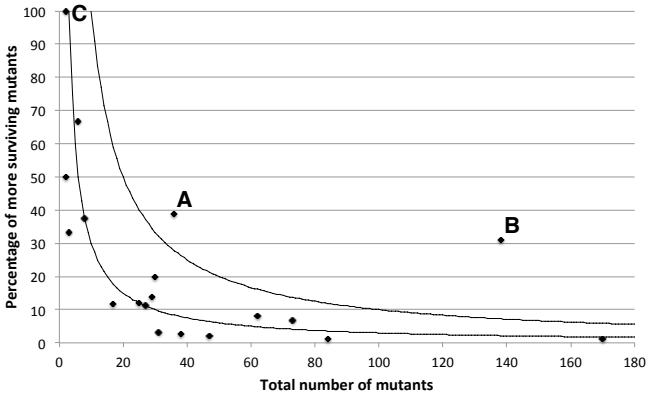

Figure 6: Percentages of survived mutants for Cruisecontrol

V. Discussion

In this section we discuss the results shown in the previous section and revisit the research questions introduced in Section I. We also indicate threats to validity.

A. Revisiting the research questions

RQ1 *What is the size reduction of the unit test suite in the face of a particular change operation?* In Section IV-A,

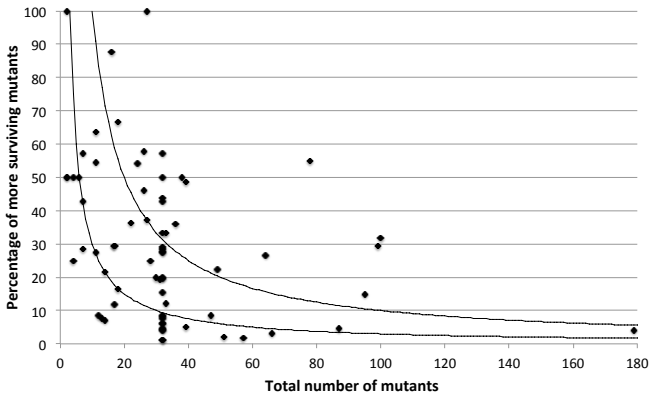

Figure 7: Percentages of survived mutants for PMD

we have shown that according to our test selection algorithm, in most cases more than 99% of the entire test suite is discarded. Thus, for most changes only a handful of tests need to be rerun. We identified three outliers in the case of PMD. Two of these outliers can be explained due to the fact that the selected class is actually an enum with hundreds of references in the rest of the project. The third outlier is a class named `AbstractNode`, which serves as the base class for 199 concrete node classes.

RQ2 *What is the precision and recall of our approach?* We have shown in Section IV-B that we reach high precision scores: on average 0.87 for Cruisecontrol and 0.83 for PMD. This implies that only 13% and 17% of the selected tests are irrelevant, when compared against the baseline obtained via dynamic analysis. The recall values are a bit lower for Cruisecontrol (with an average of 0.77) and much lower for PMD (with an average of 0.58). This means that for Cruisecontrol 23% of the tests that are relevant according to the dynamic analysis were found to be irrelevant with our change-based test selection. For PMD on the other hand almost half of the relevant tests are missing from our reduced test sets. Whether this is adequate remains an open question: since the build process involves a retest all anyway, there is a safety net in place which catches those defects not detected by the reduced subset. For some systems, this safety net will be used half of the time.

RQ3 *How many mutants does the reduced test suite kill?* In Section IV-C we have shown that in 88% of the reduced test suites for Cruisecontrol and 50% of the reduced test suites for PMD kill the same amount of mutants as the complete test suite.The other 12% and 50% however are cases where the reduced test suites killed fewer mutants than the whole test suite. Although we can still put this in perspective: these reduced test suites kill on average 23.8% (for Cruisecontrol) and 29.8% (for PMD) less mutants. This image is a bit warped as the total number of mutants introduced varies between 2 and 179. There are a few cases where we miss a high percentage of a low amount of mutants (e.g. 100% of 2 mutants) which increases the average. To counter this we can look at a weighted average of the percentage fewer mutants killed, using the total number of mutants as weight. We then find for Cruisecontrol and PMD, a weighted average of respectively 12.3% and 23.9 % less mutants killed.

The fact that some reduced test sets kill fewer mutants means that in we missed some relevant test cases. This is also confirmed by the lower recall values. In fact when we look at the cases where less mutants were killed we can also find a low value for the recall of those reduced test sets. In the other direction this is not necessarily true. A low recall value does not indicate that there will be a lower number of mutants killed as several tests might be able to kill the same mutants.

The fact that for some classes we kill less mutants and have a lower recall can be explained by a number of factors. A general observation is that this can be explained due to the fact that some language constructions can not yet be modeled in our change model. These missing constructs include invocations of class constructors, as well as invocations of polymorphic methods, abstract methods or methods declared in interfaces.

A common example in the PMD case is the invocation of a visit method in the `JavaParserVisitor` interface. This interface has many implementations, one example is the `DataFlowFacade` class, which invokes the following method in the `ASTCompilationUnit`.

```
@Override
public Object jjtAccept(JavaParserVisitor visitor, ...) {
    return visitor.visit(this, ...);
}
```

It is clear that this method is invoked from the test class, but there is a link missing between the method declared in the interface and the actual implementation in the `DataFlowFacade` class. Any method invoked from the actual implementation will be missing a relevant test.

Similar examples can be found for invocations of abstract methods and methods that overload a method declared in a superclass.

Another issue with finding relevant tests in PMD is that this project uses polymorphism in its test classes as well. As such we can find the class `AbstractRendererTst` to be a relevant test for a number of classes, but it will be harder to find any of its subclasses to be a relevant test.

B. 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 (see [24], [25]) we organize them into four categories.

Construct validity – do we measure what was intended: We evaluated the reduced test suites with three criteria, the size of the reduction, the accuracy of the tool via precision and recall and the fault detection ability of the reduced test suites measured through the number of mutants killed. There are however other criteria that can be used to evaluate test selection algorithm, e.g., test coverage, inclusion.

Internal validity – are there unknown factors which might affect outcome of the experiment: The change model currently does not include polymorphism or constructor invocations, which leads to relevant tests being omitted erroneously. We can improve the results by incorporating these language constructs in the change model. A method invocation will then not only be invocationally dependent on the method that is being invoked but also on every polymorphic variation of this method. This will improve recall, at the expense of precision.

External validity – to what extent is it possible to generalize the findings: In this study we investigated two cases: Cruisecontrol and PMD. We chose them to be sufficiently different, yet, with only two datapoints, we cannot claim that our results generalise to other systems.

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. Our dynamic analysis is performed using a simple aspect that we wrote ourselves in AspectJ and which we thoroughly tested. The mutation experiment is performed using an external tool PIT, which is still actively being developed and improved, but which can be considered reliable.

VI. Related Work

Regression test selection is a problem that has been investigated intensely over the last decade as demonstrated in the systematic literature review conducted by Engström et al. [11], [26]. 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 [21]. From a research point of view it is equally interesting as it results in interesting trade-offs: 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, dataflow 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 [12]. Graves et al. compared a representative algorithm for four categories of techniques: minimization, dataflow, safe, random and retest all using criteria like the test size reduction and fault detection effectiveness [13].

Developer Tests. However, with the advent of agile processes and their emphasis on test-driven development [27] and continuous integration [28], 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 [2]. 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 maximize the chance of exposing regression bugs; 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. On the other hand, 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, ...) to assess the fault detection effectiveness, for instance by comparing the number of mutants killed [23].

Test selection heuristics. 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 [29]. Qusef et al. compared against these heuristics with their tool SCOTCH, which exploits dynamic slicing and conceptual coupling [30]. 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 [31].

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* [32], 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 [33]. 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.

VII. Conclusion

With the advent of agile processes and their emphasis on test-driven development and continuous integration, obtaining rapid feedback from executing a suite of developer tests remains a challenge. Given the size of the complete test suite, it is 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.

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 since the complete test suite will be executed as part of the integration build anyway. To assess the significance of the reduction and the detection ability of this subset, we turn to the subsidiary research questions:

RQ1 *What is the size reduction of the unit test suite in the face of a particular change operation?* If a developer changes a single method, the test selection algorithm reduces in 75% of the cases the subset to approximately 1% of the complete test suite; in most cases this corresponds with a single unit-test. There are a few outliers however, yet in the worst-case the reduction is still 17% corresponding to 37 test cases out of 215.

RQ2 *What is the precision and recall of our approach?* For half of the method-level changes, the test selection algorithm identifies the exact subset of tests covering those changes. For the remaining half, the algorithm selects few irrelevant test cases, however misses a lot of the relevant ones. This is quite dependent on the implementation however: methods exhibiting polymorphism have multiple possible targets and then the test selection becomes less adequate. Parts of the PMD design relies heavily on template methods and in those cases the test selection missed up to 35 out of 42 test cases.

RQ3 *How many mutants does the reduced test suite kill?* Here as well, we observe that this is quite dependent on the implementation. In at least half of the cases (9 out of 10 in Cruisecontrol, 1 out of 2 in PMD), the selected subset kills exactly the same number of mutants. In the remaining cases, there are very few mutants who survive, typically 1 out of 5. There are a few outliers however (in the worst case 43 mutants survived out of 78), which again can be explained by the use of polymorphism.

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 when a given method has changed.
- We applied the prototype on two cases — PMD and CruiseControl — 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.

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 dataflow and slicing are documented. Hence it is worthwhile to see whether these techniques achieve better results.

What are acceptable thresholds for precision, recall and surviving mutations survived? In this paper, we used these measures as indicators 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 defects and then the question is 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 here.

ACKNOWLEDGMENTS

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.

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 RAAKPRO project EQuA (Early Quality Assurance in Software Production) of the Foundation Innovation Alliance, the Netherlands.

REFERENCES

[1] V. Garousi and T. Varma, "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*, vol. 83, no. 11, pp. 2251–2262, 2010.

[2] G. Meszaros, *XUnit Test Patterns: Refactoring Test Code*. Prentice Hall PTR, 2006.

[3] G. Venolia, R. DeLine, and T. LaToza, "Software development at microsoft observed," Microsoft Research, Tech. Rep., 2005, http://research.microsoft.com/pubs/70227/tr-2005-140.pdf.

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

[5] A. Zaidman, B. Van Rompaey, A. van Deursen, and S. Demeyer, "Studying the co-evolution of production and test code in open source and industrial developer test processes through repository mining," *Empirical Software Engineering*, vol. 16, no. 3, pp. 325–364, 2011.

[6] N. Tillmann and W. Schulte, "Unit tests reloaded: Parameterized unit testing with symbolic execution," *IEEE Software*, vol. 23, no. 4, 2006.

[7] J. McGregor, "Test early, test often," *Journal of Object Technology*, vol. 6, no. 4, 2007.

[8] P. Runeson, "A survey of unit testing practices," *IEEE Software*, vol. 23, no. 4, pp. 22–29, 2006.

[9] L. Moonen, A. van Deursen, A. Zaidman, and M. Bruntink, "The interplay between software testing and software evolution," in *Software Evolution*, T. Mens and S. Demeyer, Eds. Springer, 2008, pp. 173–202.

[10] S. Elbaum, D. Gable, and G. Rothermel, "The impact of software evolution on code coverage information," in *Proc. of the Int'l Conference on Software Maintenance (ICSM)*. IEEE CS, 2001, pp. 170–179.

[11] E. Engström, P. Runeson, and M. Skoglund, "A systematic review on regression test selection techniques," *Journal Information and Software Technology*, vol. 52, no. 1, pp. 14–30, 2010.

[12] N. Mansour, R. Bahsoon, and G. Baradhi, "Empirical comparison of regression test selection algorithms," *Journal of Systems and Software*, vol. 57, no. 1, pp. 79—90, Apr. 2001.

[13] T. L. Graves, M. J. Harrold, J.-M. Kim, A. Porter, and G. Rothermel, "An empirical study of regression test selection techniques," *ACM Transactions on Software Engineering and Methodology*, vol. 10, no. 2, pp. 184–208, 2001.

[14] M. Pezze and M. Young, *Software Testing and Analysis: Process, Principles and Techniques*. Wiley, 2007.

[15] Q. D. Soetens and S. Demeyer, "ChEOPSJ: Change-based test optimization," in *Proc. of the European Conference on Software Maintenance and Reengineering (CSMR)*. IEEE CS, 2012, pp. 535–538.

[16] R. Robbes and M. Lanza, "A change-based approach to software evolution," *Electronic Notes in Theoretical Computer Science*, vol. 166, pp. 93–109, 2007.

[17] P. Ebraert, J. Vallejos, P. Costanza, E. V. Paesschen, and T. D'Hondt, "Change-oriented software engineering," in *Proc. of the Int'l Conference on Dynamic Languages (ICDL)*. ACM, 2007, pp. 3–24.

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

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

[20] S. Demeyer, S. Tichelaar, and P. Steyaert, "FAMIX 2.0 - the FAMOOS information exchange model," University of Berne, Tech. Rep., 1999.

[21] R. Binder, *Testing Object-Oriented Systems: Models, Patterns, and Tools*. Addison-Wesley, 1999.

[22] B. Fluri, M. Wuersch, M. PInzger, and H. Gall, "Change distilling: Tree differencing for fine-grained source code change extraction," *IEEE Transactions on Software Engineering*, vol. 33, no. 11, pp. 725–743, 2007.

[23] J. H. Andrews, L. C. Briand, and Y. Labiche, "Is mutation an appropriate tool for testing experiments?" in *Proc. Int'l Conf. on Software Engineering (ICSE)*. ACM, 2005, pp. 402–411.

[24] P. Runeson and M. Höst, "Guidelines for conducting and reporting case study research in software engineering," *Empirical Softw. Engineering*, vol. 14, no. 2, pp. 131–164, 2009.

[25] R. K. Yin, *Case Study Research: Design and Methods, 3 edition*. Sage Publications, 2002.

[26] E. Engström, M. Skoglund, and P. Runeson, "Empirical evaluations of regression test selection techniques: a systematic review," in *Proc. Int'l Symp. Empirical Softw. Engineering and Measurement (ESEM)*. ACM, 2008, pp. 22–31.

[27] K. Beck, *Test Driven Development: By Example*. Addison-Wesley, 2002.

[28] M. Fowler, "Continuous integration," http://www.martinfowler.com/, Tech. Rep., May 2006, http://www.martinfowler.com/articles/continuousIntegration.html.

[29] B. Van Rompaey and S. Demeyer, "Establishing traceability links between unit test cases and units under test," in *Proc. of the Conference on Software Maintenance and Reengineering (CSMR)*. IEEE CS, 2009, pp. 209–218.

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

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

[32] D. Saff and M. D. Ernst, "An experimental evaluation of continuous testing during development," in *Proc. Int'l Symp. Softw. Testing and Analysis (ISSTA)*. ACM, 2004, pp. 76–85.

[33] V. Hurdugaci and A. Zaidman, "Aiding software developers to maintain developer tests," in *Proc. European Conf. on Softw. Maintenance and Reengineering (CSMR)*. IEEE CS, 2012, pp. 11–20.