METHODOLOGICAL

An approach for Test Impact Analysis on the Integration Level in Java programs

Muzammil Shahbaz

Thales Under Water Systems, United Kingdom
Correspondence
Email: muzammil.shahbaz@uk.thalesgroup.com

Summary
Test Impact Analysis is an approach to obtain a subset of tests that are impacted by the code changes. This approach is mostly applied to unit testing where the link between the code and its associated tests is easy to obtain. On the integration level, however, it is not straightforward to find such a link programatically, especially when the integration tests are held into separate repositories. We propose an approach for selecting integration tests based on the runtime analysis of code changes to reduce the test execution overhead. We provide a set of tools and framework that can be plugged into existing CI/CD pipelines. We have evaluated the approach on a range of open-source Java programs and found ≈50% reduction in tests on average, and above 80% in a few cases. We have also applied the approach to a large-scaled system currently in production and found similar results.

KEYWORDS:
Test Impact Analysis, Regression Test Selection, Continuous Integration, Continuous Testing

1 | INTRODUCTION

Continuous Integration/Delivery (CI/CD) is a modern way of building and releasing software in a shorter development lifecycle. Developers continuously implement small changes into their code and merge back to the main branch of the repository as often as possible and many times a day. These changes are validated by the automated system of building, testing, and deployment of software, which makes the integration easier and delivery quicker. Running automated integration tests is a vital phase of CI/CD that makes sure that the new changes to the code does not break the software or acceptance criteria whenever new merges are committed to the main branch. Moreover, they also serve as a regression test suite, verifying that no bugs are introduced into the existing behavior by the new changes.

Figure 1 shows a typical CI/CD pipeline that how a software component is moved across various stages in the development lifecycle before it is released. The pipeline is triggered when some code changes are committed into a repository of a version control system. The code goes to the build phase where it is compiled, followed by running the unit tests. Given there are no failures in the build, the component moves to the next phase where it is deployed in staging or a test server. Thereby, the integration or acceptance tests are checked out from the test repository and run on the component along with the other components in the system collectively. When all the integration tests are passed, the component is tagged or promoted to release.

1.1 | Motivation

As software matures with time, the integration test-suite tends to grow to the extent that running all tests takes a considerable amount of time. This is especially true when the software is delivered via iterative process, where the first release might contain a minimum functionality, but its codebase grows manifolds due to adding new features and fixing defects. This increases the rate of code changes, for instance, Google’s code
churn rate — a commit every second on average — produces 800K builds with 150 Million tests runs daily. Even with the company’s massive compute resources, it is not cost effective to test each code commit individually at this rate. In practice, developers delay their merges until tests are completed in the pipeline for each of the prior changes. This eventually slows down the whole pipeline process and diminishes the benefit of automation. Developers tackle this situation by either disabling some tests, or deferring them to the very end of the release cycle, which again results in lower productivity and quality.

1.2 Test Impact Analysis

The basic principle of Test Impact Analysis (TIA) is to determine the subset of tests that is impacted by the code change. Thus, running only the subset results into faster execution of the pipeline.

Finding the subset of impacted tests is easy on a component level where code and unit tests are kept in the same repository. Modern IDEs and tools for structural coverage can efficiently compute call graphs of unit tests either statically or dynamically. These call graphs can trace the tests back to the methods changed and then only those unit tests can be selected to run.

On the integration level, however, is a different story. The integrated tests cover the whole system and execute many components at once or in a particular order. There might not be a direct relation between an integration test and a method changed in one of those components. This is a typical case of microservices, where a monolithic application is replaced by a suite of small services that are built independently. They are implemented in different repositories, run on different servers and communicate via different network protocols. Thus there is no easy way of determining the (indirect) dependencies between the tests and the new commits in various components.

We propose an approach for test selection that combines static and dynamic analysis of Java components irrespective of their organization into repositories or system topology. We have implemented the approach into existing CI/CD pipelines, so that it only runs a subset of tests required to validate the code being committed without losing quality, i.e., the outcome of the tests that are not selected will not be affected by the changes. Trivially, it results into faster execution of the pipeline because for a given code commit, our approach selects and runs only the relevant tests required to validate that commit.

The rest of the paper is organized as follows. Section 2 explains the approach with formal settings. Section 3 presents the experimental evaluation. Section 4 discusses the existing works related to this topic. Section 5 concludes the paper with a note on future works.

2 METHODOLOGY

2.1 Overview

In order to perform the impact analysis, we need to understand the relationship between the existing tests and the source code. Our approach is to “sniff” the method invocations at the test execution runtime. At the end of testing, we produce a map that links each test with the associated list of methods invoked. For instance, we have a map of three tests and the methods they call when executed:

\[
\begin{align*}
\text{map}(T_1) &= \{A.m1, B.m2, C.m1\} \\
\text{map}(T_2) &= \{A.m2, B.m1\} \\
\text{map}(T_3) &= \{B.m2, C.m2\}
\end{align*}
\]
That means the test $T_1$ calls the method $m_1$ of class $A$, $m_2$ of class $B$ and $m_1$ of class $C$. Similarly, the test $T_2$ calls the method $m_2$ of class $A$ and $m_1$ of class $B$, and so on.

We store this map in the integration tests repository. In the next code commit, we compute the difference with the previous commit to analyze code changes. Thereby, we determine which tests are actually impacted by looking at this map and the methods that are changed in the last commit. Suppose the next commit changes the method $m_2$ of the class $B$, then we perform the lookup into the map and pick tests $T_1$ and $T_3$; the only tests that are impacted. We leave $T_2$ as it is not relevant for this change.

Our approach is safe in the sense that the set of selected tests contains all tests whose behavior may have been affected by the change in the methods.

Figure 2 shows the pipeline with TIA adjusted. We define the concepts used in the approach in the following section, and then explain the implementation stages of the approach.

2.2 | Formal Settings

We assume the source code consists of classes and each class has a set of implemented methods. We consider explicit constructors as methods. We also assume a set of tests that call a subset of those methods when they are executed.

Let $M$ be the set of all methods and $T$ be the set of all tests. We assume that $M$ and $T$ are non-empty sets, i.e., there are one or more methods in the source code and one or more tests that execute a subset of those methods. For this purpose, we define a function $\lambda : T \times M \rightarrow \{1, 0\}$, such that $\lambda(t, m) = 1$ if the method $m$ is executed by the test $t$, and $\lambda(t, m) = 0$, otherwise. We also use a map for each test and the methods it executes defined as $T \rightarrow 2^M$ such that each $t \in T$ maps to a set $\{m_1, \ldots, m_n\} \subseteq M$ where $\lambda(t, m_i) = 1$ for $1 \leq i \leq n$.

A method declaration has six components: name, arguments, return type, modifiers, exception list and an implementation body.

A method is called modified if the method is 1) changed, 2) removed, or 3) added. Each of these types are explained below.

2.2.1 | Changed

A method is changed if its modifiers, exception list or implementation body is changed. Thus, we create the following rule to track method changes.

**Rule 1.** A method is marked modified if any of the following changes occur:

- implementation body, e.g., adding/removing/changing statements.
- modifiers, e.g., adding/removing access modifiers (including static) or keywords like synchronized, throws or abstract.
- exception list, e.g, adding/removing checked exceptions.

Changes to return types do not pose a particular challenge to our methodology. This is because the return statement in the implementation body is changed almost every time the return type is changed. The only exception to changing the return type without changing the return statement is the covariant return type, which does not impact the semantic change without changing the implementation.

Changes to a method name or arguments force changes into other methods which call the changed method. In this case, the other methods will be marked as modified as per Rule 1.
If a method has not been called by any other method, but changing its name or arguments cause it to override/overload an existing method, then it can alter the behavior of the program. Therefore, any changes to the name or arguments are considered as the method is removed and a new method is added. Both of these cases are covered in the next sections.

2.2.2 Removed
A method is removed if its whole declaration is deleted, or changes to its name or arguments occur. Thus, we create the following rule to track method removals.

Rule 2. A method is marked modified if any of the following changes occur

- declaration is removed.
- name is changed.
- arguments (including number, order or types) are changed.

All methods having explicit references to a removed method would require changes to its implementation. Those methods will also be marked modified as per Rule 1. Removing a method may also cause some tests to be invalid if they have an explicit reference to the method. Any adjustments to tests due to a source code change is not in the scope of our methodology.

2.2.3 Added
Any new methods likely result into changes to existing methods or tests. However, if the new method overrides or overloads an existing method, it may alter the behavior of tests without changing the existing code.

If the new method overrides an existing method, the new method may be called in lieu of the existing method due to dynamic binding. In this case, we mark the existing method as modified. We create the following rule to cover this case.

Rule 3. If a method \( \alpha \) is a new method such that it overrides an existing method \( \beta \), then \( \beta \) is marked modified.

If the new method overloads an existing method such that it matches the name, number, order and type of arguments with another method in the class hierarchy, then it can change the static binding of the method in the other class. This is because the binding of such a method depends upon the compile-time types of the arguments. Therefore, all methods in the class hierarchy that has the same name, number, order, and type of arguments are marked as modified. We create the following rule to cover this case.

Rule 4. If a method \( \alpha \) is a new method such that it matches the name, number, order and (super or sub) type of arguments with a method \( \beta \) in any class in the hierarchy, then \( \beta \) is marked modified.

2.3 Example
We explain the approach with the help of the example in Figure 3. The original program consists of class \( A \) and its subclass \( B \). Three tests, \( T_1, T_2 \) and \( T_3 \), and their associated maps of method calls can also be seen. The program goes through four versions of changes as shown in Figure 4. The figure also shows "Modi/uniFB01ed Methods" in each version, "Selected Tests" based on those methods, and "New Mappings" calculated after running those tests. We have shown the map of all tests in each iteration for completeness.

Version 1 adds a new method \( B.bar(Object) \) that overrides \( A.bar(Object) \), which marks it a modified method (as per Rule 1). We find this method in \( map(T_3) \) and therefore \( T_3 \) is selected. The new map for \( T_3 \) is obtained after running the test on Version 1 that now includes \( B.bar(Object) \).

Version 2 overloads \( A.bar(Object) \) by adding \( A.bar(String) \). Note that the new method has the same name, number, order and type of arguments as the original method. The only difference is that the new method uses a subtype of the existing method. This alters the behavior of \( T_3 \) which now resolves to the new method (instead of \( B.bar(Object) \)) due to the explicit \( String \) type argument (i.e., "hello") in the test. We mark \( A.bar(Object) \) and \( B.bar(Object) \) as modified methods (as per Rule 4). The latter is included in \( map(T_3) \), which was updated after Version 1. Thus, \( T_3 \) is selected once again and the new map is updated that now includes \( A.bar(String) \).

\[^1\] Version 1 in Figure 4 provides an example of such a case.
\[^2\] Version 2 in Figure 4 provides an example of such a case.
Program

```java
class A {
    void foo() {
    }
    void bar(Object obj) {
    }
}

class B extends A {
    void foo() {
    }
    void bar(Object obj) {
    }
    void bar(String text) {
    }
}
```

Tests

```java
// T1
test1() {
    A a = new A();
    a.foo();
}

// T2
test2() {
    A a = new B();
    a.foo();
    a.bar("hello");
}

// T3
test3() {
    A a = new B();
    a.bar("hello");
}
```

Mappings

```java
// map(T1)
{ A.A(), A.foo() }

// map(T2)
{ B.B(), A.A(), B.foo() }

// map(T3)
{ B.B(), A.A(), A.bar(Object) }
```

FIGURE 3 Example: Original program (left) and its associated tests with mappings (right).

Version 1: Overriven

```java
B.bar(Object) added
```

Version 2: Overloaded

```java
A.bar(String) added
```

Version 3: Changed

```java
B.foo() changed
```

Version 4: Removed

```java
A.foo() removed
```

Modified Methods

```java
A.bar(Object)  A.bar(Object)  B.foo()  A.foo()
B.bar(Object)  A.bar(String)  A.bar(Object)  B.bar(Object)
```

Selected Tests

```java
T3  T3  T2  T1
```

New Mappings

```java
// map(T1)
{ A.A(), A.foo() }  { A.A(), A.foo() }  { A.A(), A.foo() }  N/A - T1 cannot execute

// map(T2)
{ B.B(), A.A(), B.foo() }  { B.B(), A.A(), B.foo() }  { B.B(), A.A(), B.foo(), A.bar(String) }  // map(T2)

// map(T3)
{ B.B(), A.A(), B.bar(Object) }  { B.B(), A.A(), B.bar(String) }  { B.B(), A.A(), B.bar(String) }  // map(T3)
```
FIGURE 5 Stages of jTIA: 1) Code Analysis, 2) Test Selection, 3) Test Execution/Map Update

Version 3 changes the body of `B.foo()`, which is therefore a modified method (as per Rule 1). As this is included in `map(T2)`, `T2` is selected and map is updated consequently.

Version 4 removes `A.foo()`, which is therefore a modified method (as per Rule 2). As this is included in `map(T1)`, `T1` is selected. However, this test cannot be executed because of the explicit reference to the non-existent method, and hence needs revision.

2.4 Implementation Stages

We have implemented our TIA approach into a pipeline plugin called jTIA, in addition to a set of tools supporting the different stages of the approach. Figure 5 illustrates the stages, which are explained below.

2.4.1 Stage 1: Code Analysis

The first stage of jTIA performs code change analysis by checking out the component’s source code repository. The list of methods `H` that are modified in the last commit is obtained and passed on to the next stage.

Our current implementation deals only with GIT repositories for identifying code changes. For this purpose, we have implemented a tool called GitDiff which is responsible for managing GIT repositories (e.g., retrieving change logs). GitDiff retrieves the list of changes in the last commit by executing GIT commands and then parse the output to find the new/changed/deleted Java files.

Our tool JavaDiff is responsible for analyzing Java files without regard to formatting, comments, whitespaces, or method ordering. The tool implements the four rules defined in Section 2.2. For a changed or deleted Java file, the previous version is checked out and compared to obtain the list of any methods modified. For new Java files, changes in the existing class hierarchy are reported for any overloaded or overridden methods. For this purpose, it uses the call graph analysis of the Soot Java optimization framework. The current implementation supports up to Java 8.

2.4.2 Stage II: Test Selection

The second stage of jTIA selects tests based on the list of modified methods from the previous stage. The integration tests repository is checked out and the mapping for each test to its associated methods is obtained. The test that contains a modified method in its mapping is selected. Any test that has no mappings, such as newly added tests, are also selected. This means that all tests are selected when the pipeline is triggered for the very first time. Any test that has been changed since its last execution, or previously failed will have its map deleted, i.e., all changed and failing tests will also be selected.

Formally, if `H \subseteq M` is the set of modified methods and `T` is a set of tests, then the set of selected tests `T' \subseteq T` is computed in the following way. For each test `t \in T` and a modified method `h \in H`, `t` is added to `T'` if

1. `map(t) = \emptyset`, or
2. `h \in map(t)`

\[\text{git diff-tree -no-commit-id -name-only -r HEAD}\]

\[\text{git show HEAD}^>: <file>\]
okhttp3.Cache.<init>(Ljava/io/File;Lokhttp3/internal/io/FileSystem;)V
okhttp3.HttpUrl$Builder$Companion.portColonOffset(Ljava/lang/String;II)I
okhttp3.internal.cache.DiskLruCache$Companion.create(Ljava/io/File;IIJ)Lokhttp3/internal/cache/DiskLruCache;
okhttp3.internal.cache.DiskLruCache.<init>(Lokhttp3/internal/io/FileSystem;IIJLjava/util/concurrent/Executor;)V
okhttp3.internal.cache.DiskLruCache$cleanupRunnable$1.<init>()V

FIGURE 6 Snippet of a method invocation record of the pragmaNoCache test from the OkHttp case study in JVM's representation of type signatures.

2.4.3 Stage III: Test Execution and Map Update
The third stage of jTIA runs tests in $T'$ one by one on the test servers. For each test $t' \in T'$, map($t'$) is recomputed in order to capture any changes after the commit.

Our tool JVMSniffer\(^1\) is responsible for capturing the method calls for each running test. The tool uses Java Debug Interface API\(^2\) to hook into the target JVM in the debug mode. It provides the ability to trap target events, including method entry and exit events. Therefore, it is able to capture all method invocations on the target JVM objects including types and values of the arguments. It also provides an option to filter method calls based on the package name, which is useful in collecting the information only related to the system under test and avoiding method call traces from third-party or system libraries.

JVMSniffer is deployed per JVM on each test server. Before running a test, jTIA starts JVMSniffer on each server that attaches itself to the target JVM. jTIA then starts test execution, and JVMSniffer starts recording all method invocations during the test execution. For each method invocation, it records the name of the package, class, method, its argument list and return type in JVM's representation of type signatures in the format

<package>.<class>.<method>(arguments)return_type

When the test is complete, jTIA stops all instances of JVMSniffer and collates the method invocation records from each server. Finally, jTIA updates the test map with the list of method calls and checks into the integration tests repository. Figure 6 shows the snippet of the list of method calls from one of our case studies.

3 EVALUATION
In this section, we evaluate our approach and discuss results, threats to validity of our approach and limitations.

3.1 Assumptions
We assume each component in the system is a Java program. Trivially, all components have at least one public class and each of those has at least one method implemented. All tests run under identical conditions, i.e, our test environment, resource configuration and external dependencies remain unchanged during the evaluation to produce deterministic results. This means the results are reproducible if we run the same test suite multiple times on the same code base.

3.2 Case Studies
The experiments have been conducted on 10 well-known opensource projects. They are selected from GitHub\(^3\) based on their popularity and repository size to include both popular and huge projects as well as medium and small sized projects. Table 1 lists the case studies with their versions and size in terms of lines of code. A short description for each case study is given below.

**Apache Commons Collections**: A library from the Apache Software Foundation that extends the Java Collections Framework.

**Apache Commons Lang**: A library for Java utility classes from the Apache Software Foundation.

**Eclipse Collections**: A collections framework for Java with optimized data structures from the Eclipse Foundation. Only two modules, forkjoin and testutils, are considered for experiments.

\(^1\)https://github.com
### TABLE 1 Case Studies with versions and lines of code in thousands (kLoC).

| Case Studies          | Version | kLoC    |
|-----------------------|---------|---------|
|                       |         | total   |
|                       |         | src     |
|                       |         | test    |
| Apache Commons Col.   | 4.4     | 92      |
| Apache Commons Lang   | 3.6     | 112     |
| Eclipse Collections   | 10.0.0  | 327     |
| Netty                 | 3.10    | 105     |
| RxJava                | 2.2.0   | 329     |
| Seata                 | 0.5.0   | 32      |
| OkHttp                | 4.0.0   | 47      |
| MyBatis               | 3.4.0   | 61      |
| UML Reverse Mapper    | 1.4.4   | 2       |
| Symja Parser          | 1.0.0   | 9       |

#### Netty: An asynchronous event-driven network application framework for rapid development of maintainable high performance protocol servers and clients.

#### RxJava: A JVM library for composing asynchronous and event-based programs by using observable sequences.

#### Seata: A high-performance distributed transaction solution for microservices architecture.

#### OkHTTP: A performant HTTP/2 client for Android and Java applications.

#### MyBatis: A persistence framework with support for custom SQL, stored procedures and advanced mappings.

#### UML Reverse Mapper: An automatic UML diagrams generator from code.

#### Symja Parser: An implementation of math expressions parsers for Java double values and Apache Commons Math's complex numbers.

### 3.3 | Research Questions

**RQ1:** Are the selected tests from the proposed approach as effective as the whole test-suite?

To analyze the effectiveness of our approach, we used mutation testing that provides a reliable metric to measure the quality of a test-suite. We used PIT mutation testing tool in our experiments. The mutants are created based on code coverage, i.e., only the methods covered by the tests are mutated. PIT generates mutants via number of mutation operators targeting various semantics in Java. The details of each operator can be seen on PIT’s website. The selected operators in our experiments are

#### Default operators: Conditional Boundary, Increments, Invert Negatives, Math, Negate Conditionals, Return Values, Void Method Call, Empty/False/True/Null/Primitive Returns.

#### Optional operators: Constructor Calls, Inline Constant, Non Void Method Call, Remove Conditionals, Remove Increments.

First, we ran the original program with PIT in order to compute the mutation coverage with the whole test-suite. Then, we changed the tool’s configuration to run only the tests selected by our approach and computed the mutation coverage again. We compared the mutation coverage between the two runs by looking at the number of mutants killed. Ideally, the mutation coverage of the selected test-suite should be equal to the mutation coverage of the entire test-suite, otherwise the selected test-suite is not complete with respect to the changes introduced to the program due to mutations.

We used default settings of operators to generate as many mutations possible depending on the size of the project. There are multiple reasons for which a mutant can be considered as killed, all of which were considered as fault detection.

---

*http://pitest.org/quickstart/mutators/*
TABLE 2 Comparison of mutation coverage.

| Case Studies          | # of mutations | Killed by whole test-suite | Missed by selected tests |
|-----------------------|----------------|----------------------------|--------------------------|
| Apache Commons Collections | 348            | 37%                        | 3%                       |
| Apache Commons Lang   | 1360           | 53%                        | 0%                       |
| Eclipse Collections   | 29             | 53.5%                      | 0%                       |
| Netty                 | 1363           | 33%                        | 0%                       |
| RxJava                | 329            | 72%                        | 0%                       |
| Seata                 | 192            | 17%                        | 0%                       |
| OkHttp                | 543            | 55.67%                     | 0%                       |
| MyBatis               | 169            | 84%                        | 0%                       |
| UML Reverse Mapper    | 27             | 55%                        | 0%                       |
| Symja Parser          | 182            | 32%                        | 0%                       |

- The test was executed and failed.
- The test was run and caused a memory error, e.g., stack overflow or a heap space error.
- The mutation caused an endless loop and terminated due to a predefined timeout (i.e., 3000 msec).

RQ2: What is the gain in terms of the number of tests reduced?

The effectiveness of our approach is measured by the ratio of number of tests selected over the total number of tests, that is,

\[
Gain = 1 - \left(100 \times \left| T' \right| / \left| T \right| \right)
\]

3.4 Results

RQ1: Are the selected tests from the proposed approach as effective as the whole test-suite?

Table 2 shows the number of mutations generated for each case study in the first column. The second column shows the mutants killed by the whole test-suite. The third column shows how many mutants were missed by the reduced test-suite.

The results confirm that all mutants that were killed by the whole test-suite, were also killed by the selected test-suite. The only exception was Apache Commons Collections where 3% mutants survived. The detailed analysis showed that PIT added some mutations to the static initializers in the Java code. The fact that the JVMSniffer only traces method calls, the tests that cause those blocks to be executed were not selected. When adding those tests manually, the number of mutants missed dropped to zero.

RQ2: What is the gain in terms of the number of tests reduced?

Table 3 shows the total number of tests in the whole test-suite and the selected test-suite in the first and second columns respectively. The third column shows the percentage reduction in tests. As per the results in Table 2, running only the selected test-suite is sufficient for the changes introduced due to mutations. There is a significant reduction in the number of tests averaging ≈50%. Eight out of ten case studies had double digit reduction and a few of them is above 80%.

The only case study that could not achieve any reduction was Symja Parser, which is a math utility library. Due to the nature of functions implemented in this library and the fact that we have used math related operators in PIT, almost all methods were mutated. Therefore, all tests in the whole test suite were selected by our approach.

3.5 Threats to Validity

The case studies used in the experiments are real-life projects, but they are not representative of a large integrated system of components. As with any type of evaluation, additional case studies will be required for gaining further confidence in our approach. Having said that, our objective
TABLE 3 Tests minimization gain against the whole test-suite.

| Case Studies               | # of tests | # of selected tests | Gain   |
|----------------------------|------------|---------------------|--------|
| Apache Commons Collections | 3900       | 744                 | 80.92% |
| Apache Commons Lang        | 3039       | 1814                | 40.31% |
| Eclipse Collections        | 178        | 61                  | 87.24% |
| Netty                      | 1213       | 956                 | 21.19% |
| RxJava                     | 3571       | 1917                | 46.32% |
| Seata                      | 245        | 47                  | 80.81% |
| OkHttp                     | 1241       | 1125                | 9.34%  |
| MyBatis                    | 878        | 617                 | 29.73% |
| UML Reverse Mapper         | 67         | 16                  | 76.12% |
| Symja Parser               | 83         | 83                  | 0%     |
| **Average**                | 14593      | 7380                | 49.43% |

was to evaluate the approach for test selection based on code changes, for which, these projects provide a sufficient ground. All of these projects are in active development and widely used in the opensource community. We believe that they provide realistic and diverse examples of what our approach needs to be able to handle in practice.

One threat to validity might come from the mutation strategy used in our experiments. In order to reduce bias, we used the default settings of PIT’s operators for all case studies. We repeated each experiment 3 times and found the same result. This is due to the fact that PIT always generates same mutants for an unchanged code. Another related threat is the presence of equivalent mutants. PIT tries best to ignore such mutants using a built-in bytecode matching library and removes mutants that are trivially equivalent. Nevertheless, equivalent mutants do not pose a particular problem in our experiments as we compare results on the same set of mutants.

Another threat may come from the implementation of JVMSniffer that how accurately it computes the mapping between tests and method executions. The tool uses Java Debug Interface API that has been around since Java 1.4 and quite stable. Moreover, we tested JVMSniffer on all case studies before the experiments and compared the results with the ones in the call graph generated by IntelliJ IDEA.

3.6 Limitations

Our implementation of the approach has some limitations listed below:

3.6.1 Sequential execution:

The approach is designed to run tests in a distributed environment, and multiple JVMs are supported. However, tests must execute sequentially, otherwise JVMSniffer cannot distinguish which method invocation is caused by which test.

3.6.2 Method level changes:

Changes to field initializations, static blocks and constants are currently not considered. This limitation is not fundamental to our approach as most regression test selection techniques are applied on the method level, in contrast to coarser-grained approaches which are based solely on Java file/class level changes.

3.6.3 External resources:

Changes to configurations, properties files, third-party dependencies and other external factors like database, environment variables, are not considered in this approach. Generally speaking, changing resources between multiple test runs is not recommended from the perspective of reproducibility.
4 | RELATED WORKS

We divide the related works into two categories. First, we explore research works aiming for regression test selection. Then, we note some production tools that use change analysis for test selection.

4.1 | Research Works

Traditional Regression Test Selection techniques compute detailed program changes by traversing control flow graphs of two versions of a program and their associated tests at different code granularities, such as statement block, method, class, or file level. Then, the tests which overlap the elements of the graph are selected for execution. Such techniques require a lot of processing and incur non-trivial overhead. Therefore, researchers have also proposed techniques that are applied only on the method level granularity with the combination of static and dynamic analysis.

One of the early works in this realm is Chianti that relies on the computation of structural differences between the two versions of a Java program. It computes Abstract Syntax Tree (AST) of the source program, followed by identifying changes using predefined rules. It generates Control Flow Graph (CFG) to associate tests with the method changes and determines a subset of tests relevant to the changes. Our approach is similar to Chianti with respect to applying rules to cater the language specific semantics and the concepts of object-oriented programming (e.g., polymorphism). Our approach is different in how the test selection process works. We do not compute AST or CFG to determine the test association, which could be computationally expensive. We obtain this information dynamically from the test execution and create the mapping (or association) between the tests and the methods. This fits well with the CI/CD methodology where the code merges are frequent, and therefore computing such trees or graphs could have a detrimental effect on the build timescales. In our case, there is no overhead of creating this mapping. The only pre-processing required in our approach is the identification of method changes at the bytecode level (via Soot optimization framework). We argue that this is a significantly less overhead compared to computing a full CFG of the source code as well as the tests. The main reason is that we use the CFG analysis only if a newly added method has the same method signature as an existing method; so the method override/overload rules, i.e., Rules 3, could be applied.

In contrast to static analysis techniques, dynamic techniques have been used for impact analysis to support regression testing. Orso et al. collect execution traces from program instrumentation. The execution traces consists of all methods that were called during program execution. For each code change, they compute an approximate dynamic slice based on the execution traces that traverse the change. The impact set is the union of the slices computed for each change. Separately, they select an initial set of tests that traversed at least one change based on coverage information. Then, they use the impact set from the execution traces to assess whether, according to the user supplied information (which the authors called ‘field data’), the initial set of tests is adequate, i.e., there exists at least one test that traverses the affected method after traversing the change. This step is based on the intuition that the executions that traverse changed parts of the code are more likely to traverse the affected methods. The main difference between their work and ours is that their goal is to determine which affected methods are not tested by the given set of test cases. On the contrary, our aim is to find the test cases that are affected by those changes. Secondly, their technique is not safe as they rely on coverage data from static analysis. In contrast, we collect test mapping from execution traces. Thirdly, they rely on program instrumentation, whereas we do not require any changes to the source code for our implementation.

Lehnert et al. used UML state machines to identify impacted test cases. In their approach, a test path is a sequence of state machine transitions, which is used to test the class that corresponds to the state machine. For every method changed, the corresponding transition in the state machine which uses this method in its events, guards, or actions is marked as modified. The test sequences that contain this transition is then selected. The underlying hypothesis is that the changes can be identified through design models as the equivalence between the UML models and the implementation. Therefore, they introduced specific rules to perform impact analysis on the changed model elements and to build traceability links between model elements and test cases. Cazzola et al. noted that UML behavioral models such as state machines are often incomplete or quickly outdated, which limits the applicability of existing model-based approaches. As an improvement, they proposed UML class diagrams as a more reliable modeling source, and applied fuzzy logic to deal with inconsistencies. They also rely on test cases modeled as a class diagram, the so-called call usage dependency relationships model. Similarly, Sun et al. used OOCMDG — a dependency graph of classes, methods and variables to create impact rules. The rules identify the impact set according to the change types and then computing the union of all change types for each entity. However, their approach does not work for method body changes and statement changes. In contrast to all these works, we do not rely on any design models or documentation to identify changes. Moreover, our aim is to select test cases that are already implemented as a code rather than represented as abstract models.

Azizi and Do used information retrieval (IR) techniques to select tests during regression testing. They identify common keywords in the difference of two program versions and construct IR queries using the keywords. They also tokenize the source code of the tests to compute their diversity, and build a graph to represent the relationship among tests annotated with the diversity score. Then, they collect tests whose cosine
similarity to the queries is higher than a predefined threshold. Finally, they select the most diverse tests among the collected tests. As normally the case with IR techniques, this solution only works if the source code of the program and tests use mostly the same keywords. The ability to select tests is highly dependent on the textual similarities (adjustable by the cosine similarity threshold) and the ratio of the number of tests to the number of queries.

Symbolic execution techniques have also been used in regression test selection\textsuperscript{30,31,32}. Yang et al.\textsuperscript{32} leveraged symbolic execution with static analysis to identify program statements relevant to changes in the program. They used static analysis to determine the modified program statements; then only the affected parts of the program was explored during symbolic execution. Whilst the goal of this technique is to make symbolic execution efficient in program analysis, Guo et al.\textsuperscript{30} argued that this may leave out some paths if the affected paths are equivalent to some paths in the previous version. They demonstrated their findings on an example of a concurrent program that is executing two threads, where a change in the code of one thread is not identified in the other thread. Hence, the path is not selected in the symbolic exploration. Our approach is not affected by this shortcoming because the test executing the two threads is selected if a change in the code of any of the threads is detected. Nevertheless, the most important drawback of the symbolic execution approach is the algorithmic complexity that can be very expensive\textsuperscript{31}.

4.2 Tools

Code coverage tools use some degree of impact analysis to generate test reports. For instance, \textit{junit4git}\textsuperscript{7} implements JUnit 4/5 extensions to generate reports on tests impacted by the code changes. By using the JVM instrumentation mechanism, the tool can deduce which methods are invoked by which tests. This information is then stored as GIT notes, which is used at the next run of tests. The tool can only apply the impact analysis at the unit level. As noted before, TIA at the unit test level is not very beneficial as they are usually faster to run. The real benefit is filtering integration or acceptance tests.

Google’s internal build system \textit{Blaze}\textsuperscript{8} associates tests with the production code by maintaining a fine-grained directory tree of their source code. Similar technique is applied in \textit{Buck}\textsuperscript{9} for Android apps from Meta and \textit{Pants}\textsuperscript{10} from Twitter. However, all of them support only monorepos\textsuperscript{33}, where code for many projects are stored in the same repository. This is akin to performing TIA at the unit level since all tests can be traced back to their relevant methods statically.

Microsoft’s Visual Studio has incorporated a TIA approach on the integration level for Azure Pipelines\textsuperscript{34}. It examines the test execution data to identify tests which call the modified methods of a C# class. Unfortunately, the details of their work are not publicly available. Furthermore, it only works in a single machine topology where tests and system under test must be running on the same server\textsuperscript{35}. Our approach does not have this limitation thanks to \textit{JVMSniffer} that is deployed on each server running tests.

5 SUMMARY AND FUTURE WORKS

This paper presents an approach for Test Impact Analysis on the integration level, where the source code is distributed in multiple repositories. The approach computes program changes on the method level granularity statically, and selects the impacted tests based on their association with method calls that are collected dynamically. Additionally, we have proposed rules for method changes, additions and removals to cater for Java language semantics. We have used mutation testing to evaluate our approach on a number of opensource Java projects. First, we ran the mutated programs with the whole test suite, then executed the tests selected by our approach to compute the effectiveness. The results achieved ≈50% of test minimization gain with same mutants killed as with the whole test suite.

Our approach for Test Impact Analysis is based on test selection at the method level, but it can be applied on different levels of granularity of program changes, such as statement, class, file or even module level. A simple solution would be to extend the mapping to store additional information besides method calls for each test. However, coarse-grained levels may select tests more than necessary, whereas finer-grained levels may have more processing overhead\textsuperscript{21,24}. We shall investigate multiple granular levels in our approach to find the best trade-off between precision and efficiency in the context of CI/CD pipelines. Another interesting area of research is combining test selection with test prioritization to make delivery pipelines more cost-effective. We are looking into techniques to prioritize the selected tests that are more likely to fail based on historical test executions in the pipeline.

\textsuperscript{7}https://github.com/rpau/junit4git
\textsuperscript{8}https://bazel.build/
\textsuperscript{9}https://buck.build/
\textsuperscript{10}https://www.pantsbuild.org/
References

1. Humble J, Farley D. Continuous Delivery: Reliable Software Releases Through Build, Test, and Deployment Automation. Addison-Wesley Professional. 1st ed. 2010.

2. Memon A, Gao Z, Nguyen B, et al. Taming Google-Scale Continuous Testing. In: Proceedings of the 39th International Conference on Software Engineering: Software Engineering in Practice Track, ICSE-SEIP 2017. IEEE Computer Society; 2017: 233–242.

3. Hammant P. The Rise of Test Impact Analysis. https://martinfowler.com/articles/rise-test-impact-analysis.html. 2017.

4. Gousset M. Test Impact Analysis in Visual Studio 2010. Visual Studio Magazine 2011.

5. Peng Z, Chen TH, Yang J. Revisiting Test Impact Analysis in Continuous Testing From the Perspective of Code Dependencies. IEEE Transactions on Software Engineering 2020: 1–1. doi: 10.1109/TSE.2020.3045914.

6. Molnar A. Live Visualization of GUI Application Code Coverage with GUITracer. CoRR 2017; abs/1702.08013.

7. Indrasiri K, Siriwardena P. Microservices for the Enterprise: Designing, Developing, and Deploying. Apress Berkeley, CA. 2018.

8. Yoo S, Harman M. Regression testing minimization, selection and prioritization: A survey. Software Testing, Verification and Reliability 2012; 22: 67–120.

9. Lindholm T, Yellin F, Bracha G, Buckley A. The Java Virtual Machine Specification, Java SE 8 Edition. Addison-Wesley Professional. 1st ed. 2014.

10. Shahbaz M. Integration Test Impact Analysis. https://github.com/muzammil786/IntegrationTestImpactAnalysis. 2020.

11. Shahbaz M. JVM Sniffer. https://github.com/muzammil786/JvmSniffer. 2020.

12. Vallée-Rai R, Co P, Gagnon E, Hendren L, Lam P, Sundaresan V. Soot: A Java Bytecode Optimization Framework. In: CASCON ’10. IBM Corp.; 2010; USA: 214–224

13. Oracle. Java Platform Debugger Architecture (JPDA) Documentation. https://docs.oracle.com/javase/8/docs/technotes/guides/jpda/.

14. Andrews JH, Briand LC, Labiche Y. Is Mutation an Appropriate Tool for Testing Experiments?. In: ICSE ’05. Association for Computing Machinery; 2005; New York, NY, USA: 402–411.

15. Coles H, Laurent T, Henard C, Papadakis M, Ventresque A. PIT: A Practical Mutation Testing Tool for Java (Demo). In: ISSTA 2016. Association for Computing Machinery; 2016; New York, NY, USA: 449–452.

16. Krochmalski J. IntelliJ IDEA Essentials. Packt Publishing. 2014.

17. Law J, Rothermel G. Whole Program Path-Based Dynamic Impact Analysis. In: ICSE ’03. IEEE Computer Society; 2003; USA: 308–318.

18. Orso A, Apiwattanapong T, Harrold MJ. Leveraging Field Data for Impact Analysis and Regression Testing. SIGSOFT Softw. Eng. Notes 2003; 28(5): 128–137. doi: 10.1145/949952.940089.

19. Ren X, Shah F, Tip F, Ryder BG, Chesley O. Chianti: A Tool for Change Impact Analysis of Java Programs. SIGPLAN Not. 2004; 39(10): 432–448. doi: 10.1145/1035292.1029012.

20. Orso A, Apiwattanapong T, Law J, Rothermel G, Harrold MJ. An Empirical Comparison of Dynamic Impact Analysis Algorithms. In: ICSE ’04. IEEE Computer Society; 2004; USA: 491–500.

21. Gligoric M, Eloussi L, Marinov D. Practical Regression Test Selection with Dynamic File Dependencies. In: ISSTA 2015. Association for Computing Machinery; 2015; New York, NY, USA: 211–222.

22. Celik A, Vasic M, Milicevic A, Gligoric M. Regression Test Selection across JVM Boundaries. In: ESEC/FSE 2017. Association for Computing Machinery; 2017; New York, NY, USA: 809–820.

23. Frattini F, Pietrantuono R, Russo S. Reproducibility of software bugs. Principles of performance and reliability modeling and evaluation 2016: 551–565.
24. Zhang L. Hybrid Regression Test Selection. In: Proceedings of the 40th International Conference on Software Engineering, ICSE 2018. Association for Computing Machinery; 2018; New York, NY, USA: 199–209

25. Lehnert S. A Taxonomy for Software Change Impact Analysis. In: IWPSE-EVOL '11. Association for Computing Machinery; 2011; New York, NY, USA: 41–50

26. Heger C, Heinrich R. Deriving work plans for solving performance and scalability problems. In: . vol. 8721 of Lecture Notes in Computer Science. Springer International Publishing; 2014; Cham: 104–118

27. Cazzola W, Ghosh S, Al-Refai M, Maurina G. Bridging the model-to-code abstraction gap with fuzzy logic in model-based regression test selection. Software and Systems Modeling 2022; 21(1): 207–224.

28. Sun X, Li B, Tao C, Wen W, Zhang S. Change Impact Analysis Based on a Taxonomy of Change Types. In: Proceedings of the 34th Annual IEEE International Computer Software and Applications Conference, COMPSAC 2010.

29. Azizi M, Do H. ReTEST: A Cost Effective Test Case Selection Technique for Modern Software Development. In: 29th IEEE International Symposium on Software Reliability Engineering, ISSRE 2018. IEEE Computer Society; 2018; Los Alamitos, CA, USA: 144-154

30. Guo S, Kusano M, Wang C. Conc-ISE: Incremental Symbolic Execution of Concurrent Software. In: ASE 2016. Association for Computing Machinery; 2016; New York, NY, USA: 531–542.

31. Păsăreanu C, Visser W, Bushnell D, Geldenhuys J, Mehlitz P, Rungta N. Symbolic PathFinder: Integrating symbolic execution with model checking for Java bytecode analysis. Automated Software Engineering 2013; 20. doi: 10.1007/s10515-013-0122-2

32. Yang G, Person S, Rungta N, Khurshid S. Directed Incremental Symbolic Execution. SIGPLAN Not. 2014; 24(1). doi: 10.1145/2629536

33. Brousse N. The Issue of Monorepo and Polyrepo in Large Enterprises. In: Programming '19. Association for Computing Machinery; 2019; New York, NY, USA.

34. Lakshman P. Accelerated Continuous Testing with Test Impact Analysis. https://devblogs.microsoft.com/devops/accelerated-continuous-testing-with-test-impact-analysis-part-1/ 2017.

35. Microsoft . Speed up testing by using Test Impact Analysis (TIA). https://docs.microsoft.com/en-us/azure/devops/pipelines/test/test-impact-analysis 2018.