Mittuniversitetet
MID SWEDEN UNIVERSITY

Exploring the use of call stack depth limits to reduce regression testing costs

Patrik Bogren
Isak Kristola

Final Project
Main field of study: Computer Engineering BA (C)
Credits: 15
Semester/Year: VT 2021
Supervisor: Raja Khurram Shahzad
Examiner: Felix Dobslaw
Course code/registration number: DT133G
Degree programme: Programvaruteknik

Exploring the use of call stack depth limits to reduce regression testing costs

Isak Kristola, Patrik Bogren
BSc. Thesis, Programvaruteknik
Institute of Computer and Systems Sciences
Mid Sweden University
Östersund, Sweden
{iskr1300, pabo1800}@student.miun.se

Abstract –

Regression testing is performed after existing source code has been modified to verify that no new faults have been introduced by the changes. Test case selection can be used to reduce the effort of regression testing by selecting a smaller subset of the test suite for later execution. Several criteria and objectives can be used as constraints that should be satisfied by the selection process. One common criteria is function coverage, which can be represented by a coverage matrix that maps test cases to methods under test. The process of generating and evaluating these matrices can be very time consuming for large matrices since their complexity increases exponentially with the number of tests included. To the best of our knowledge, no techniques for reducing execution matrix size have been proposed. This thesis develops a matrix-reduction technique based on analysis of call stack data. It studies the effects of limiting the call stack depth in terms of coverage accuracy, matrix size, and generation costs. Further, it uses a tool that can instrument Java projects using Java's instrumentation API to collect coverage information on open-source Java projects for varying depth limits of the call stack. Our results show that the stack depth limit can be significantly reduced while retaining high coverage and that matrix size can be decreased by up to 50%. The metric we used to indicate the difficulty of splitting up the matrix closely resembled the curve for coverage. However, we did not see any significant differences in execution time for lower depth limits.

Index Terms - Regression Testing, Test Case Selection, Code Coverage, Dynamic Program Analysis, Execution Matrix

ACKNOWLEDGEMENT

We would like to thank our supervisor Raja Khurram Shahzad for his guidance throughout this work and Felix Dobslaw for the initial idea for this thesis. Furthermore, we would like to express our appreciation to the #java community on IRC for answering our questions about how to use the ASM framework successfully.

I. Introduction

Regression testing is a form of software testing conducted after changes have been made to existing software to ensure that the introduced changes do not introduce problems in other parts of the program [1]. The simplest form of regression testing is *retest-all* where the entire test suite is rerun. However, as the project grows in size, rerunning all tests is usually not feasible due to time and resource constraints. For large-scale projects it is often necessary to reduce the cost of regression testing by various means [2] [3].

One common way of reducing regression testing costs is *test case selection,* which identifies a subset of test cases that can still detect any introduced faults in the changed parts of the software [3]. However, a consequence of selecting a subset of the test suite is that we usually aren't guaranteed to find all faults that may have been introduced in the code. Thus, various heuristics and metrics are used in the selection process to obtain a set of test cases that have a high probability of finding all possible faults. A standard measurement is *code coverage*, which provides knowledge about which parts of the software that were executed by the test suite. This thesis focuses on a particular way of representing coverage data called *coverage matrix*, or *execution matrix,* that maps test cases to structural elements in the system under test [3].

Based on available research, the general effectiveness of code coverage selection methods is hard to establish. Although its wide usage [4] implies that it is an effective technique in many cases, it has also been shown that it in other cases, coverage based selection methods result in no test suite reductions at all [5]. Another issue raised by Inozemetseva and Reid [6] is that the costs of the coverage measurements can be too high to justify the additional information it provides. A problem related explicitly to coverage matrices is that they can be too large to use during test case selection [3]. Currently, no technique of reducing coverage matrix size, or the costs of producing them, has been proposed. However, Niedermayr and Wagner [7] show how stack depth between test cases and tested methods correlates to test effectiveness. Their work studies its implication for mutation testing. There is a lack of research on how the relationship between stack depth and coverage could be used in a test case selection scenario. A hypothesis can be made that limiting the stack during test execution is a viable method for reducing coverage matrix size and costs during test case selection.

This thesis investigates the relationship between stack depth and function coverage by analyzing the execution data of Java applications. Existing coverage tools for Java (such as *JaCoCo* and *JCov*) don't allow the user to limit the stack depth during execution, nor can they produce coverage matrices, so we develop a tool that can create coverage matrices of arbitrary stack depths. The tool is applied to open-source projects from GitHub to generate the empirical data for the thesis. In analyzing the data, we also attempt to understand how much information is lost at different call stack depths of coverage analysis and the possible consequences for the test case selection techniques.

II. Problem Statement

Execution matrices can aid the process of regression test case selection. However, these matrices can be huge for large projects and may be of limited use during the selection process [3]. Furthermore, the overhead costs associated with coverage measurements might cancel out the benefits it provides [6]. Reducing matrix size and generation costs could make the selection process more efficient, but no such techniques have been proposed. It has been shown that a program method is less relevant to the behavior examined by a test case if it appears at a considerable distance from the test case on the call stack [7]. This suggests that excluding methods deep in the call stack from the execution matrix could reduce its size and evaluation costs while retaining a high failure detection accuracy.

To address the problem, this thesis presents a technique for generating coverage matrices for arbitrary call stack depths and examines how it can be applied in a test case selection scenario. To evaluate the effectiveness of the proposed technique and its potential usage during test case selection, it is applied to open-source Java projects to generate empirical data, which is analyzed in terms of coverage accuracy, matrix size, and matrix generation cost.

The first research question we define concerns the relationship between the call stack and coverage matrix size. The size of the matrix has two aspects. First is the two-dimensional size i.e., the number of test cases and methods recorded in the matrix. This is tightly coupled with the function coverage of the test suite. The second aspect is the possibility of splitting up the matrix into several smaller matrices that would be less costly to analyze during the selection process. We hypothesize that when limiting the call stack depth, several methods that link together otherwise unrelated test cases will be excluded from the matrix, making the matrix easier to split into smaller submatrices. We formulate the first research question as

RQ1: What size reductions of execution matrices can be made if the call-stack depth is limited during coverage collection?

We further hypothesize that an approximate relationship between the depth limit and the coverage of the resulting matrix can be described. For instance, "given a depth limit of *n,* it is a good approximation that the resulting matrix will have a function coverage of *c,* where *c* is expressed as a percentage of full coverage". In other words, if we require an accuracy of 95% coverage, there exists a general limit on call-stack depth that we can use as a heuristic that will result in the target coverage. Our second research question is thus:

RQ2: Can we heuristically predict the function coverage for arbitrary depth limits?

Here, we define function coverage as the percentage of functions marked as executed by the matrix when running a test suite with a limited depth compared to running the tests without a depth limit. Hence, if the matrix shows that 90 functions have been executed when the test suite is run with some limit *d* on call stack depth and 100 functions have been executed when run without a limit, the function coverage for the depth *d* is 90%.

The third research question we formulate is related to the costs associated with the collecting of coverage data. With fewer functions being recorded during coverage collection, the execution time will likely decrease.

RQ3: What time savings during coverage collection can be made if the call-stack is limited during coverage collection?

III. Background

Regression testing is performed after existing software has been modified. The goal of regression testing is to make sure that the newly added modifications did not introduce any new faults in the previous features of the program [3]. There are several strategies to achieve this goal and many of them can be combined. The most straightforward approach is the *retest-all* approach in which the entire test suite is run [5].

However, regression testing can be complex and expensive [8], especially when performed on large code bases. Regression testing is considered a significant challenge in industrial settings with large code bases and high standards for software quality due to its resource costs [2]. To reduce the costs of regression testing, several strategies can be employed. These strategies are described in [9]. The major strategies are *test suite minimization*, *test case selection* and *test case prioritization*. Test suite minimization tries to remove redundant tests to minimize the size of the test suite. Test case prioritization attempts to order the tests to increase early detection frequency [10].

The third strategy, which is the topic of this thesis, is *test case selection*. Test case selection is a strategy that involves selecting a smaller subset of the test suite [3]. The selected test cases should ideally be the ones that are relevant to the modifications. Some test selection techniques, referred to as *safe*, guarantee that the selected test cases will find all introduced faults in the modified program [11]. However, most selection techniques are unsafe since these techniques can considerably reduce the number of tests to execute [5]. The drawback is that care must be taken to ensure that the fault detection capabilities are as high as possible for the selected sub-set of the test suite [11]. For this purpose heuristics are usually employed [12]. The heuristics can, for example, be based on the software specifications [13], fault coverage history or test case diversity [4]. However, most test case selection heuristics are based on analysis of the program code, especially various code coverage metrics [4].

Code coverage is a type of measurement that indicates how much, or what parts of the source code are executed when a test case runs [14]. In a regression testing context, coverage can be used as a criterion for the selected test cases [15]. The level of detail of coverage measurements can vary [16] [7]. Fine-grain coverage information potentially renders more efficient test case selection solutions but they are usually performed dynamically, that is, they require execution of the tests and the related code from the software [4]. Dynamic code coverage analysis can be performed by instrumenting the program with code that analyzes the program at runtime. When the coverage information has been collected it can be represented in an *execution matrix* that is the product of the set of test cases and program functions. Each element in the matrix represents whether a particular program function was called during the execution of a specific test case. Given knowledge about the modified functions in the program, the execution matrix is a helpful tool to select the test cases that makes direct or indirect calls to these functions [9].

One way of collecting coverage data from an executed program is *call stack* analysis. Each thread in a stack-based architecture has a related call stack that stores the programs' active subroutines [17]. A function is pushed to the stack when it is called and popped off the stack when it returns. Every function on the stack exists at a certain *depth*, where the depth is the size of the call stack sequence that led to the function being called [18]. Call stack data can be used in various ways to facilitate software testing. For example, the call stack distance between a test case and a given function can be measured [7]. If a function has a high minimum stack distance to a given test case, that could indicate that the function is ineffectively tested. Call stack data can also be used as a code coverage criterion during test suite reduction [18]. Test cases that generate a call stack that is the same or highly similar to another are good candidates for removal since the impact on coverage will be small if removed.

IV. Related Work

An overview of the main strategies used to reduce the cost of regression testing (test case selection, minimization and prioritization) can be found in [9]. They show how the three main strategies are closely related and describe existing problems in detail. Further, they define the test selection problem and give descriptions of various approaches to solve it. Overall, this is a good primer for regression testing.

Since there are so many different test selection techniques it is important to evaluate and compare them. In an influential study, Rothermel and Harrold [13] suggests four categories for evaluating techniques: *inclusiveness*, *precision*, *efficiency*, and *generality*. However as Engström et al. [19] notes, it can be difficult to compare individual techniques since implementations and context can vary subtly. Instead, categories of techniques can be compared. Five categories, *minimization, safe, dataflow-coverage-based, ad-hoc/random,* and *retest-all* was described by Graves et al [20]. Another study by Orso et al. [16] classifies techniques according to the degree of granularity, i.e., at higher levels (methods and classes) and lower levels (statements). However, care must be taken to use adequate criteria and algorithms from each category during comparison. As Jones and Harrold [21] point out, test selection efficiency varies with different coverage criteria and optimization algorithms.

Multiple literature reviews specifically describe and compare test selection techniques. Engström et al. [19] reviews the empirical evaluations of 32 test selection techniques and concludes that no particular technique stands out as superior. This conclusion is echoed in a literature review by Narciso et al. [12], who builds upon the work of Yoo et al. and Engström et al., but has a broader scope and considers selection techniques from other fields in computer science. A more recent review finds that during 2007-2015 mutation score was gradually replacing coverage for evaluation of test suites [22].

Nonetheless, code coverage has been a staple approach for test case selection for many years [4]. A description of coverage based techniques is given in Rothermel and Harrold [13]. Even though strategies other than code coverage (such as mutation testing) have become more popular recently, code coverage remains a widespread test selection criterion with a steady output of research in the area. In [8] the authors show that the costs of reducing the test suite, in terms of fault detection capabilities of the remaining test suite, may be higher in certain circumstances. However, they note that coverage-based reduction techniques significantly outperform random selection. They conclude that the risk of reducing the test suite must be weighed against potential savings and that the relationship between the two is not as clear as previously thought.

Some recent studies have focused on methods that use code coverage in combination with other objectives. Yoo and Harman [9] point out that one difficulty with regression testing is that it often requires the satisfaction of multiple objectives and constraints. Mondal et al. [4] show that the solution sets of the coverage and *diversity* methods generate barely overlapping solution sets and can be efficiently combined.

Code coverage might not be the best approach in every situation. Di Nardo et al. [5] found that the size of selected test suites varies significantly between studies, with some reporting no reductions at all. Inozemetseva and Reid [6] show that there is not necessarily a strong correlation between code coverage and test suite effectiveness. In addition, they suggest that the expenses associated with complex coverage measurements might not justify the additional information that it provides. Mondal et al. [4] compared test case diversity optimization (maximizing dissimilarity of test cases) to code coverage when performing test case selection on real-world Java programs. They found that diversity was slightly more efficient in reducing test execution time.

Call stacks produced during program execution can be utilized to make regression testing more effective. McMaster et al. [18] demonstrate how call stack information can compare test cases during test suite reduction. They follow up their initial work on stack coverage with a study explicitly addressing challenges in multithreaded GUI environments [17]. Another stack-based approach is demonstrated by Niedermayr and Wagner [7]. They show how the stack depth of a tested function correlates with test case effectiveness. The greater the minimal distance between the test case and the tested function, the more likely it is to be inefficiently tested. However, they don't apply this knowledge to regression testing techniques but instead show how stack analysis can be used in place of mutation testing.

This thesis presents a method for generating high-level code coverage data in a scalable way regarding generation cost and accuracy. The data is in the form of execution matrices where test cases are mapped to the tested functions. To approach the general code coverage issues related to generation cost and accuracy described by other works [4], [6], [8], and the specific size-related problems of coverage matrices explained by Yoo and Harman [9], we propose limiting the call stack depth during test execution. The study examines how call stack depth is related to matrix size, accuracy, and generation costs, and its implications for test case selection. Smaller, low-cost matrices could prove helpful in the multiple-constraint test selection scenarios described in [3] where the effectiveness of the selection depends on a combination of constraints/objectives.

Hopefully, the thesis provides some new insights into how the call stack can be used in a regression testing setting. Where previous work on call stacks and test effectiveness have focused on the comparison of call stacks [17], [18], or its implications for the related field of mutation testing [7], this thesis is situated in the context of test case selection during regression testing.

V. Research Methodology

This thesis investigated how varying the call-stack depth during coverage analysis affects function coverage and execution matrix size. The first step was to construct the coverage analysis tool. Next, open-source projects to analyze were selected from GitHub according to a set of predefined criteria. The third step was primary data collection and primary analysis. The developed tool was used to find a suitable subset of depths that could be used for data collection in the fourth step (the data collection/analysis phases are visualized in Fig. 1). The fourth step, i.e., secondary data collection, used the tool with the selected depths to generate data that could be used to analyze and compare with the findings from the third step to find out how coverage and matrix size change with depth.

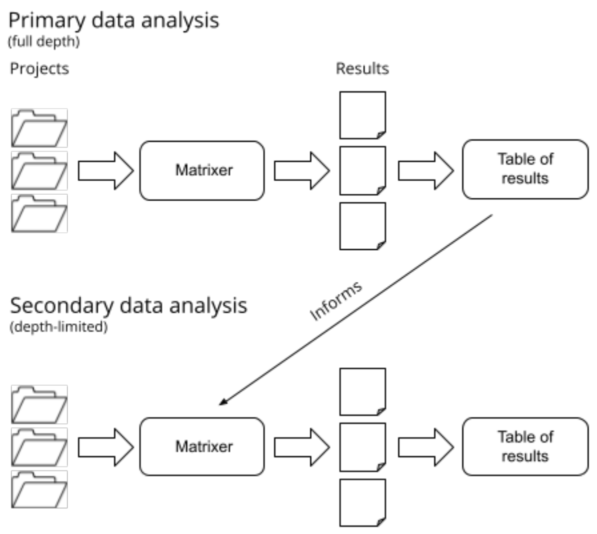


Fig. 1. The workflow for analyzing the projects. The primary analysis informed the parameters for the secondary analysis.

1. *Tool construction*

An analysis tool called Matrixer[1] was developed in the initial phase of the project. The tool was constructed to facilitate data collection for the thesis. It enabled the collection and analysis of execution data from Java programs and generated coverage matrices for arbitrary call stack depths. The tool is described in more detail in section VI. Besides developing the tool itself, a simple Java application for testing the functionality of the Matrixer was created[2].

2. *Project selection*

This section describes the process of selecting projects from GitHub for the empirical part of the study. GitHub offers various project metrics, i.e., user stars or number of forks, as well as an advanced search function. We used the star rating to rank Java repositories from highest to lowest, which allowed us to find the most popular projects on the site (stars indicate user interest or satisfaction with a project and can be used to gauge its popularity [23]). Furthermore, we filtered out projects smaller than 1 MB to increase the frequency of larger projects with many tests and methods. Since project size depends on multiple factors, some smaller projects could have been included in the first selection. After the final selection, the size variation in lines of code ranged from 1370 to 32236 with an average of 12994 (standard deviation 8154). We also excluded repositories that were android based, non-applications (course files, implementations of algorithms/design patterns, etc.), or not documented in English. With these restrictions and rankings in place, we had a systematic method for finding the most popular, large-scale open-source Java application projects.

We selected the first 100 projects that matched the selection criteria from the top hits. We had to make sure that the Matrixer could adequately analyze the selected projects. The requirements that a project must satisfy to be analyzable by the Matrixer tool are the following:

- The project must be developed in Java.
- Gradle or Maven must be used as a build system.
- The project must be able to build without errors.
- JUnit or Testng must be used as a test framework.

Many of the projects had initial build issues or complications with running the test suite. Therefore, we allowed a small amount of time to investigate and try to fix the errors. Usually, 15 minutes was enough to discern whether the problems could be fixed or if the project had to be rejected. Out of the selected projects only 20 would build without errors. Six projects of the remaining 20 had a build structure that prohibited the loading of some of the classes in the agent. These problems could be fixed by a small, manual modification of the build file. This required the tool to be modified to skip the automatic manipulation of the build file.

3. *Data collection and analysis*

The empirical part of the thesis work consisted of four steps: primary data collection, primary analysis, secondary data collection, and secondary analysis. In the preliminary phases, the Matrixer was used to collect coverage of the selected projects without limiting the call stack depth. The results were then analyzed and used to guide the secondary data collection phase where depth limited analysis was made to produce coverage matrices of smaller sizes. In the concluding analysis step, the secondary data was compared to the primary data to understand how varying the call stack depth affects coverage matrix size.

The tool was used by running the command

```
matrixer --target <path to project> --pkg <target package
name>
```

with the flag `--testpkg <test package name>` set if necessary. The main package name for every project had to be manually identified and provided as an argument to the tool. When running this way, the tool automatically handles attaching the agent to the target program by build-file modification, program executing, and generation of coverage data. In the secondary data collection step, the argument `--depth <depth limit>` were also provided to cap the call stack depth.
In the primary data analysis, the execution matrices were manually investigated for generalizations regarding average, median, maximum, and minimum call stack depth. From

[1] https://github.com/ikristola/matrixer/tree/master

[2] https://github.com/ikristola/matrixer-test

these, we could draw some conclusions about whether there is a range of depths that encompasses the most relevant (for regression testing purposes) methods and selecting a set of depths to test in the secondary data collection step. The Matrixer was run with the `--depth <depth>` flag set to collect data for different depths. A run script was constructed that automatically executed the matrixer for the target range of depths. We discovered that the most critical changes in coverage metrics appeared at lower depth limits, so we wrote the script to start with a depth of one and increment the depth limit by 1 for each run. Whenever the script noticed that the number of recorded methods was repeated for consecutive depths, it increased the step at which the depth was increased. This gave us dense data points when the data changed rapidly and sparser data points when the changes decreased. It also sped up the process of running projects that had a significant difference between the average and maximum depth of calls.

The secondary data analysis compared the matrices from both data collection steps. The analysis method we used for individual programs is described in the following section. For every program tested, relevant coverage data was lifted from the matrices and inserted into a spreadsheet designed to perform the analysis calculations. We could also compare full stack and depth limited matrices to see how many methods were not executed for a given depth.

4. *Analysis method*

For each run the Matrixer gave us information about the execution time of the test suite and on the resulting matrix. The matrix data consisted of the number of methods and test cases it contained, the depth of calls, and the number of unique calls per method, expressed as an average and a median for the entire set of methods.

We can define the following:

Coverage	The number of functions executed at least once when running the entire test suite.
F_d	Coverage when using a stack depth limit *d*.
$C = F_\infty$	Full coverage baseline.
$C_d = F_d / C$	The coverage ratio at depth *d*
I_{avg}	The average number of unique invocations per method
T	The number of executed tests
$C_T = I_{avg} / T$	Test coverage per method

For RQ1 we will be looking at C_T, RQ2 relates to C_D and RQ3 is concerned with the execution time of the tests. The *test coverage per method* metric, C_T, gives us a notion of how many test cases are connected through a single method, on average. Another way to look at this value is how full the average row (representing a method) is in the matrix. An increased number of such connections could make the matrix harder to split up into smaller disjoint matrices. Fig. 2 presents a simplified example. It shows a matrix with the minimum depth of calls from a test case to a method. At a depth of six the matrix is impossible to split up without losing information or creating redundancy between the resulting matrices. But if we limit the depth to five it is trivial to divide it up into two matrices, where one of them contains T_1, T_2 and M_2 and the other contains T_3, T_4 and M_1. Hence if we know the matrices where the methods we want to cover are located, the process of selecting the best test cases has fewer tests to consider. This metric is not a perfect indicator of how trivial the matrix is to split up. Instead, what is interesting is how this number varies for different depth limits.

	T_1	T_2	T_3	T_4
M_1			3	4
M_2	3	4	6	

Fig. 2. An example of a matrix that can be split up easily at depth 5 but is not possible to split up at depth 6.

For each metric above, the data from each project was collected into a table and the average, median and standard deviation was calculated.

We expect to see a decrease in C_T as the depth limit is more restrictive. This follows from the hypothesis that passive methods – methods that are deep in the call stack and less relevant to the tested behavior, link together otherwise unrelated test cases. Furthermore, we expect to see a decrease in coverage and execution time at the shallower depths. Some functions may only appear at deeper levels of the call stack, which means they will not be recorded at shallower depths. And if we record fewer method calls there is less work to do for the stack recorder and the execution time should decrease.

5. *Limitations*

The Matrixer had some difficulties analyzing large projects with very complex build structures. Since projects with big test suites and a large number of methods stand to benefit the most from matrix size reductions, it is possible that these were excluded from the study. Furthermore, no considerations were taken about the categories (except being non-android) of the projects during the selection phase. Another limitation related to the Matrixer tool is that there is no way of knowing that it found and instrumented all methods in the target. It is thus conceivable that some of the analyzed projects did not produce correct coverage matrices.

VI. Artefacts

This section describes the Matrixer tool that we used to collect the empirical data for the thesis. The tool was required to analyze Java projects in terms of function coverage per test case and gather call stack data from the program's execution. Furthermore, it allowed us to limit the stack depth for the coverage collection during test execution. Lastly, the tool needed to convert the analyzed results into a visual format that is easy to understand by the user. The tool is written in Java (JDK 11) and uses Gradle for build automation.

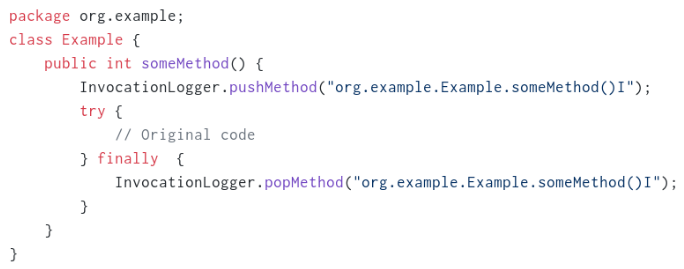
The target applications were dynamically instrumented with code that could collect the coverage information. Instrumentation is the practice of injecting functionality into the source code or the compiled binaries of a program for diagnostic or performance-measuring purposes [24]. A common way of instrumenting Java programs is by using Java Instrumentation API[3] facilitated by Java Agents, packaged in separate JAR files, that attach to the program statically at JVM startup or dynamically through an API call. These agents can change a program by byte-code additions to the compiled code. A custom agent was developed as a submodule to the Matrixer. The agent performs the necessary instrumentation using the commonly used ASM[4] framework.

1. Program flow

When the tool is executed, it begins by identifying the build file of the target project and modifies it so that the agent is attached to the JVM that runs the test suite(s). The tool then uses the build automation tool's native test commands to execute the test suite in a forked process. The agent is started before the tests and target classes are loaded and registers a class file transformer with the instrumentation API. The transformer is then called by the regular class loading process whenever a new class is loaded and modifies the byte code accordingly. When the methods of the target project are instrumented, execution data will be continuously collected while the test suite is running. The tool waits for the test suite to finish executing and then analyzes the execution data to map test cases to the target methods it called and at which depth in the stack the call occurred.

2. Instrumentation

This section describes the Java Agent that is used to extract coverage data from the target program. Each target method is instrumented such that the original code in the method is wrapped in a `try-finally`[5] block. Before the original code is executed, the method calls a static class to notify it that a new method has been called. Similarly, the `finally` block contains a call to notify that the method has finished execution. Placing the last call in a `finally` block guarantees its execution, regardless of whether the function is exited through a normal return or through an exception. Fig. 3 shows how a decompiled instrumented class might look.



```
package org.example;
class Example {
    public int someMethod() {
        InvocationLogger.pushMethod("org.example.Example.someMethod()I");
        try {
            // Original code
        } finally  {
            InvocationLogger.popMethod("org.example.Example.someMethod()I");
        }
    }
}
```

Fig. 3. Example of decompiled instrumented method

Analogous instrumentation is performed on every method in the test classes that are annotated with a `Test` annotation. These methods are instrumented to notify the invocation logger when test cases are started and when they finish. Other methods in the test classes are ignored entirely, making them transparent to the logger. Thus, it appears to the tool that any calls e.g., from test helper methods etc., were made from the `Test`-annotated method. Furthermore, the invocation logging class is thread-aware: it keeps track of when a new thread is created and the parent thread. This allows the logger to record the depth of calls across threads. This is achieved by instrumenting the native `Thread` class.

3. Limitations of the tool

Third party classes and native Java classes (excluding the Thread class) are not instrumented and will not be considered when calculating the stack depth. Furthermore, the tool uses the location of the class files and *Test* annotations to identify test classes and their test cases. If, for some reason, the location is not available, the class will not be instrumented. We could not find any information about when the location of the class files would be missing, it might differ between different JVM implementations. Still, it may be important since these tests are not considered. Furthermore, the method we used for instrumentation does not work with constructors, since they must not be instrumented with code before the call to the super constructor[25]. ASM has special adapter classes developed for handling the complexity of this particular use case, but we were unsuccessful in applying them. For this reason, constructors are not recorded.

The first version of the tool collected the stack depth using Java's native stack trace mechanism in the `Thread` class. As mentioned in [7] this turned out to be a costly operation and not viable for many projects. For this reason, we abandoned

[3] https://docs.oracle.com/en/java/javase/11/docs/api/java.instrument/java/lang/instrument/package-summary.html

[4] https://asm.ow2.io/

[5] https://docs.oracle.com/javase/tutorial/essential/exceptions/finally.html

the approach and created the stack recorder class instead. Initially, we also used the Javassist[6] framework to do the instrumentation. But it did not have the capacity to instrument Java's native `Thread` class. As such we had to resort to the lower-level framework ASM. As the instrumentation required was not too complicated, we decided to remove dependency on Javassist entirely and instead used the ASM framework to do all the instrumentation.

VII. Results

This thesis selected 20 out of 100 open-source Java programs to analyze. The analysis was performed by creating coverage matrices of the test suites for different call stack depths. This section presents the data collected and analyzed to answer the research questions of the thesis.

The projects were first run without depth limitations to acquire baseline data that could be later compared to depth limited tests. Moreover, this data was used to guide the depth limits of the secondary data analysis. The results are shown in Table 1.

Project	Stars	LoC	Test time	Test time. Instr.	Methods	Tests	Max. depth	Avg. depth	Med. depth
Initializr	2.5k	32236	03:05	02:30	2286	967	45	8.2	7.0
Java-faker	2.6k	8388	01:08	04:53	967	556	29	3.9	3.0
Logstash logback encoder	1.8k	15166	00:29	00:32	876	320	14	3.5	3.0
Bootique	1.3k	18599	00:41	00:39	824	235	58	9.5	9.0
apollo	1.6k	14599	00:51	00:56	811	432	28	5.7	5.0
Jimfs	2k	17876	00:48	03:17	735	457	174	2.2	2.0
GCViewer	3.7k	22270	00:52	01:24	646	427	514	5.0	5.0
Telegrambots	2.4k	22424	01:47	01:03	596	64	6	1.2	1.0
Git commit id maven plugin	1.2k	7243	04:29	03:27	330	131	16	6.9	6.0
jcasbin	1.4k	6413	00:10	00:16	300	107	16	4.7	5.0
Client java	1.5k	10228	01:33	05:11	298	179	11	4.4	5.0
Pushy	1.4k	9033	00:56	01:00	267	144	10	2.3	2.0
Scribejava	5.2k	20997	00:49	00:30	262	97	9	3.7	3.0
socketIO client java	4.7k	3791	02:28	02:20	243	86	100	15.5	12.0
Easy rules	3.1k	5917	00:20	00:26	224	189	11	3.7	4.0
Jvm profiler	1.5k	7576	00:08	05:12	161	64	6	2.0	2.0
Egads	1k	7056	00:19	00:19	159	44	5	2.7	3.0
Oryx	1.8k	20214	06:29	03:24	129	98	15	2.8	2.0
Nv websocket client	1.8k	8491	00:09	00:14	90	73	13	4.3	3.0
JavaVerbalExpressions	2.5k	1370	00:07	00:07	66	85	5	1.7	2.0

Table 1. Results from primary data collection

The number of test cases per project range from 44 to 967 with an average of 240, and the number of tested methods range from 66 to 2286 with an average of 513. While the maximum call stack depth in most cases doesn't exceed 50, three projects stand out with very high maximum depths (in one case 514). The average depth varies between 1.2 and 15.5 with all projects but one having an average below 10. Median depth follows the average with a margin of +-1.3, except for one project that has a median depth that is 3.5 smaller than the average.

To determine how limiting the call stack affects matrix size, the projects were instrumented with the tool and the tests executed with various depth limits. The primary results indicated that the shallower depths might be more important to function coverage in general. Yet, the large variation in maximum depth suggests that this might not be true in all cases. Based on these preliminary findings, a set of coverage data for depth one to 20 was collected using the tool. Any project that had not reached 100% coverage at depth 20 was run continuously, with a larger step between depths, until full coverage was achieved.

Table 4 and Fig. 4 present the results of the collection of coverage data. The average coverage at depth one starts at 45.87% and then increases rapidly up to depth six where avg. coverage is 89.59%. The curve then flattens out at around 95% coverage. The median initially follows the average but flattens out at 100% coverage at depth 13. This difference is due to one project (SocketIO-client-java) having a considerably lower coverage, especially at depths 5-15 compared to the rest of the projects. The standard deviation is 19.27% at depth one, indicating high initial coverage variation among the projects (the lowest is 16.10% and the highest 84.80%). However, this variation gradually shrinks, and at depth 20 the deviation is 13.44.

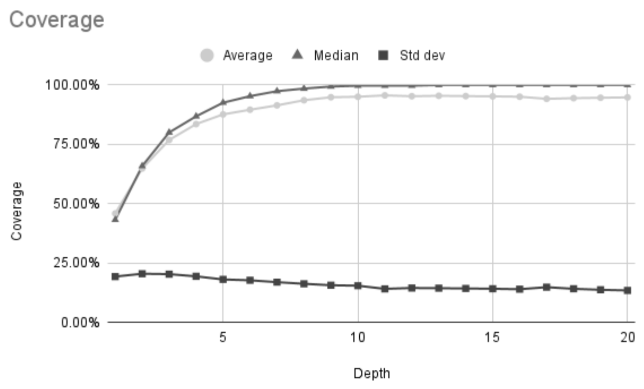


Fig. 4. Average and median coverage for stack depths 1-20

The size of the matrix can be defined as *F x T* where *F* is the number of functions it contains, and *T* is the number of tests. If the tests are the same, then the size of the matrix is proportional to the coverage. When it comes to the possibility of splitting the matrix into several smaller ones, some indication can be found in Fig. 5. It shows the percentage of tests in the matrix that covers the average and median method, C_T. The behavior is highly similar to that of the coverage presented in Fig. 4.

[6] https://www.javassist.org/

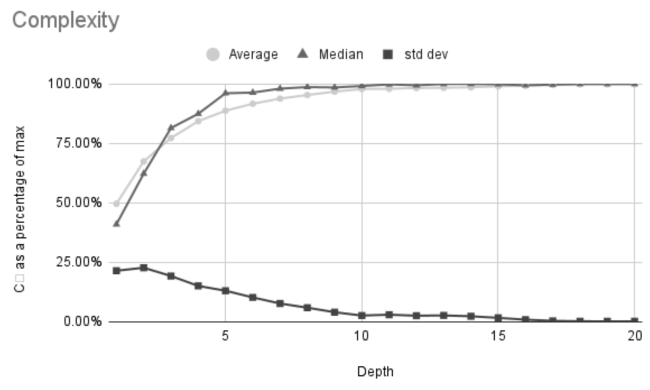


Fig. 5. Average number of tests that execute a given method for depths 1-20. The individual values are compared to the max value for the project: a value of 100% means that the number of tests at that depth limit is equal to the maximum value for the project.

To visualize how individual projects meet arbitrary coverage criteria, the minimum depths required to reach 60, 70, 80, 90 and 100% coverage are shown in Fig. 7. This chart shows that in general, the depths required for coverage targets between 60 and 90% are quite consistent. On average, coverage targets between 50% and 90% require depths between two and seven. To reach 100% coverage, the graph shows larger variations in depths. In average, a depth of 15.35 is required however, only three projects require a depth larger than 15 to reach full coverage. These, on the other hand, needs depths up to 65 for 100% coverage. These results are summarized in Table 2.

Target coverage	Average	Median	Std dev.	Min.	Max.
60%	3.05	2	3.98	1	19
70%	4.10	3	5.22	1	25
80%	5.05	4	6.27	1	30
90%	6.75	5	8.25	2	40
100%	15.35	9.5	17.10	2	65

Table 2. Coverage targets and depths required to meet them

To adjust for variations in maximum depths among the projects tested, Fig. 8 presents the coverage targets with depths expressed as a percentage of the full depth. These results show more significant variations between projects compared to Fig. 6. For 60% coverage, between 0.58% and 40.00% of the stack depth are used. For 70% coverage, the depths vary between 0.78% and 46.15%. For 90% coverage, the numbers are between 1.17% and 61.54%. Table 3 summarizes these findings.

To assess the overhead costs associated with limiting the call stack during coverage collection, the time costs of running the instrumented tests at different depths were measured and compared to the time for coverage collection of the entire depth (Fig. 6). The data does not show any significant changes in execution time except for the first two depths (85.36% and 94.86% of the time for full depth coverage); however, the standard deviation is 26.29% and 20.32%, so these numbers should be taken with caution.

Target coverage	Average	Median	Std dev.	Min.	Max.
60%	14.03%	14.58%	11	0.58%	40.00%
70%	19.01%	17.71%	14	0.78%	46.15%
80%	24.67%	25.00%	19	0.78%	61.45%
90%	31.84%	33.81%	22	1.17%	63.64%
100%	61.57%	69.17%	32	2.33%	100%

Table 3. Coverage targets and percentage of maximum depth required to meet them

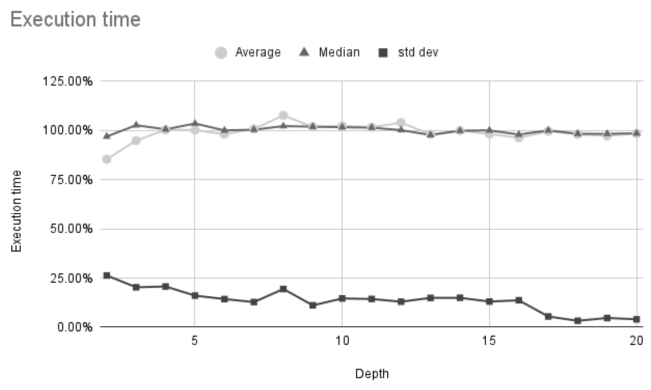


Fig. 6. Average and median execution time for stack depths 1-20

VIII. Discussion

The results show that test coverage behavior is consistent for most projects in the study at call-stack depths up to 10. All projects except one reach 90% coverage within a depth of eight, and 85% reach full coverage at or before a depth of 15. No significant variations due to differences in project size, test execution times, and test cases can be found. When the depths required to reach various coveragep goals are expressed as a percentage of maximum depth, no trends can be detected. However, coverage behavior and maximum stack depth of the tests can vary significantly in extreme cases. This is revealed as one of the projects requires a stack depth of 41 to reach a 90% coverage.

RQ1: What size reductions of execution matrices can be made if the call-stack depth is limited during coverage collection?

As hypothesized, the size of the matrix decreases at shallower depths, along with the coverage of the matrix. However, the

Project	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**	**16**	**17**	**18**	**19**	**20**
Apollo	49.60%	72.50%	81.20%	87.20%	92.40%	95.30%	97.10%	98.50%	99.00%	99.40%	99.60%	99.60%	99.90%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%
Easy-rules	49.10%	63.80%	84.80%	96.40%	99.10%	99.60%	100.00%	100.00%	100.00%	100.00%	100.00%									
TelegramBots	67.70%	89.30%	94.90%	97.70%	99.00%	99.80%	99.70%	99.80%	100.00%	99.90%	99.80%	99.80%	100.00%	99.80%						
Logstash Logback Encoder	40.00%	61.50%	73.40%	82.20%	89.80%	94.50%	97.70%	98.50%	100.00%											
scribejava	44.70%	67.90%	78.60%	84.40%	91.20%	97.70%	98.90%	99.60%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%
GCViewer	30.50%	52.30%	68.90%	81.70%	85.90%	90.60%	94.10%	97.30%	98.60%	98.80%	99.70%	100.00%	100.30%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%
maven git commit id plugin	16.10%	34.50%	48.50%	62.10%	72.10%	86.70%	92.40%	95.20%	97.30%	99.10%	99.70%	99.70%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%
jimfs	49.90%	75.60%	86.00%	93.10%	96.40%	98.90%	99.60%	99.90%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%				
pushy	49.50%	62.30%	74.00%	81.30%	87.20%	93.80%	96.00%	98.50%	100.00%	100.00%	100.00%									
oryx	83.60%	96.90%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%					
nv-websocket-client	30.30%	50.60%	59.60%	61.80%	66.30%	71.90%	78.70%	93.30%	98.90%	100.00%	100.00%	100.00%	100.00%							
JavaVerbalExpressions	84.80%	100.00%	100.00%	100.00%	100.00%															
java-faker	60.20%	75.00%	86.60%	95.70%	96.70%	97.30%	97.80%	98.60%	98.90%	99.00%	99.20%	99.50%	99.70%	99.80%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%
jcasbin	41.70%	56.60%	78.80%	86.40%	92.70%	96.00%	98.30%	99.00%	99.70%	99.70%	99.70%	99.70%	99.70%	100.00%	100.00%	100.00%				
jvm-profiler	64.60%	86.30%	96.90%	99.40%	100.00%	100.00%														
egads	40.50%	70.30%	94.30%	100.00%	100.00%															
client-java	40.50%	70.30%	94.30%	100.00%	100.00%															
initializr	34.20%	49.30%	61.80%	72.30%	80.30%	86.30%	89.90%	92.50%	94.20%	95.70%	96.40%	97.10%	97.70%	98.10%	98.20%	98.40%	98.70%	98.90%	99.10%	99.40%
Socket.IO-client Java	13.90%	17.60%	20.00%	21.60%	23.30%	26.90%	31.40%	33.50%	36.70%	39.60%	44.90%	47.30%	47.80%	50.20%	52.70%	55.50%	57.60%	59.60%	60.80%	61.60%
Bootique	25.90%	43.70%	54.30%	67.40%	79.90%	87.80%	91.20%	93.20%	94.30%	94.60%	95.50%	95.90%	95.90%	96.00%	96.50%	96.80%	97.10%	97.20%	97.20%	97.30%
Average	45.87%	64.82%	76.85%	83.54%	87.62%	89.59%	91.43%	93.59%	94.85%	95.05%	95.63%	95.28%	95.46%	95.33%	95.22%	95.07%	94.18%	94.46%	94.64%	94.79%
Std dev	19.27%	20.46%	20.30%	19.37%	18.07%	17.70%	16.92%	16.23%	15.62%	15.43%	14.10%	14.47%	14.37%	14.26%	14.15%	13.94%	14.82%	14.12%	13.71%	13.44%
Median	43.20%	65.85%	80.00%	86.80%	92.55%	95.30%	97.40%	98.50%	99.35%	99.70%	99.70%	99.70%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%

Table 4. Average and median coverage for stack depths 1-20

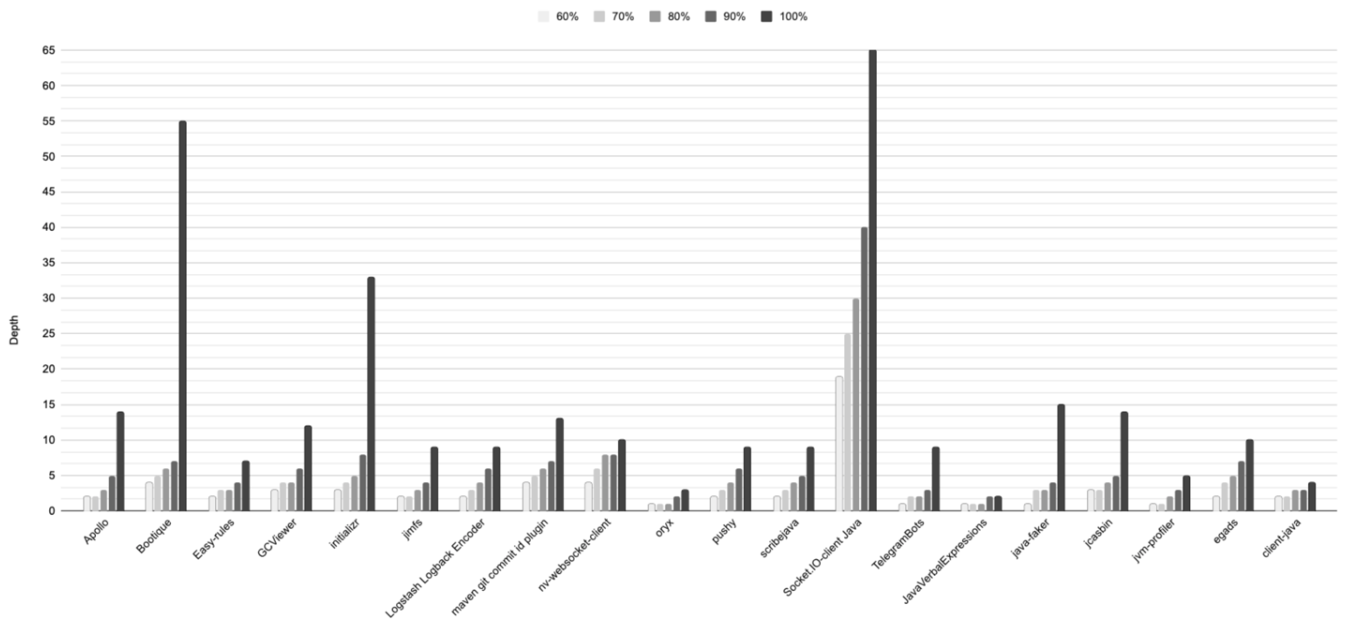


Fig. 7. Depths required to meet various coverage criteria

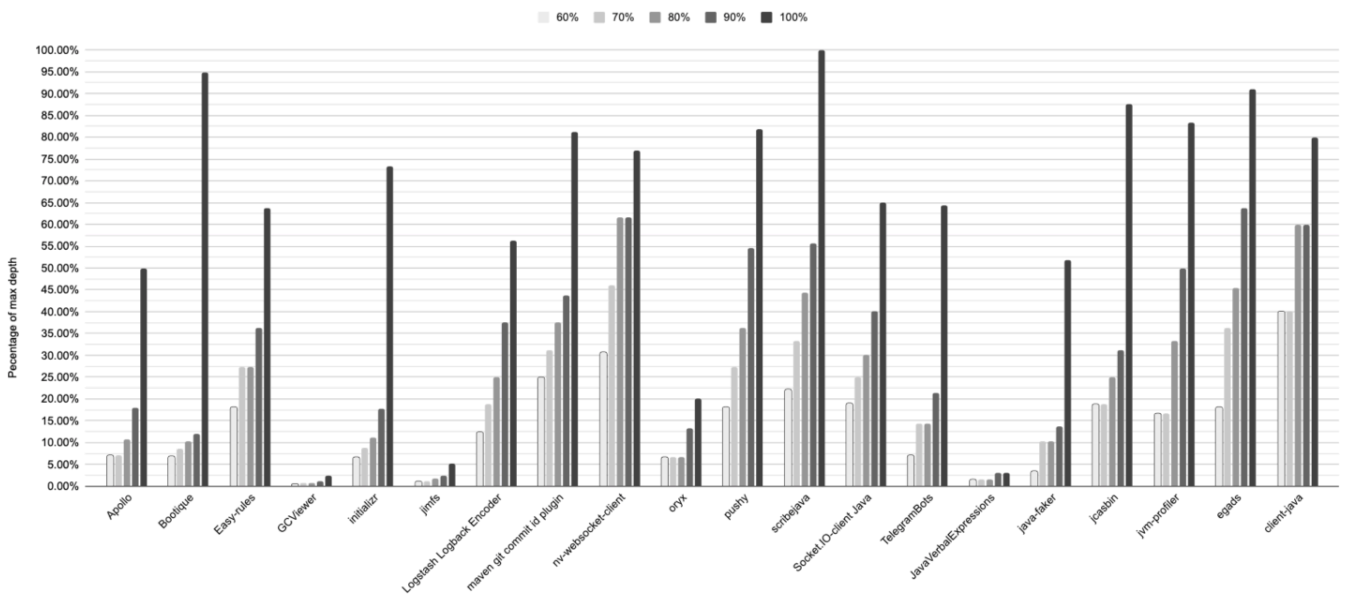


Fig. 8. Depths (percentage of maximum) required to meet various coverage criteria

curve quickly flattens and reaches a close to a maximum value at depth 10. Similarly, the steady increase in C_T in Fig. 5 suggests that as we increase the depth limit, the number of tests that executes the average method is also increasing. This could indicate that it will be more challenging to split up the matrix for higher depths. But how hard that would be in practice needs to be examined further. It may also be because the metric we chose as the heuristic for the possibility of splitting the matrix were not sufficiently useful.

RQ2: Can we heuristically predict the function coverage for arbitrary depth limits?

The consistency of the coverage behavior at smaller depths of the analyzed projects suggests that limiting call stack during coverage collection is a viable technique for increasing the effectiveness of test selection procedures. Depending on coverage requirements, varying depth limits can be used heuristically for predictable size reductions of coverage matrices. In scenarios where a high coverage is required, a stack depth of 10 yields on average 95% coverage. If lower coverage is acceptable, such as when used in combination with other criteria, a smaller depth could be used. At depth seven, the average coverage is 91.43% which could be enough for many applications. However, no coverage guarantees are provided so if meeting the coverage target is critical to the test selection operation, regular full call-stack coverage collection might be necessary. And importantly, these results may differ for larger, more complicated projects than the ones included in this thesis.

RQ3: How much, if any, time savings can be made if the call-stack is limited during coverage collection?

Like Niedermayr and Wagner [7], we noticed a single digit factor slowdown in execution time for the instrumented test suite. The results further show no significant variation between recorded times of coverage collection for different depths, except for the two smallest depths that are executing slightly faster than the rest. However, the standard deviation for those depths is relatively high so these results should be taken with caution. The lack of variation in execution times could be caused by the relatively small size of the tested projects. Most projects (18/20) have test execution times that are under five minutes. It is possible that the largest part of the recorded test execution times consists not of executing code but of various overhead costs such as JVM startup and module initialization. It is still conceivable that the technique we present might render time savings in test suites with long execution times. This might be a good direction for future studies since inefficient coverage measurement methods, as pointed out by Inozemetseva and Reid [6], might not justify its informational value.

Surprisingly, for some of the projects the original test suites had faster execution times in the instrumented version. In most cases where this occurred, the difference was so small that it could probably have been caused by momentary changes in throughput of the test environment. We have no reasonable explanation for the instances with a more significant execution time difference.

1. *Limitations*

We experienced some difficulties with finding large projects that were compatible with the Matrixer. Execution matrix size problems are mainly a problem for industrial-scale projects, however the projects selected for this thesis were comparatively small (in terms of lines of code). Furthermore, many of the selected projects don't have explicit integration test suites that could be run. Instead, we evaluated the entire test suite which included both unit tests and integration tests. This fact may skew the results, since an integration testing suite has a broader reach across software modules than unit tests.

2. *Threats to validity*

There were some notable exceptions among the projects in the coverage behavior and identified maximum depth. This may be a consequence of poor compatibility between the instrumentation method and project structure, but perhaps more likely due to natural variations in program design. This thesis did not examine the internal structure of the projects; hence a direction for future studies could be to investigate coding patterns, APIs, and programming paradigms and find possible connections to call-stack performance during testing.

For most of the projects, the number of test cases recorded by the Matrixer was close to the number discovered by the build tool. But for some projects our tool found a significantly smaller number of tests. One contributing factor to this discrepancy is that some test suites use parameterized and repeating tests. That means that the same test method is run multiple times, possibly with variations in parameters, and the build tool treats each run as a separate test. However, the Matrixer is only concerned with the test method's name, and hence the repeated tests will be treated as a single test.

Concurrency is infamously challenging to test [26], and although the Matrixer itself does not use concurrency, some of the tested projects do. We did our best to write the agent in a thread safe manner, but we cannot exclude the possibility that concurrency bugs may affect the results.

Considering the above limitations, the results of this thesis may not be directly applicable to industrial size projects. More research needs to be conducted on the coverage reduction method proposed in this work and its effectiveness on large scale projects. Yet, the results of this thesis may still point the teams of large projects in the right direction, providing a starting point for their analysis and providing an additional technique that can help make the test case selection process more effective.

3. *Ethical aspects and social relevance*

The motivation for this study was to increase the knowledge on how limiting the stack depth during function coverage collection can make the process of regression testing more efficient. As a large portion of the software development lifecycle is spent on maintenance [27], this can have a beneficial impact on the cost of developing software. Furthermore, if some of the resources used for testing and related activities can be redirected towards other areas of the development process, the software quality may increase. We cannot see any ethical consequences of this study as it doesn't involve any sensitive data or any malicious use cases. All the projects included in this study where open-source and published on GitHub.

IX. Conclusions

This work examined how the use of stack depth limits during coverage collection affects matrix size, coverage, and the execution time of the dynamic analysis. We found that coverage, and thus matrix size, is increasing drastically at shallow depths. However, this increase quickly diminishes and the coverage soon reaches a steady value for most of the projects analyzed. A very similar pattern can be seen for the difficulty of splitting up the resulting coverage matrix into several smaller, disjoint matrices that retain the same coverage. Still, the execution times for the dynamic analysis did not change to a noticeable degree for the tested projects. This suggests that if the particular use case for the matrix can afford to reduce enough coverage, the size of the matrix can be reduced, and potentially be more trivial to split up. Since the size of the matrix affects the cost of analysis, this could reduce the effort of test case selection constraint satisfaction. On the other hand, if the dynamic analysis and coverage data collection are the bottleneck, then other strategies are required.

This thesis only used the average number of tests that invokes any given method as a heuristic for the difficulty of splitting up the matrix in smaller matrices, that would be easier to analyze. Future work may investigate various algorithms with different goals for splitting up the matrix. The most trivial example would be to select a method in the matrix at random and walk the connected test cases and then walk the method they invoke and so on, until a disjoint matrix has been found. The found tests and methods can be put in a separate matrix, and the process is repeated until every method has been traversed. This will retain the same coverage as the original matrix but will likely be hard to achieve in practice.

Other strategies may trade off coverage to achieve a better split between the created matrices. We further suggest that the applicability of the findings of this thesis be explored for larger and more complex projects.

References

[1] C. Coviello, S. Romano, G. Scanniello, A. Marchetto, A. Corazza, and G. Antoniol, "Adequate vs. inadequate test suite reduction approaches," *Inf. Softw. Technol.*, vol. 119, p. 106224, Mar. 2020, doi: 10.1016/j.infsof.2019.106224.

[2] N. bin Ali *et al.*, "On the search for industry-relevant regression testing research," *Empir. Softw. Eng.*, vol. 24, no. 4, pp. 2020–2055, Aug. 2019, doi: 10.1007/s10664-018-9670-1.

[3] S. Yoo and M. Harman, "Regression Testing Minimisation, Selection and Prioritisation : A Survey," p. 60, 2012.

[4] D. Mondal, H. Hemmati, and S. Durocher, "Exploring Test Suite Diversification and Code Coverage in Multi-Objective Test Case Selection," in *2015 IEEE 8th International Conference on Software Testing, Verification and Validation (ICST)*, Apr. 2015, pp. 1–10. doi: 10.1109/ICST.2015.7102588.

[5] D. Di Nardo, N. Alshahwan, L. Briand, and Y. Labiche, "Coverage-based regression test case selection, minimization and prioritization: a case study on an industrial system: COVERAGE-BASED REGRESSION TESTING: AN INDUSTRIAL CASE STUDY," *Softw. Test. Verification Reliab.*, vol. 25, no. 4, pp. 371–396, Jun. 2015, doi: 10.1002/stvr.1572.

[6] L. Inozemtseva and R. Holmes, "Coverage is not strongly correlated with test suite effectiveness," in *Proceedings of the 36th International Conference on Software Engineering*, Hyderabad India, May 2014, pp. 435–445. doi: 10.1145/2568225.2568271.

[7] R. Niedermayr and S. Wagner, "Is the Stack Distance Between Test Case and Method Correlated With Test Effectiveness?," in *Proceedings of the Evaluation and Assessment on Software Engineering*, New York, NY, USA, Apr. 2019, pp. 189–198. doi: 10.1145/3319008.3319021.

[8] G. Rothermel, M. J. Harrold, J. von Ronne, and C. Hong, "Empirical studies of test-suite reduction," *Softw. Test. Verification Reliab.*, vol. 12, no. 4, pp. 219–249, 2002, doi: https://doi.org/10.1002/stvr.256.

[9] S. Yoo and M. Harman, "Pareto efficient multi-objective test case selection," in *Proceedings of the 2007 international symposium on Software testing and analysis*, New York, NY, USA, Jul. 2007, pp. 140–150. doi: 10.1145/1273463.1273483.

[10] H. Zhong, L. Zhang, and H. Mei, "An experimental study of four typical test suite reduction techniques," *Inf. Softw. Technol.*, vol. 50, no. 6, pp. 534–546, May 2008, doi: 10.1016/j.infsof.2007.06.003.

[11] J. Bible, G. Rothermel, and D. S. Rosenblum, "A comparative study of coarse- and fine-grained safe regression test-selection techniques," *ACM Trans. Softw. Eng. Methodol.*, vol. 10, no. 2, pp. 149–183, Apr. 2001, doi: 10.1145/367008.367015.

[12] E. N. Narciso, M. E. Delamaro, and F. D. L. D. S. Nunes, "Test Case Selection: A Systematic Literature Review," *Int. J. Softw. Eng. Knowl. Eng.*, vol. 24, no. 04, pp. 653–676, May 2014, doi: 10.1142/S0218194014500259.

[13] G. Rothermel and M. J. Harrold, "Analyzing regression test selection techniques," *IEEE Trans. Softw. Eng.*, vol. 22, no. 8, pp. 529–551, Aug. 1996, doi: 10.1109/32.536955.

[14] J. Lee, S. Kang, and P. Jung, "Test coverage criteria for software product line testing: Systematic literature review," *Inf. Softw. Technol.*, vol. 122, p. 106272, Jun. 2020, doi: 10.1016/j.infsof.2020.106272.

[15] D. S. Rosenblum and E. J. Weyuker, "Using coverage information to predict the cost-effectiveness of regression testing strategies," *IEEE Trans. Softw. Eng.*, vol. 23, no. 3, pp. 146–156, Mar. 1997, doi: 10.1109/32.585502.

[16] A. Orso, M. J. Harrold, D. Rosenblum, G. Rothermel, M. L. Soffa, and H. Do, "Using component metacontent to support the regression testing of component-based software," in *Proceedings IEEE International Conference on Software Maintenance. ICSM 2001*, Nov. 2001, pp. 716–725. doi: 10.1109/ICSM.2001.972790.

[17] S. McMaster and A. Memon, "Call-Stack Coverage for GUI Test Suite Reduction," *IEEE Trans. Softw. Eng.*, vol. 34, no. 1, pp. 99–115, Jan. 2008, doi: 10.1109/TSE.2007.70756.

[18] S. McMaster and A. M. Memon, "Call stack coverage for test suite reduction," in *21st IEEE International Conference on Software Maintenance (ICSM'05)*, Sep. 2005, pp. 539–548. doi: 10.1109/ICSM.2005.29.

[19] E. Engström, M. Skoglund, and P. Runeson, "Empirical evaluations of regression test selection techniques: a systematic review," in *Proceedings of the Second ACM-IEEE international symposium on Empirical software engineering and measurement*, New York, NY, USA, Oct. 2008, pp. 22–31. doi: 10.1145/1414004.1414011.

[20] T. L. Graves, M. J. Harrold, J.-M. Kim, A. Porter, and G. Rothermel, "An empirical study of regression test selection techniques," *ACM Trans. Softw. Eng. Methodol.*, vol. 10, no. 2, pp. 184–208, Apr. 2001, doi: 10.1145/367008.367020.

[21] J. A. Jones and M. J. Harrold, "Test-suite reduction and prioritization for modified condition/decision coverage," *IEEE Trans. Softw. Eng.*, vol. 29, no. 3, pp. 195–209, Mar. 2003, doi: 10.1109/TSE.2003.1183927.

[22] R. Kazmi, D. N. A. Jawawi, R. Mohamad, and I. Ghani, "Effective Regression Test Case Selection: A Systematic Literature Review," *ACM Comput. Surv.*, vol. 50, no. 2, p. 29:1-29:32, May 2017, doi: 10.1145/3057269.

[23] H. Borges, A. Hora, and M. T. Valente, "Understanding the Factors That Impact the Popularity of GitHub Repositories," in *2016 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, Oct. 2016, pp. 334–344. doi: 10.1109/ICSME.2016.31.

[24] D. Tengeri, F. Horváth, Á. Beszédes, T. Gergely, and T. Gyimóthy, "Negative Effects of Bytecode Instrumentation on Java Source Code Coverage," p. 11.

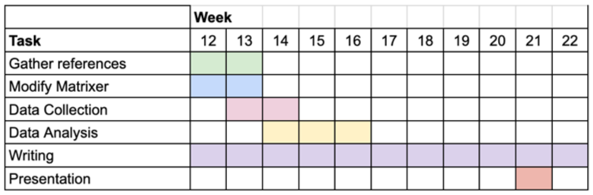
[25] E. Bruneton, "ASM 4.0 A Java bytecode engineering library," p. 154.

[26] C. Wang, M. Said, and A. Gupta, "Coverage guided systematic concurrency testing," in *Proceedings of the 33rd International Conference on Software Engineering*, New York, NY, USA, May 2011, pp. 221–230. doi: 10.1145/1985793.1985824.

[27] P. K. Chittimalli and M. J. Harrold, "Recomputing Coverage Information to Assist Regression Testing," *IEEE Trans. Softw. Eng.*, vol. 35, no. 4, pp. 452–469, Jul. 2009, doi: 10.1109/TSE.2009.4.

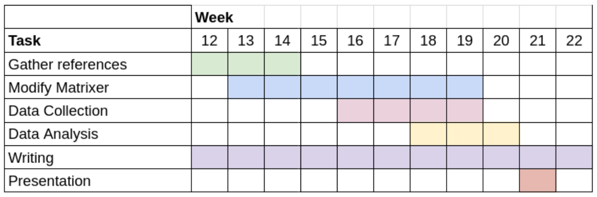
Appendix I: Time Plan

Initial



	Week										
Task	12	13	14	15	16	17	18	19	20	21	22
Gather references	🟩	🟩									
Modify Matrixer	🟦	🟦									
Data Collection		🟥	🟥								
Data Analysis			🟨	🟨	🟨						
Writing	🟪	🟪	🟪	🟪	🟪	🟪	🟪	🟪	🟪	🟪	🟪
Presentation										🟧	

Actual



	Week										
Task	12	13	14	15	16	17	18	19	20	21	22
Gather references	🟩	🟩	🟩								
Modify Matrixer		🟦	🟦	🟦	🟦	🟦	🟦	🟦			
Data Collection					🟥	🟥	🟥	🟥			
Data Analysis							🟨	🟨	🟨		
Writing	🟪	🟪	🟪	🟪	🟪	🟪	🟪	🟪	🟪	🟪	🟪
Presentation										🟧	

Appendix II: Contributions

Patrik Bogren

Report
Abstract & Introduction **50%**
Problem Statement **50 %**
Background **30%**
Related Work **20%**
Research Methodology **70%**
Artefacts **50%**
Results **50%**
Discussion & Conclusion **50%**

Other
Tool implementation **80%**

Isak Kristola

Report
Abstract & Introduction **50%**
Problem Statement **50 %**
Background **70%**
Related Work **80%**
Research Methodology **30%**
Artefacts **50%**
Results **50%**
Discussion & Conclusions **50%**

Other
Tool implementation **20%**

Appendix III: Project Repositories

Apollo	https://github.com/spotify/apollo
Bootique	https://github.com/bootique/bootique
Easy-rules	https://github.com/j-easy/easy-rules
Socket.IO-client Java	https://github.com/socketio/socket.io-client-java
TelegramBots	https://github.com/rubenlagus/TelegramBots
Logstash Logback Encoder	https://github.com/logstash/logstash-logback-encoder
Scribejava	https://github.com/scribejava/scribejava
GCViewer	https://github.com/chewiebug/GCViewer
maven git commit id plugin	https://github.com/git-commit-id/git-commit-id-maven-plugin
Jimfs	https://github.com/google/jimfs
pushy	https://github.com/jchambers/pushy
Oryx	https://github.com/OryxProject/oryx
initializr	https://github.com/spring-io/initializr
nv-websocket-client	https://github.com/TakahikoKawasaki/nv-websocket-client
jCasbin	https://github.com/casbin/jcasbin
jvm-profiler	https://github.com/uber-common/jvm-profiler
egads	https://github.com/yahoo/egads
client_java	https://github.com/prometheus/client_java
java-faker	https://github.com/DiUS/java-faker
JavaVerbalExpressions	https://github.com/VerbalExpressions/JavaVerbalExpressions