
Master Thesis

# Software Testing a Verification System

*By:*

**Mahmoud A. Bokhari**

a1600329

*Supervisor:*

**Markus Wagner**

**10 June 2015**

# Contents

# List of Figures

# List of Tables

**Abstract**

The correctness of software verification systems is vital, since they are used to confirm that safety- and security- critical software systems satisfy their requirements. Modern deductive verification systems need to understand their target software, which can be done by using an axiomatisation base. It captures the semantics of the programming language used for writing the target software. To ensure the correctness of a deductive verification system, it is necessary to validate both parts: the implementation of the system and the axiomatisation base. As a result, it is essential to increase the axiom coverage in order to verify its correctness. However, even for verification engineers, manually creating test cases manually is a time consuming and difficult task. We present a beam search approach to automatically generate test cases by modifying existing test cases as well as a comparison between axiomatisation and code coverage. Our results reveal several interesting points. The overall coverage of the existing test suite can be improved by more than 20%. Furthermore, our approach explores the search space more effectively than existing approaches. In addition, our comparison between the axiomatisation and code coverage criteria shows that there is no clear correlation between them. Our results and some parts of this research report are accepted for publishing in [7, 8].

# Chapter 1

# Introduction

## 1.1 Introduction

Although there are a vast number of books, scientific papers and proven methodologies in software testing, the field can still be considered more of an art than a science. In fact, there is less known about testing than other software development life cycle (SDLC) processes [20]. Software testing involves executing and assessing software systems with the intent of discovering deeply hidden and elusive bugs. This exploratory journey can be accomplished through the assistance of experience, proven methods, principled and well-studied testing strategies as well as gut feelings.

One of the main goals of conducting software testing during the SDLC is to evaluate a computer program's attributes and capability to meet its required functionalities according to its specifications. To achieve this goal with safety- and security-critical systems, the software and/or some of its critical components must undergo

rigorous processes, including formal verification.

Generally speaking, the purpose of software verification is to prove or disprove the correctness of software using formal or informal methods [20]. Examples of formal methods that use mathematical approaches include model checking and deductive verification [13, 15]. Because of the error-prone nature of manually applying such methods, the need for auto-active verification software systems arises.

In order to prove or disprove a program, software systems following the auto-active verification paradigm must understand their target programs, which can be accomplished using the so-called axiomatisation base. Consisting of a large set of axioms (typically hundreds of axioms), it carries the semantics of the programming language used for writing the target programs [5].

## 1.2 Motivation

To ensure the correctness of verification tools, it is necessary to validate both parts: the implementation of the tool, as well as the axiomatisation base. Only testing the implementation is not sufficient, even if a high code coverage is achieved. For example, it was noted in [5] that the axiomatisation coverage was as low as 1% for some tests (for the given verification system), while code coverage was never less than 25%. In other words, the used test suite exercises a certain amount of the code, while it uses only a small number of "core axioms". As a consequence, some logical defects stay hidden within the axiomatisation unless it is fully exercised. The work in [5] discovers two bugs in the axiomatisation as a result of the coverage maximisation research.

Our motivation comes from the fact that it is essential to prove the correctness

of these types of software systems, particularly when they are used in practice. More specifically, it is vital to evaluate the axiomatisation base because it is the core component of verification systems [29]. Such verification can be completed by maximising the proportion of axioms successfully used during the verification process [5, 9]. However, due to the large number of axioms as well as the time required to verify a program, it is a difficult task even for experienced test engineers to manually create adequate test cases from scratch to maximise the axiom coverage [29].

## 1.3 Challenges

This problem of maximising the axiomatisation coverage is challenging for iterative search approaches due to two main reasons: the large number of axioms (typically hundreds) and the time consuming verification process (sometimes minutes). In addition, the latter reason also makes it unsuitable for population-based evolutionary algorithms or ant-colony optimization as they require many evaluations [2, 12]. Besides the time-consuming evaluation process, the vast number of infeasible ways of creating test cases renders the problem inappropriate for disruptive approaches, such as simulated annealing and even the simple (1+1) evolutionary algorithms.

In addition, the solution space for such a problem –maximising the axoimatisation coverage– is massive. For example, the KeY verification system uses an axiomatisation base that contains 1,520 axioms [7]. Consequently, the search space contains $2^{1520}$ possible solutions. Furthermore, valid solutions are scattered significantly throughout the search space.

## 1.4 Contribution

After analysing the work of [29] to maximise the axiomatisation coverage using five different variants of breadth-first and depth-first search methods, we observed that there are some instances where the implemented search approaches ignore valid solutions. Moreover, the approaches explore blindly the vast solution space. Therefore, we propose an extension to the author's framework –one that applies two different search methods. The first one is a heuristic approach that involves a learning process and is based on the beam search algorithm [14]; the second method is a fast version of the first approach that heuristically eliminates minutes and sometimes hours of the evaluation time. In addition, we have implemented a tool that can be used to analyse the test logs and build a guidance table to inform and guide our search approaches towards high coverage.

Our experimental results reveal interesting points. There is no clear correlation between code and axiomatisation coverage; therefore, it is essential to focus on maximising the axiom coverage to uncover hidden defects. In addition, our approach achieves axiomatisation coverage comparable to that of the union of five previous approaches. Furthermore, the overall coverage has been improved by 24%.

# Chapter 2

# Background

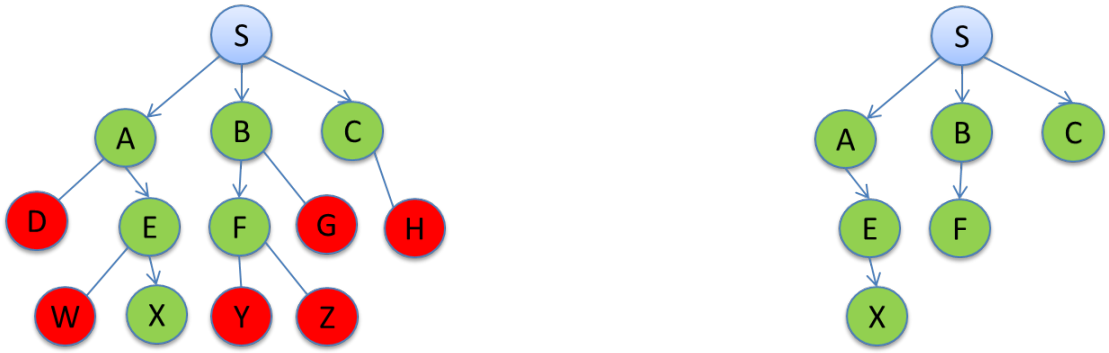
## 2.1 Introduction

In this chapter, we briefly provide some background information, to facilitate the reading of the next chapters.

## 2.2 Beam Search

Beam search is a heuristic search technique that has a similar exploration technique to the breadth-first search (BFS) method. Figure 1 illustrates the mechanism used for exploring a graph using the beam search approach. It expands the search space by visiting only the most promising nodes in each level and discarding the rest. Figure 2.1(a) shows the promising nodes in green and the non-promising nodes in red. Its mechanism is unlike the BFS, which explores all nodes at each level [14]. In addition, the beam search technique requires domain information regarding the

solution space in order to determine the next best nodes to explore. As can be seen in Figure2.1(b), using the beam search approach, the number of nodes to visit is reduced by half.




(a) The graph's nodes to be explored by the beam search algorithm. Green nodes are promising nodes, while the red ones are non-promising nodes.

(b) Nodes chosen for exploration. A 50% reduction in the number of nodes.

Figure 2.1: Beam search algorithm's search space

## 2.3 Code Instrumentation

During the test phase, test engineers and software developers need to know what happens inside the system under test (SUT). To do so, they use techniques by which valuable data can be collected in regards to the execution of the SUT. One such technique is code instrumentation, which is the process of inserting several code segments into the SUT [28]. These segments permit the collection of test coverage data without changing the logical properties and the major functions of the SUT. When the inserted code is executed, it generates data and dumps it into a file for later use in producing reports. Using the generated data, a developer can analyse the product's performance, diagnose errors and focus on areas that need

additional attention.

### 2.3.1 Static Instrumentation

Static instrumentation is sometimes called source code instrumentation, which is a technique for inserting instructions into the source code of the SUT before compiling it [28]. It can be only used when the source code is available and requires recompilation of the SUT. After the instrumentation is complete, these extra instructions generate data to be used for testing and runtime analysis. Although it increases the compilation and execution time, it is adaptable to a wide range of processors and platforms as well as it produces accurate results [28].

### 2.3.2 Dynamic Instrumentation

Dynamic instrumentation can be categorised into two techniques: byte code and on-the-fly instrumentations. Byte code instrumentation is a method of modifying byte code files before the loading phase, while the on-the-fly instrumentation technique starts working after the SUT is loaded.

To illustrate this process, the typical development phases of Java can be used as an example [28]. The first phase occurs when a programmer writes the source code. In the second phase, the source code has to be compiled into Java byte code, which is known as a ".class" file format. This file contains the instructions to be executed later in the execution phase. In the third phase, the Java Virtual Machine (JVM) loads the byte code files and transfers them to the main memory. After the .class files are loaded, JVM verifies these files using the byte code verifier. JVM assures that they are valid and do not violate Java's security policies. In the final stage,

the JVM executes the instructions.

Coverage tools that use the byte code instrumentation method modify the .class files before JVM loads them. When on-the-fly instrumentation is used, the probes are inserted into memory to monitor the SUT [28]. In other words, the SUT remains unchanged; therefore, unlike the other techniques, there is no need to remove these probes after the test is conducted. Moreover, no recompilation is needed in dynamic instrumentation methods, while it is essential when static instrumentation is used. Finally, the availability of the source code is not required as no effects are applied to it.

## 2.4 Verification Systems

A verification system takes a program and its specification as an input in order to prove the program correctness with respect to its specifications. It uses different formal verification methodologies such as formal model checking and deductive verification. In addition, each methodology has its own processes for finding a proof.

## 2.5 Deductive Verification Systems

Generally speaking, a deductive verification system transforms the software to be verified into mathematical expressions and then proves its correctness. To find a proof, it is important to provide the software's requirement specifications. The specification can be written in informal or formal language. A requirement specification can be only limited to the safety. For instance, the software will

always execute without encountering a fatal error such as accessing an invalid memory location or dividing by zero. Nevertheless, ensuring such a property is a complex task [15].

For example, a bug was discovered within the Java standard library. It was linked to the computation of the average of two integer numbers [21]. It occurs in some binary search implementations when computing $(i + j) / 2$. For some very large numbers $i$ and $j$, the result might be become a negative number if the addition overflows. As a consequence, such cases result in some fatal errors in accessing array elements, which is accessing outside of the array boundaries. As can be noted in this example, proving safety requires proving the absence of arithmetic overflows, which sometimes includes checking other properties.

### 2.5.1 Proof Procedure

In deductive verification systems, axioms and calculus are used to derive the proof. Axioms are a set of rules that helps the verification system to understand and verify its target software. In other words, axioms encode the semantics of the programming language that is used to write the software. On the other hand, calculus is the implementation of the proof procedure, i.e. the steps for reaching the proof.

In this research we concentrate on modern deductive verification systems that follow the auto-active verification paradigm. In such systems, besides the program to be tested, the specification as well as all relevant information for finding a proof, such as loop invariants, is given to the verification tool right from the beginning of the verification process.

We consider only system test in this research, i.e. the verification system is tested as a whole. Although the correctness of a tool depends on the correctness of its components and it makes sense to also test these components independently, not all components are easy to test individually. For example, it is quite challenging to test the verification condition generator separately as it is extremely difficult to specify its correct behaviour. In addition, we only consider functional tests that can be executed automatically, and therefore, user interface properties are not considered.

### 2.5.2 Test Case Design

As is typical for verification tools following the auto-active verification paradigm, we assume that a verification problem consists of a program $P$ to be verified and a requirement specification $REQ$ that is added in the form of annotations to the program. Typical annotations are, for example, invariants, pre-/post-condition pairs, and assertions of various kinds. Besides the requirement specification, a verification problem usually contains additional auxiliary annotations $AUX$ that help the system to find a proof. We assume that all auxiliary input such as loop invariants are made part of the testing input, such that the test can be executed automatically.

Possible outcomes of running a verification tool on a test $P{+}(REQ \cup AUX)$ (a verification problem consisting of a program $P$, a requirement specification $REQ$, and auxiliary annotations $AUX$) are:

**proved:** A proof has been found showing that the program $P$ satisfies $REQ \cup AUX$.

**not provable:** There is no proof (either $P$ does not satisfy $REQ$ or $AUX$ is not

sufficient); the system may provide additional information on why no proof exists, e.g. by a counter example or by showing the current proof state.

**timeout:** No proof could be found given the allotted resources (time and space).

## 2.6 Test Coverage

In 1972, Dijkstra observed that software testing can be utilised to reveal the presence of bugs, but cannot be used to prove their absence [31]. Over the last four decades, there has been rapid growth in the field of software testing. Furthermore, researchers have discovered rigorous methodologies and have conducted formal experiments. Meanwhile, software testing has been expanded to include software verification and validation techniques.

In addition to Dijkstra's observation, Goodenough and Gerhart made a breakthrough in the software testing field, pointing out that the central question to be answered in software testing is "What are the test criteria?" [31]. In other words, a test criterion defines what constitutes an adequate test. As a consequence, this area has become a major research focus, and several test criteria have been discovered and used in the software industry, including code coverage measures [31].

## 2.7 Code Coverage Types

As mentioned above, a number of test coverage criteria have been discovered in the field of software testing; one widely-used measurement is code coverage [31]. It helps testers to discover the code segments in the SUT that have been executed during a test run as well as the dead code that has not been executed. It can also

be used to measure the quality of a test suite and to improve it to cover more code segments.

### 2.7.1 Statement Coverage

Any written software consists of a sequence of statements [11]. A statement can be declarative or executable. An example of a declarative statement is the *#define* and *float* statements in the C programming language. By comparison, an executable statement can include *if, while* or assignment statements.

The statement coverage reports whether or not each statement has been executed in the SUT. In other words, it calculates the ratio between the total number of executed statements and the total number of all statements.

### 2.7.2 Branch Coverage

In control statements, such as *if* and *switch* statements, more than one execution scenario may occur. Each scenario can be considered a branch that depends on the conditions in the control statement from which it started. To consider the branch as fully covered, all outcomes of its starting condition must be evaluated. In other words, the conditions have to hold true using some test cases, and they have to be false using the same or other test cases [19].

### 2.7.3 Method Coverage

Methods consist of several line of codes (LOC) that accomplish a specific task. They sometimes take in data, process it and return the result. They can also be called as required by other functions and programs. Method coverage is used to report whether each method has been invoked during a test run. Since a method may contain several exit points, keeping track of each method is essential in identifying any dead code.

### 2.7.4 Class Coverage

Class coverage reports all covered classes during the test phase. In the object-oriented programming paradigm, software is built up using one or more classes. A class is the blueprint composing an object and it contains the object's attributes and actions. To be considered for class coverage, classes have to be executables. Each class is counted as covered when it is loaded and initialised.

## 2.8 Axiomatisation Coverage

Even though measuring code coverage is an essential task in testing verification systems to assess the quality of a test suite, it is not an adequate metric for measuring how well the axioms are tested in a verification system [5]. For instance, our experiment shows that while the axiomatisation coverage is as low as 0.32%, the exercised code is 36%. As a result, it is important to maximise the axiomatisation coverage.

To address this problem (the huge gap between axiom and code coverage), the authors of [5] propose the notation of axiomatisation coverage in their research work. It measures the amount of axioms used in a test suite during the testing of a verification system. The major idea is to calculate the percent of axioms that are successfully used to find the proof. An axiom is defined to be *needed* to verify a program, if it is an element of a *minimal axiom subset*, by which the verification system can find a proof. That is, if the axiom is removed from this subset, the tool is no longer able to prove the correctness of the program.

**Definition 1 ([5])** *A test case* $P{+}(REQ \cup AUX)$ *covers the axioms in a set* $Th$ *if* $Th \vdash P{+}(REQ \cup AUX)$ *but* $Th' \nvdash P{+}(REQ \cup AUX)$ *for all* $Th' \subsetneq Th$.

## 2.9 Summary

This chapter presented several interesting topics. It started with the beam search algorithm, which is the backbone of our approaches. It heuristically expands the nodes within the search space by visiting only the promising nodes.

Verification systems are used to prove the correctness of a software product. One type of such system is deductive verification systems that transform their inputs into mathematical expression in order to prove it. A deductive verification tool utilises an axiomatisation base to understands its input. The axiomatisation base contains axioms that encode the semantic of the programming language used for writing the input. The input consists of a program and its formal specification. The output of the verification system in general is either the program is correct or is incorrect.

In software testing it is important to define the test criteria. Code coverage is considered an important criterion in software testing that must be taken into consideration. It measure the exercised code in the SUT. It has several types such as statement and branch coverage. Another example of test coverage is the axiomatisation coverage. It is the process of measuring the exercised axioms during the test phase.

In order to measure the code coverage, the SUT has to be modified. This can be achieved through code instrumentation techniques: static and dynamic instrumentation. The former modifies the source code by inserting special code segments while the latter alters the SUT's class files before or after they are loaded in the main memory.

# Chapter 3

# Related Work

## 3.1 Introduction

In this chapter, the related work is presented. It is divided into three sections. The first is devoted for the code coverage, while the related work of the axiomatisation coverage is discussed in the second section. The analysis of the current framework for measuring and maximising the axiomatisation coverage is presented in the last section.

## 3.2 Code Coverage

To compute test coverage, software developers and test engineers use code coverage tools to ensure all LOC in the SUT have been executed. These tools vary in their methodologies and supported features. For example, some tools use source and/or byte code instrumentation techniques. In addition, they range in terms of the type

of code coverage. For instance, EMMA [24] supports the statement, method and class coverage, whereas, Jcover [25] supports all those types of coverage as well as the branch coverage. The reporting techniques differ between each tool, however, most of them produce file-based reports. Table 3.1 summarises the main features of the tools, while Table 3.2 illustrates the additional features.

### 3.2.1 AgitarOne

AgitarOne is an integrated and comprehensive unit testing tool for Java software [22]. It automatically creates Junit tests and observes the SUT's behaviour, including the code coverage. It uses dynamic instrumentation techniques to insert probes into the SUT. It also supports the statement, method, class and branch coverage. GUI reports are used in AgitarOne to show code coverage. It also reports the complexity scores of methods and classes in the SUT, which in turn helps developers concentrate on more complex components. Finally, it can be used to assist in software debugging by providing snapshots and stack traces to identify bugs and their causes.

### 3.2.2 Cobertura

Cobertura is an open source analyser for Java programs. It calculates the percentage of the executed code in the SUT [23]. It uses the byte code instrumentation approach to monitor the SUT. It also reports on the statement and branch coverage. The styles of the generated reports are HTML and XML files.

### 3.2.3 EMMA

EMMA is an open source Java code coverage tool [24]. It uses byte code instrumentation approaches to insert probes into the SUT's classes. Furthermore, on-the-fly instrumentation techniques can be utilised in cases where the byte code has already been loaded by the JVM. It also reports coverage in file-based reports, such as HTML and XML files. These reports contain statement, method and class coverage. Testers can define a coverage threshold; as a result, items with coverage percentages less than the predefined threshold will be highlighted. It also provides a special feature for merging coverage data obtained from different test runs.

### 3.2.4 eXVantage

eXVantage is a tool suite that provides code coverage analysis, debugging and performance profiling [30]. It uses source code analysis for C and C++ software and byte code instrumentation for Java programs. It also supports the statement, method and branch coverage. It reports the code coverage in file-based reports while providing its own GUI for displaying state-of-the-art reports. Finally, it can be used for program profiling to detect heavily executed components of the SUT.

### 3.2.5 Jacoco

The Jacoco is an open source tool that provides coverage reports for Java software [16]. It instruments the source code of the SUT. For source code analysis, it has to be used with the Eclipse plug-in EclEmma [16]. In addition, it implements byte code and on-the-fly instrumentation techniques. It also produces several report

formats such as HTML, XML and CSV for the statement, method, class and branch coverage.

### 3.2.6 JavaCodeCoverage

The JavaCodeCoverage tool is a code analyser developed in the Indian Institute of Technology. It uses a byte code instrumentation technique to evaluate Java programs [18]. It has its own reporting system to produce code coverage reports, including statement, branch, method and class coverage. In addition, it uses a MySQL database management system to store all gathered data, thus extending the utility of the tool to several other purposes. For instance, it stores test data obtained by running each individual test case, which allows the tester to review and improve it.

### 3.2.7 JCover

JCover is a code coverage measurement tool for Java programs [25]. It generates statistical information on the code coverage of the SUT during the test process. It supports statement, method, class and branch coverage. In addition, it can be used with the SUT source and byte code. It generates several types of reports, such as HTML, XML and CSV files while maintaining its own GUI for viewing the coverage reports. Moreover, it has a unique feature, "coverage differencing", which can be used to analyse the coverage of test cases. In other words, it shows whether the test cases are overlapping or disjointed, allowing a test engineer to improve the test suite by removing any redundant test cases.

### 3.2.8 Jtest

With static and dynamic instrumentation techniques to monitor the SUT, Jtest is a powerful analyser tool that encompasses those features as well as other features [26]. In addition, it provides reports that include all coverage types; it also has its own reporting system for displaying sophisticated reports while supporting file-based reports, such as HTML. It checks the code against built-in and customised rules for code errors, making it useful for detecting the most frequent errors made by developers. Additionally, it manages code reviews by automating the review preparations, notifications and tracking. In order to maximise branch coverage, it automatically generates test cases in JUnit format to test each branch while the SUT is running.

### 3.2.9 PurifyPlus

The IBM Rational PurifyPlus tool is a powerful solution for measuring code coverage [27]. It supports both source and byte code instrumentation approaches. Unlike the other tools, it only supports statement and method coverage. In terms of the supported programming languages, it can be used with C, C++, Java, Basic and all programming languages in the .net framework. Moreover, it has a GUI-based reporting system to produce coverage reports. Similar to eXVantage, it provides debugging and performance profiling.

| | **Instrumentation** | | | **Coverage Type** | | | | **Reporting** | |
|---|---|---|---|---|---|---|---|---|---|
| **Tool** | **Source code** | **Byte code** | **On-the-fly** | **Statement coverage** | **Method coverage** | **Class coverage** | **Branch coverage** | **GUI** | **File** |
| **Agitar** | N | N | Y | Y | Y | Y | Y | Y | N |
| **Cobertura** | N | Y | N | Y | N | N | Y | N | Y |
| **EMMA** | N | Y | Y | Y | Y | Y | N | N | Y |
| **eXVantage** | Y | Y | Y | Y | Y | Y | Y | Y | Y |
| **Jacoco** | Y | Y | Y | Y | Y | Y | Y | N | Y |
| **JavaCodeCoverage** | Y | Y | N | Y | Y | Y | Y | Y | Y |
| **JCover** | Y | Y | N | Y | Y | Y | Y | Y | Y |
| **Jtest** | N | Y | Y | Y | Y | Y | Y | Y | Y |
| **PurifyPlus** | Y | Y | N | Y | Y | N | N | Y | N |

Table 3.1: The main features of code coverage tools.

| **Tool** | Language | License | Year | Junit Test | Debugging Assistant | Other |
|---|---|---|---|---|---|---|
| **Agitar** | Java | Commercial | 2010 | Y | Y | Classes complexity scores |
| **Cobertura** | Java | Open source | 2010 | N | N | |
| **EMMA** | Java | Open source | 2005 | N | N | Coverage threshold Merging test run reports |
| **eXVantage** | C, C++ | Commercial | 2006 | Y | Y | Define heavily used components |
| **Jacoco** | Java | Open source | 2014 | N | N | |
| **Java Code Coverage** | Java | Open source | 2009 | N | N | MySQL support for storing historical data |
| **JCover** | Java | Commercial | 2006 | N | N | Coverage differencing |
| **Jtest** | Java | Commercial | 2014 | Y | Y | Built-in and customised rules Code reviews management |
| **PurifyPlus** | C,C++, Basic.net | Commercial | 2007 | N | Y | |

Table 3.2: Additional features of code coverage tools

## 3.3 Measuring Axiomatisation Coverage

The authors of [5] implement a framework that automatically executes test cases for verification systems as well as calculating the axiomatisation coverage for the given test suite. In addition, it uses five different variants of the breadth-first and depth-first search approaches to increase the axiomatisation coverage. The five approaches improve the coverage by 18

To compute the axiomatisation coverage for a test case $P+(REQ \cup AUX)$, the procedure is as follows. In the first step, the verification tool finds a proof for the test case using the complete axiomatisation base available. The framework gathers information on resource consumption for each proof attempt, such as number of proof steps and time needed. In addition, information on which axioms are used to find the proof are recorded as set $T$.[1]

In the next step, the iterative reduction phase starts. In the reduction step, the framework starts from an empty set $M$ of *covered* axioms –sometimes called minimal set or mandatory set. For each axiom $t$ in the set of initially used axioms $T$, an attempt is made to prove the test case using all axioms in $T$ except $t$, i.e. $M \cup (T \setminus \{t\})$. If the proof does not succeed, $t$ is considered to be necessary and it is added to the set $M$. Then, the axiom $t$ is removed from $T$ and the framework starts the next iteration until $T = \varnothing$.

After each single iteration of this computation, the current set of axioms $M$ is only an approximation of the coverage of the test case, as not every applied axiom was necessarily crucial for the proof process (for example, axioms can depend on each other). The above procedure is repeated with $M$ as input as long as the result

[1] "Used" does not imply that the application of the axiom was necessary to find the proof.

is different from the input. Eventually, this fixed-point algorithm finds the true minimal set of axioms necessary to construct the proof.

## 3.4 Current Framework Analysis

In this section, we briefly discuss the current framework for measuring the axiomatisation coverage implemented by [5].

### 3.4.1 Maximising The Axiomatisation Coverage

The test coverage is increased by creating additional tests from the existing test cases. This can be achieved by preventing the verification system from using certain parts of the axiomatisation base. Thus, the system is forced to find alternative ways to construct a correctness proof for a given test case $P{+}(REQ \cup AUX)$ using only a subset of the total set of axioms. This subset of allowed axioms is referred to as the whitelist $WL$. As a result of using such a mechanism, the test case changes to a tuple of $\langle P{+}(REQ \cup AUX), WL\rangle$, of a program $P$ with a requirement specification $REQ$ and auxiliary annotations $AUX$, and a whitelist $WL$.

The introduction of whitelists makes it possible to reuse existing test cases, which can be achieved by modifying the $WL$ for each program and its specification. This is a big advantage over writing new test cases, which is a very time consuming process, even for experienced verification engineers. Nevertheless, it cannot fully replace the need to extend test suites through additional test cases.

For instance, let us consider axioms for bitwise XOR-operations or for certain simplifications of inequalities. If a test suite does not contain test cases that

include such characteristics, those axioms will probably not be exercised using that test suite even though many parts of the axiomatisation base will be reused many times.

### 3.4.2 The Search Approaches

The authors of [5] implement five different search algorithms to explore axioms in the whitelist $WL$. In the first approach, axioms in $M$ are chosen to be dropped from $WL$ in a depth-first fashion. A random depth-first method is used where the axioms in $M$ are picked in a random order. Another technique is also used: depth-first random step sizes. It randomly selects up to six axioms from $M$ to be dropped from $WL$. When a proof cannot be constructed using the current $WL$, the step size is reduced by 50%. The last two search methods are the breadth-first and random breadth-first approaches. The former explores the axioms in $M$ in a breadth-first fashion, whereas the latter randomly picks them to be dropped from $WL$.

### 3.4.3 Issues

After analysing the current framework and its test log files, several major points were observed. The search space is quite large, i.e. $2^n$, where $n$ is the number of axioms in the axiomatisation base. In addition, its valid solutions are notably scattered. In other words, roughly 5-10% of a valid solution's neighbours can be considered valid solutions as well [29]. Furthermore, when using a heuristic methodology, increasing the randomness in an unsystematic fashion reduces the chance of finding a valid solution; moreover, it fails to guide the process of maximis-

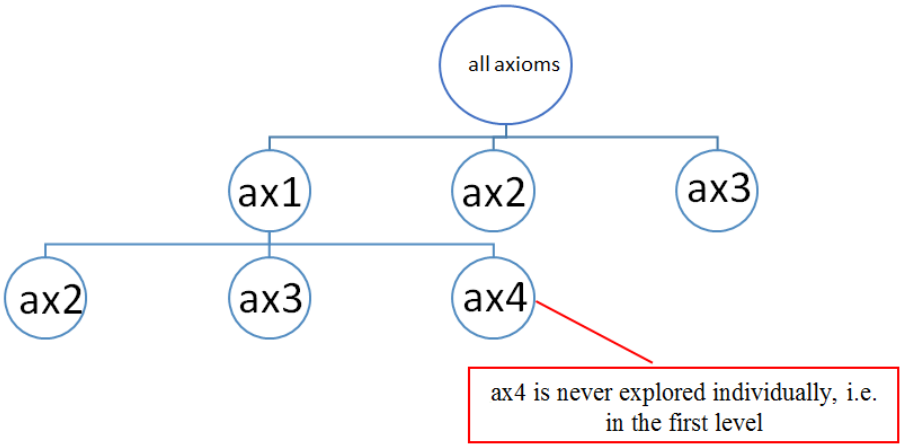



Figure 3.1: Discarding some individual axioms in the current framework.

ing the axiomatization coverage. Although the third search approach, depth-first random step sizes, increases the overall axiomatization coverage, the failure rate is considerably high [29]. This high rate of failure is likely occurs due to the nature of the search space, which requires a more structured mechanism to heuristically explore it.

In addition, the log files indicate that some instances of replaced axioms were not chosen to be dropped individually. In other words, after the first level of the breadth-first search approach, newly-discovered axioms are not chosen for individual dropping; instead, the approach explores it within the current level. For example, assume the first minimal set $M1 = \{ax1, ax2, ax3\}$ is used to find a proof for a test case. The approach chooses each axiom of $M$ to be replaced individually. In the case where *ax1* is successfully replaced by *ax4*, and the remaining axioms cannot be replaced, the second minimal set *M2* will be $\{ax2, ax3, ax4\}$. In the following step, the approach will explore the next level of the search space, which is the second level. The next set of axioms to be dropped is $\{(ax1, ax2), (ax1, ax3), (ax1, ax4)\}$; figure 3.1 illustrates this example. As can be seen, the approach discards choosing ax4 to be dropped individually. Consequently, it reduces

the number of explored nodes, which might misguide the search for more axioms to be covered.

## 3.5 Summary

For measuring the code coverage, there are several tools that can use static and/or dynamic instrumentation. They produce reports in different styles: file-based styles and GUI styles. The reports includes statement, branch, method and/or class coverage.

In order to measure the axiomatisaion coverage, only the needed axioms must be counted. In other words, the initial set of axioms $T$ used in a proof construction needs to be reduced. This can be achieved by dropping one axiom a time from the and reprove the same test using the modified $T$‘. If $T$‘ successfully proves the test case, that means only needed axioms remain in the $T$‘.

For maximising the axiomatisation coverage, the verification system is prevented from using some parts of the axiomatisation base. This can be accomplished by dropping some axioms from it. As a result, the system is forced to find alternative ways to prove the given test case. Five different variants of breadth-first and depth-first approaches are used to choose which axiom to be dropped. However, these approaches are not informed, and therefore, the chances of dropping an appropriate axiom is relatively high. As a consequence, valid solutions are ignored.

# Chapter 4

# The Proposed Approach

## 4.1 Introduction

In this section, we present our heuristic approach that uses a learning process to guide the search.

## 4.2 Guidance Table

To efficiently handle such search space, we propose a new approach that uses a Guidance Table (GT) to guide and inform the search. In this section, the GT and its uses are described.

### 4.2.1 Description

The GT consists of the following: each found axiom in all minimal sets, its replacement sets if it is successfully replaced, the total number of successful uses for each replacement set, the total number of all successful replacements and the total number of unsuccessful replacements. All data is recorded for each axiom. Table 4.1 illustrates an example of the guidance table. It shows that *ax3* was replaced successfully four times: once by {*ax5*, *ax6*} and three times by {*ax7*}. It also shows that {*ax3*} was unable to be replaced a single time.

| **Axiom** | **Replacement Set** | **Successful Times** | **Total of Successful Times** | **Total of Unsuccessful Times** |
|---|---|---|---|---|
| ***ax1*** | {*ax4*} | 1 | 1 | 0 |
| ***ax2*** | | 0 | 0 | 1 |
| ***ax3*** | {*ax5*, *ax6*} | 1 | 4 | 1 |
| | {*ax7*} | 3 | | |

Table 4.1: Guidance table example

In addition, this GT can be built off-line and on-the-fly. Initially, we implemented the GT tool to generate guidance tables for each test case in the off-line mode. Test log files for the current framework are used to build the GTs. Moreover, it can merge all test cases' GTs into one GT to make the search more accurate.

### 4.2.2 Guidance Table Uses

#### 4.2.2.1 Log File Analysis

The major purpose of using the GT is to find equivalences between axioms.[1] In other words, it lists equivalent sets of axioms for each axiom or axiom sets. For example, Table 4.1 depicts that ax3 has two equivalent sets of axiom, which are $\{ax5, ax6\}$ and $\{ax7\}$. It is worth mentioning that these two sets are only equivalent to that axiom in four cases as it is shown in the table. Nonetheless, as they are not completely equal to ax3, the first set could not replace ax3 in one instance. What is more, the GT helps to find inconsistent cases. For instance, it has been successfully used to identify the issue of escaping axioms in the current approach, as mentioned in the previous section.

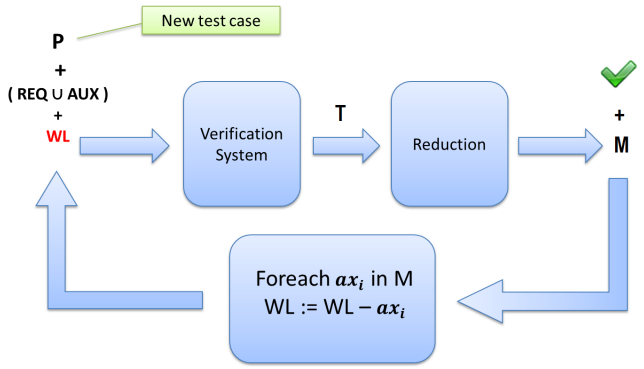
#### 4.2.2.2 Guiding the Search

In addition, discovering relationships between axioms enables the proposed technique to find axioms that have a relatively large number of equivalent sets to guide the search. Since our proposed method is based on the beam search technique, which requires information regarding the search space, it is essential to construct a table that can be used to inform and guide it towards promising nodes. Moreover, the GT can identify irreplaceable axioms that have not been successfully replaced. Avoiding these axioms improves the performance of the search process and the framework.

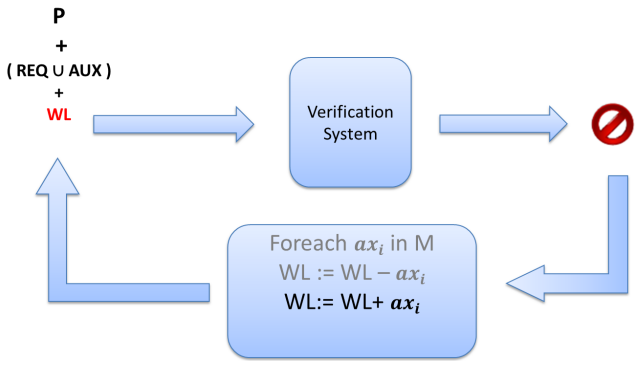
[1] "Equivalence" is not strictly logical here, but regarding the tool's capability to find a proof in a different way.

## 4.3 Test Case Generation

It is essential to increase the number of test cases in order to maximise the axiomstisation coverage. Nevertheless, generating more test cases manually is a difficult task, even for an experienced test engineer. To overcome this issue, the existing test cases have to be reused in a special manner to automatically generate additional test cases. In this research, we adapt the mechanism of maximising the axiomatisation coverage in [5], as discussed in Section 3.4.1.




(a) A proof is found.




(b) A proof is not found

Figure 4.1: Test case generation

Figure 4.1 illustrates the process of test case generation, where Figure 4.1a shows an instance of finding a successful proof and Figure 4.1b presents an example of an unsuccessful attempt. As can be seen, the verification system runs on the given test case to find a proof. In a case where the system successfully proves the correctness of the test case, it reduces the initial set of used axioms $T$ in the proof to the set of covered axioms $M$. In the next step the framework picks an axiom $m_i$ from $M$ and drops it from the available whitelist $WL$. Hence, a new test case is created.

After generating the new test case where the verifier runs it again. In case the tool cannot prove it using the new $WL$, the last axiom $m_i$ is returned to the $WL$ and the next axiom $m_i$ is chosen.

### 4.3.1 Improved Test Generation

According to [5], the reduction phase is a time consuming process that can take several minutes and sometimes even hours. Therefore, we enhance the test case generation by making it faster through additional steps. Two independent approaches have been implemented: one predicts which axioms to be dropped during the reduction phase, while the other tries to skip the whole phase.

### 4.3.2 Reduction by Prediction

In a preliminary experiment we have conducted in [6], it was observed that some axioms tend to be dropped frequently. To some extent, this can help to predict which axioms are more likely to be excluded in the reduction phase. As a result, the framework can be improved by extracting those axioms before the phase starts,

as it takes sometimes minutes up to hours to evaluate each set after each reduction iteration.

The used procedure is illustrated in Figure 4.2 and is as follow. At the beginning, the GT tool creates the axioms' removal frequency table. It contains each found axiom in previous test runs and the total number of how many times it was identified as *"not needed"*. In a new test run, the verification tool runs the test case. In the case where the tool finds a proof, the initial set of axioms $T$ is explored, before the reduction phase starts. Each axiom $t_i \in T$ that has a removal frequency greater than a predefined threshold will be removed from $T$. Finally, after examining all *T's* elements, the reduction phase starts on the remaining set of axioms.

The threshold is defined using a greedy technique. Initially it starts with a small number for removing a large number of axioms as possible. Nevertheless, removing "too many" axioms may sometimes fail the proof procedure. This is because some of them turned to be mandatory axioms in some test cases. Thus, the threshold increases gradually to overcome this issue.

### 4.3.3 Using a Hash Table

In order to reduce the computation time, we utilise some additional information obtained by the $GT$ tool from previous test runs. For each test case $TC$, the GT tool collects and stores all of the initially used axiom sets $T$ and their reduced mandatory sets of axioms $M$. In addition, it arranges these sets to speed up the whole testing process. This can be done by mapping each $TC$ and $T$ to their corresponding $M$. As a result, the tool generates a hash table where the keys are pairs of $\langle TC, T \rangle$ and the value is $M$. Table 4.2 shows an example of the hash

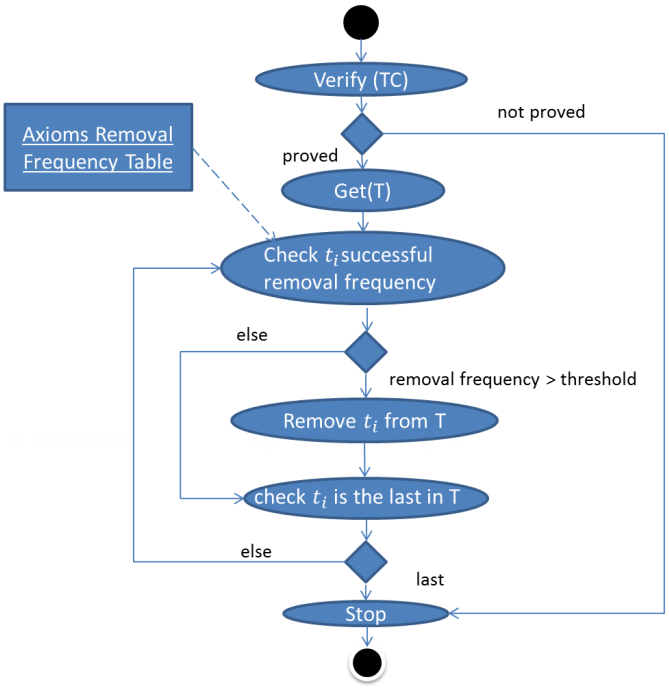



Figure 4.2: Reduction by prediction.

table.

By introducing this hash table method, the framework can skip the reduction phase for any previously found $T$ for the test case in progress. Figure 4.3 shows the improved process. After the verification tool finds the initial used set of axioms $T$, it checks whether $T$ is found in the hash table, if so, it retrieves its corresponding $M$ and tests it to ensure its validity, otherwise, the tool starts the reduction phase. It is worth mentioning that testing the retrieved $M$ is important, since the verification tool may undergo some modifications that affect the proof procedure.

| Key<$TC$, $T$> | | Value<$M$> |
|---|---|---|
| Test Case | Used Set of Axioms $T$ | Mandatory Set of Axioms $M$ |
| $tc_1$ | $r_1$ | $m_1$ |
| $tc_1$ | $r_2$ | $m_2$ |
| $tc_2$ | $r_3$ | $m_3$ |
| .... | .... | .... |
| $tc_i$ | $r_j$ | $m_k$ |

Table 4.2: Mapping each test case and each initially used set of axioms to a mandatory set.

## 4.4 Beam Search Approach

Algorithm 1 illustrates our informed beam search. As can be seen at the first stage, the GT is initialised and sorted by the values of the total successful replacement times. Then, the initial set of used axioms $T$ for proving the test case $TC$—in the form of $\langle P+(REQ \cup AUX), WL\rangle$—is obtained by running the verification tool on $TC$. If no proof can be constructed, the method terminates. One the other hand, in the case of a proof can be constructed, we reduce the set $T$ to a minimal set $M$.

In the next stage, the $GT$ is used to fill the promising node list that is used for selecting the best nodes to explore. It includes all axioms that are found in $M$ as well as in $GT$, nonetheless, in some cases, an axiom may not be found in the $GT$, which means it is a newly covered axiom. Furthermore, these axioms must have relatively high successful replacement rates. Finally, axioms that have not been

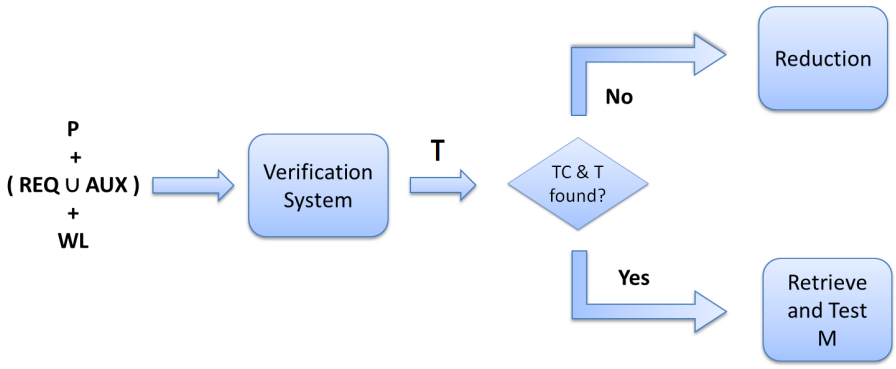



Figure 4.3: Improved test case generation.

replaced are stored in the discarded list to be avoided in the search process.

In the subsequent stage, where the axioms are not found in the $GT$, the method adds them to the promising node list. This step is considered to guarantee that all new axioms have to be dropped from the $WL$. As a result, new equivalent axioms may get covered, which increases the chances of maximising the coverage.

In the final stage, the promising list is sorted by each axiom's successful replacements in descending order, then the method starts exploring the promising axioms. We do this by dropping[2] one axiom at a time from the available axiomatization base (i.e. $WL$), and then re-running the proof procedure using the shorter $WL$ (Step 1). As a consequence, a new test case is generated.

Creating test case using such a mechanism can lead to an increase in the overall axiomatisation coverage. Preventing the verifier tool from using the complete axiomatisation base will force the tool to find alternative ways "*axioms*" for constructing a new correctness proof. Thus, new axioms are used, which increases the coverage.

[2]Drooping an axiom does not mean deleting it from the axiomatisation base in this context. It means forbidding the verifier from using it to construct the new proof.

**Algorithm 1:** BeamSearch

**Data**: GT: guidance table (sorted by successful replacements.)

**Data**: TC: test case $\langle P+(REQ \cup AUX), WL \rangle$.

**Data**: T: initially used set of axioms during the proof.

**Data**: M: minimal set of axioms needed for finding the proof.

**Data**: WL: axiomatization base used by the verification tool.

**Data**: PromisingNodeList: list containing the most promising nodes.

**Data**: DiscardedList: list containing the discarded nodes.

**Result**: union of all minimal lists.

```
1  T = Run(TC);                                   /* run the verification tool */
2  if TC is not Proved OR Timeout then
3      Stop;
4  else
5      M = Reduce(T);
6      Add M to result;                           /* add the newest minimal list */
7      foreach axiom in M do
8          if GT contains axiom then
9              if axiom in GT has total successful time greater than 0 then
10                 Add axiom to PromisingNodeList;
11             else
12                 Add axiom to DiscardedList;
13             end
14         else
15             Add axiom to PromisingNodeList;   /* adding new axiom, since it has
                                                    not been in the GT so far */
16         end
17     end
18     sort(PromisingNodeList);                  /* by total replacements (descending) */
19     foreach axiom in PromisingNodeList do
20         Drop axiom from current WL;
21         Repeat from Step 1;
22     end
23 end
24 return result;
```

## 4.5 Fast Beam Search Approach

The BeamSearch$^{\text{FastMinSet}}$ approach has two main parts: (1) it effectively tries to quickly re-discover the previously found minimal sets $M$, and (2) it constructs the promising list to inform the search. Algorithm 2 illustrates only the first part, as the second part (i.e. building the promising list) has already been discussed in Section 4.4.

As can be seen, the set $T$ of axioms used in a proof is obtained by running the verification tool on the test case $TC$ using the whitelist $WL$. Additionally, the approach looks for the corresponding minimal set $M$ from the hash table $HT$, which maps each $TC$ and $T$ to $M$.

In the next stage, when $M$ is found, the BeamSearch$^{\text{FastMinSet}}$ approach reruns the tool, but this time the $WL$ is replaced by the corresponding $M$[3] to ensure the validity of $M$. It is worth mentioning that we add this step, as the verification tool may undergo some modifications that affect the proof procedure. Then, the BeamSearch$^{\text{FastMinSet}}$ approach uses the valid $M$ for building the promising list. However, in case the $HT$ does not contain such $T$ (i.e. it is new, or $M$ is not valid), the approach continues its job as BeamSearch, by reducing $T$ to a new minimal set $M$ and then constructs the promising list.

It can be noted that using BeamSearch$^{\text{FastMinSet}}$ significantly reduces the testing time by eliminating the time needed for reducing $T$ to $M$. Although BeamSearch$^{\text{FastMinSet}}$ runs the verification tool twice, it is still considerably faster than the complete reduction of $T$ to the minimal set $M$, since the later can require dozens or even hundreds of verification attempts.

[3] which was a successful reduction at least once before

**Algorithm 2:** BeamSearch$^{\text{FastMinSet}}$

**Data**: HT: hash table ((TC, T), M).
**Data**: TC: test case $\langle P{+}(REQ \cup AUX), WL\rangle$.
**Data**: T: set of axioms initially used during the proof.
**Data**: M: minimal set of axioms needed for finding the proof
**Data**: WL: axiomatization base used by the verification tool
**Result**: union of all minimal lists

```
1  T = Run(TC);                                  /* run the verification tool on TC */
2  if TC is not proved OR Timeout then
3  |  Stop;
4  else
5  |  if HT contains ⟨TC, T⟩ then
6  |  |  M = Get M from HT by ⟨TC, T⟩;
7  |  |  WL = M;
8  |  |  T = Run(TC) ;                          /* rerun to ensure M is valid */
9  |  |  if TC is proved then
10 |  |  |  Add M to result;                    /* add the newest minimal list */
   |  |  |  /* construct promising and discarded lists as in BeamSearch   */
11 |  |  else
   |  |  |                                      // TC is not proved, run BeamSearch
12 |  |  end
13 |  else
   |  |                                  // HT does not contain ⟨TC, T⟩, run BeamSearch
14 |  end
15 end
16 return result;
```

## 4.6 Summary

This chapter discussed our methodologies for improving the process of generating test case and maximising the axiomatisation coverage. Two methods are implemented for fastening the reduction phase. The first one predicts which axioms are mostly likely will be unneeded before the starting of the reduction phase. The second method utilises a hash table. The key for this hash table consists of each test case $TC$ and each initial set of axioms used to prove that test case $T$, and its value is the mandatory set of axioms $M$ for that $TC$.

For maximising the coverage, a guidance table is utilised to inform two search methods: BeamSearch and BeamSearch$^{\text{FastMinSet}}$. The guidance table contains special information collected from the test logs. The difference between the search methods is the BeamSearch$^{\text{FastMinSet}}$ uses the hash table method to reduces the reduction time. The main idea of using the guidance table and these search approaches is to navigate the solution space effectively. The two methods expand the search space by visiting the promising nodes, which are identified by the guidance table.

# Chapter 5

# Evaluation

## 5.1 Introduction

## 5.2 Case Study: KeY System

As the target for our case study, we have chosen the KeY tool [4], a verification system for sequential Java Card programs. It was designed to integrate design, implementation, formal specification and formal verification of a software product as efficiently as possible. The main purpose of using KeY for complex tasks is to minimise the cost of applying formal methodologies to an reasonable level [4].

The core component of the KeY system is the theorem prover for program logic [1]. It consists of a variety of automated inference techniques, such as symbolic execution of programs, first order reasoning, arithmetic simplification and first order Java Dynamic Logic (Java DL), which can be considered a generalisation of Hoare logic. In addition, unlike other verification tools, it combines all these

techniques to find a proof.

In KeY, the Java Modeling Language (JML) is used to specify the properties about Java programs. JML is a powerful and popular specification language for Java software [10]. It is based on the design by contract paradigm and therefore it consists of pre- and post-conditions and class invariants. In addition, it provides auxiliary annotations such as loop invariants. Furthermore, it supports modular verification, i.e. proving one Java method at a time.

Following, we will briefly describe the workflow of the KeY system. Assuming one method is chosen to be verified against a single pre-/post-condition pair. At the beginning, the relevant parts of the Java program and its JML annotations are translated into a sequent in Java DL, a multimodal predicate logic. Validity of this sequent implies that the program is correct with respect to its specification [7]. Proving the validity is done using the automatic proof strategies within KeY, which apply sequent calculus rules implemented as so-called *taclets*.

The set of taclets provided with KeY captures the semantics of Java. Additionally, KeY contains taclets that deal with first order logic formulas. The development version of KeY as of August 16, 2012 contains 1,520 taclets and rules; we will refer to these as *axioms* in the remainder of this article to facilitate reading. Note that not all axioms are always available when performing the proof, as some exist in several versions, depending on proof options chosen.

The result of a verification attempt in KeY is one of the following: the generated Java DL formula is valid and KeY is able to find a correctness proof; or the generated formula is not valid and the proof cannot be closed; or KeY runs out of resources.

```
1 \programVariables { int a, b; }
2 \problem{
3 \< {
4     try{
5       b=a/a;
6     }
7     catch(Exception e){
8       b=1;
9     }
10 } \>
11 b=1
12 }
```

Figure 5.1: KeY's test case example.

The test case in KeY is same as discussed in 2.5.2, which is the form of $P+(REQ \cup AUX)$. For example, let us consider the following test case for KeY in Figure 5.1. It contains a simple Java program code in Lines 4-8, which represents $P$. The post condition in line 11 represents the post-condition $(AUX)$. The test goal is to check if KeY correctly deals with a *division by zero* when a number $a$ is divided by itself.

## 5.3 Experiment

In this section, we will first describe the experimental setup. We will briefly look into the information that our beam search uses before presenting and analyzing the coverage results.

### 5.3.1 Experimental Setup

Our testing framework automatically executes the 319 test cases mentioned above and measures the axiomatization coverage[1]. We also use the Emma tool version 2.0 to measure the code coverage [24]. It is worth mentioning that this section discusses two separated experiments, one for each coverage criterion.

The KeY source distribution provides a test suite containing 319 test cases (as of August 16, 2012) for testing the verification of functional properties. The complexity of these test cases ranges from simple arithmetic problems to small Java programs testing single features of Java, up to more complex programs and properties taken from software verification competitions.

This test and all subsequent runs are performed on Intel Xeon E5430 CPUs (2.66GHz), on Debian GNU/Linux 5.0.8, with Java SE RE 1.7.0. The internal resource constraints are set to twice the amount of resources needed for the first proof run recorded initially. This allows for calculating axiom coverage in reasonable time and ensures comparability of coverage measures between computers of different processing power. Still, the computation of a single fix-point typically takes minutes, and in a few cases even hours. Therefore, we limit the computation time for each of the 319 test cases to 24 hours for each approach, which means an investment of 0.87 CPU years per approach.

[1]The full code and the logfiles are available online `http://cs.adelaide.edu.au/~optlog/research/software.php`.

### 5.3.2 Preliminary Experiment

We conducted a preliminary experiment, to resolve the issue of skipping some axioms in the breadth-first search approach used in [5]. The implemented methods are as follow:

1. A one-level breadth-first search method, that only explore individual axioms.
2. A complete breadth-first search method.
3. A naive search method that tries to drop all axiom in the whilelist.
4. The BeamSearch approach that uses the reduction by prediction method.

In addition, the log files from this experiment helped us to construct an accurate guidance table that includes all possible individual axioms, and to conduct an in-depth analysis that guided us to design and implement our hash table method that minimises the reduction phase and improves the overall framework performance.

#### 5.3.2.1 Results

| Appraoch | First Minimal Sets | Total Covered Axioms |
|---|---|---|
| naive search | 610 | 630 |
| one-level breadth-first search | 617 | 665 |
| complete breadth-first search | 596 | 666 |
| BeamSearch with reduction by prediction method | 612 | 691 |

Table 5.1: Preliminary axiomatisation coverage. The *first minimal sets* refer to those found first by the approaches, which initially use all 1520 axioms.

Table 5.1 presents the axiomatisation coverage achieved by the four approaches. As expected, the BeamSearch with reduction by prediction method outperformed

the other three approaches by covering 691 axioms. It effectively covered higher number of axioms, owing to the use of guidance table and reducing the reduction time. By minimising the processing time in the reduction phase, the BeamSearch can expand more nodes within the search space.

Surprisingly, the number of covered axioms by the complete breadth-first search and one-level breadth-first search approaches are almost the same. The former was expected to cover significantly more axioms, as it managed to explore seven levels of the solution space. This is because single axioms tend to be successfully replaced more than sets of axioms, which in turn increases the chances of covering new axioms. Furthermore, the breadth-first search mechanism expands the search space blindly (i.e. works with no information regarding the solution space), which increases risk of wasting the time limit *" 24 hours per test case"* by exploring the nodes that do not contribute to the coverage.

### 5.3.3 Major Experiment

We compare our informed beam-search approach to the different variants of uninformed breadth-first and depth-first approaches reported in [29]. There, the approaches followed steps similar to BeamSearch, however, the approaches are not informed. While we make use of information learned in Step 18 of BeamSearch, the other approaches explore the list in either breadth-first or depth-first fashions.

Given a minimal set of axioms $M$ for a given pair $P{+}(REQ{\cup}AUX)$, the approaches try to remove axioms $m \in M$ from the current whitelist $WL$ (first iteration: all 1,520 axioms). If the subsequent verification of $\langle P{+}(REQ \cup AUX), WL\rangle$ is successful, then the verification system has found an alternative path to prove the

correctness of the $P$. Consequently, a new minimal set $M'$ can be found, which will only contain the elements that are in $WL$, and it will contain previously *uncovered* axioms. For the next iteration, $M'$ will be the starting point. Effectively, we iteratively check if some axioms can be replaced by others.

#### 5.3.3.1 Guidance Table Analysis

Before our experiments, we built the guidance table for our beam search based on re-runs of the five approaches in [29] as well as the approaches in the preliminary experiment. This GT contains, amongst others, the following interesting information. This step is mandatory, as otherwise our approach defaults to a simple breadth-first search.

First, let us look at individual axioms. In Figure 5.2, we show for all covered axioms the number of times that they have been successfully and unsuccessfully replaced. The use of this data is not straightforward, since there is no specific pattern. For instance, one of these axioms has 188 unsuccessful replacements, while it is successfully replaced 36 times. Such axioms have to be moved towards the end of the promising list, as, more often than not, they appear to be dead ends. On the other hand, one axiom is successfully replaced 143 times, while it is unsuccessfully replaced only three times. This makes it a good candidate for the beam search.

Once an axiom is replaced, we can often see that it is replaced by a set of two axioms or by even larger sets. Figure 5.3 shows how many different single axioms are replaced by a set of axioms: the y-axis represents the number of sets while the x-axis represents the replacement sets' sizes. For example, 446 pairs of axioms successfully replace a single axiom, and there is one case where one axiom is

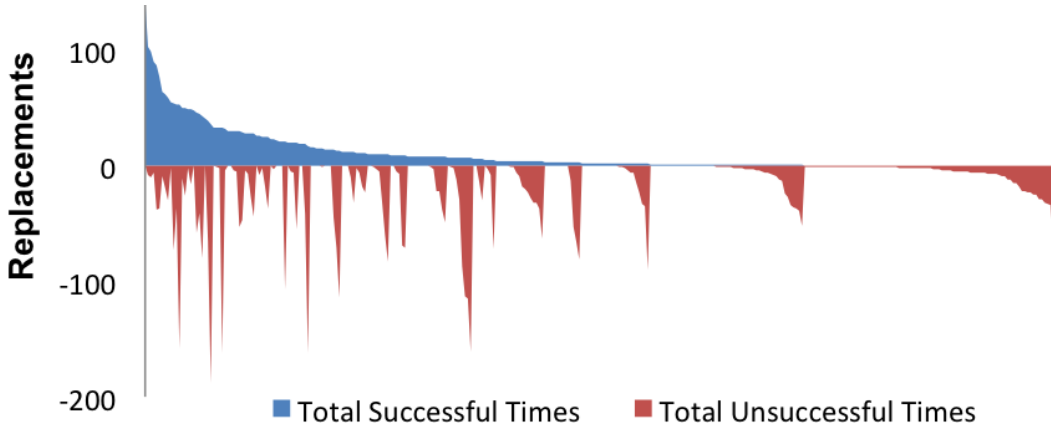



Figure 5.2: Successful vs. unsuccessful replacements for each single axiom, shown as positive and negative values. The axioms (along the $x$-axis) are sorted in decreasing order according to the number of successful replacements.

replaced by an enormous set of 63 axioms. In future work, we can study such cases in order to identify equivalent sets of axioms. Moreover, they can help us to improve the framework by restricting the number of times that we replace that one axiom in the future, as doing so may decrease the execution time for the proof procedure.

#### 5.3.3.2 Code Coverage Results

We ran KeY on the 319 test cases and measured the code coverage using the EMMA tool version 2.0 [24]. It is worth mentioning that the reduction phase was disabled during the test. As a result, the number of axioms used to prove the test cases is slightly different to those in the first minimal sets shown in Table 5.4.

Achieving high code coverage in software testing is of great importance in evaluating the test suite. Nevertheless, in verifying deductive verification systems, our

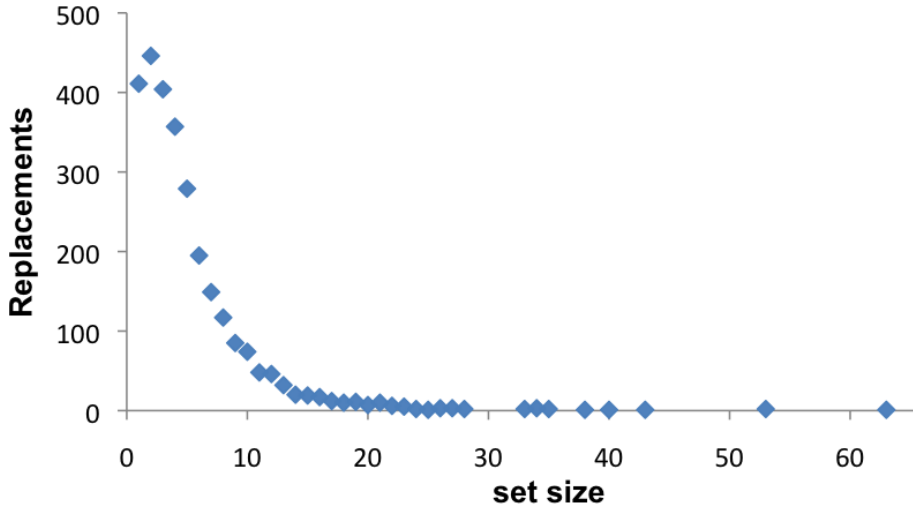



Figure 5.3: Replacement set's sizes and number of replacements for single axioms BeamSearch$^{\text{FastMinSet}}$.

results show that the axiomatization coverage is essential. This is because although in 295 test cases the LOC coverage is more than 34%, the axiomatization coverage is less than 10%; moreover, 41 test cases have less than 1% of axiom coverage. Table 5.2 shows a summary of the coverage details for the test cases.

It can be seen that the average LOC coverage is 37%, in contrast, the average coverage in the axiomatisation is only 4.43%. Most of the test cases exercised nearly the same proportion of the code, as the standard deviation for the LOC is relatively small 2.5%. On the other hand, there are fluctuations in the amount of axioms covered by the test suite. This is due to the logical properties within each test case.

In addition, some test cases managed to exercise 89% and 44% of the class and LOC coverage, respectively, which is the maximum LOC coverage; nonetheless, they could not cover even 17% of the axioms individually. In short, there is no clear correlation between the exercised code and the axioms used in the proof

| Test Cases | Code Coverage | | Axiomatization Coverage | |
|---|---|---|---|---|
| | Class Coverage | Line Coverage | Number of Axioms | Axiom Percent |
| standard_key-java_dl-arrayUpdateSimp | 87% | 35% | 3 | 0.20% |
| .... 182 test cases .... | | | | |
| heap-SmansEtAl-Iterator_list | 83% | 40% | 62 | 4.08% |
| .... 134 test cases .... | | | | |
| heap-list-ArrayList_concatenate | 85% | 44% | 255 | 16.78% |
| min/max | 81%/89% | 34%/44% | 3/255 | 0.2%/17% |
| $\text{mean}_{\text{standard deviation}}$ | $85\%_{1.9\%}$ | $37\%_{2.5\%}$ | $67_{52}$ | $4.43\%_{3.4\%}$ |
| union | 90% | 51% | 691 | 45% |

Table 5.2: Code coverage vs. axiomatization coverage. Sorted by axiom percentage in descending order.

| Class | Method Coverage | Line Coverage |
|---|---|---|
| Taclet | 66% | 62% |
| TacletBuilder | 61% | 47% |
| Proof | 54% | 50% |
| CompoundProof | 0% | 0% |

Table 5.3: Some classes within KeY tool.

construction stage.

The overall axiomatization coverage—as expected—is low at only 45%. Additionally, the LOC coverage is only 51%, which is significantly less than the 85% recommended by the software testing literature (see e.g. [31]). Though the class coverage reached 90%, after we analysing the coverage outputs using the EMMA tool, we observe that many classes are only partially covered. This includes classes that appear to be crucial for the proof procedure: the LOC coverage there ranges from 62% down to even 0% (see Table 5.3 for some examples).

#### 5.3.3.3 Axiomatization Coverage Results

The coverage statistics of the different approaches are listed in Table 5.4. The number 611 represents the result of the naive approach, where the full set of 1,520 axioms is used and no alternatives are sought. This is our base value.

We start with some general observations. First, each of the individual approaches improves the total coverage over the first minimal sets by about 12–18% each. Our BeamSearch$^{\text{FastMinSet}}$ approach achieves the highest individual improvements among the approaches.

When considering all the approaches together, the initial coverage of about 611 axioms has been increased to a total of 755 axioms through the use of whitelists. This means that all approaches together has improved the achievable coverage autonomously by about 24%, and that is without requiring a verification engineer to write a single new test case.

It is an interesting coincidence that our BeamSearch$^{\text{FastMinSet}}$ achieves a total coverage of 722 axioms, which is identical to the coverage achieved by the first five approaches together. As we can see via the 755 axioms that are covered by the union of all seven approaches, our beam search not only cover most of what the first five approaches cover, but it also covers additional 33 axioms. It appears that the combination of the GT and the fast reduction (when available) allows it to search more effectively than the previous approaches.

Let us now investigate the differences between the approaches. First, by using our GT tool, we obtain the number of times that a single axiom is successfully replaced. The results are shown in The result presented in Table 5.5 clearly show the structural differences between the approaches. For example, the depth-first

| axioms covered in | Depth-First Search | Random Depth-First Search | Greedy | Breadth-First Search | Random Breadth-First Search | Union Approaches 1–5 | BeamSearch | BeamSearch$^{\text{FastMinSet}}$ | Union of all seven approaches |
|---|---|---|---|---|---|---|---|---|---|
| the first minimal sets | 611 (40%) | 611 (40%) | 610 (40%) | 613 (40%) | 609 (40%) | 615 (40%) | 611 (40%) | 612 (40%) | 638 (42%) |
| all minimal sets | 701 (46%) | 699 (46%) | 688 (45%) | 687 (45%) | 684 (45%) | 722 (48%) | 692 (46%) | 722 (48%) | 755 (50%) |

Table 5.4: Coverage statistics. The results for Approaches 1–5 are based on reruns from [3].

approaches 1-3 have the smallest number of replacements of single axioms, which is expected given their nature: they explore shorter whitelists first. The breadth-first approaches 4-5, on the other hand, achieve significantly higher single replacements, because they explore the replacement of single axioms first. Our beam search achieves the highest number of replacements here, as it prefers the replacement of single axioms, and considers single axioms when it comes across new ones in the search. This has the advantage of utilising the guidance table; shorter keys are more likely to be existent, and therefore of help. We will see this in the following.

Next, we obtain the number of equivalent sets found by each approach. Table 5.6 presents the total *attempts*, as well as the number of equivalent sets and their percentages. As can be seen, the success rate is in favours BeamSearch with 44%. On the other hand, BeamSearch$^{\text{FastMinSet}}$ is the fourth at 21%, it comes after the Random Breadth-First search and Breadth-First search with 37% and 34%, respectively. This is because BeamSearch$^{\text{FastMinSet}}$ eliminates the reduction time for finding the minimal sets $M$, which in turn enables it to spend

| | successfully replaced single axioms | unsuccessfully replaced single axioms | total | % of successfully replaced axioms |
|---|---|---|---|---|
| Depth-First search | 29 | 230 | 259 | 11% |
| Random Depth-First search | 26 | 235 | 261 | 10% |
| Greedy | 21 | 218 | 239 | 9% |
| Breadth-First search | 181 | 259 | 440 | 41% |
| Random Breadth-First search | 193 | 267 | 460 | 42% |
| BeamSearch | 211 | 97 | 308 | 69% |
| BeamSearch$^{\text{FastMinSet}}$ | 231 | 90 | 321 | 72% |

Table 5.5: Successful vs. unsuccessful replacements: unique single axioms.

more time exploring the search space. In addition, it is worth mentioning that the number of the total attempts represents the sizes of each approach's generated GT, which shows that a large amount of information for the beam search is extracted.

In contrast, the number of equivalent sets for BeamSearch$^{\text{FastMinSet}}$ is the largest amongst the algorithms. Moreover, there is a significant difference between our beam search approaches and the breadth-first approaches Random Breadth-First search and Breadth-First search, which achieve the fourth and fifth highest numbers of equivalent sets. In total, all 74,219 unique test cases created by all approaches are stored and are ready to be used for regression testing, currently achieving a coverage that is 24% higher than that achieved by the 319 original test cases.

Lastly, Table 5.7 shows the number of total replacements for all axioms made by each search technique. As can be seen, unlike the all depth and breadth methods, the BeamSearch$^{\text{FastMinSet}}$ and BeamSearch help us to identify more logical relationships between the axioms. For example the former successfully finds 58,982

| | equivalent sets | total | % of equivalent sets |
|---|---|---|---|
| Depth-First search | 7,544 | 132,735 | 6% |
| Random Depth-First search | 3,880 | 40,446 | 10% |
| Greedy | 2,458 | 80,886 | 3% |
| Breadth-First search | 9,037 | 26,733 | 34% |
| Random Breadth-First search | 10,784 | 28,842 | 37% |
| BeamSearch | 33,178 | 75,180 | 44% |
| BeamSearch$^{\text{FastMinSet}}$ | 54,821 | 258,319 | 21% |

Table 5.6: Analysis: Equivalent sets found by each approach.

replacements for different axioms which is greater than the total number of successful replacements of the first five approaches combined together. This is due to the capability of BeamSearch$^{\text{FastMinSet}}$ to eliminate the reduction time. Moreover, it clearly shows that even though BeamSearch$^{\text{FastMinSet}}$ has heuristic information and ability to considerably decrease the reduction time, the problem of finding potential candidates within such a difficult search space makes it increasingly hard to uncover more axioms. As a result, BeamSearch and BeamSearch$^{\text{FastMinSet}}$ gives us a better understanding of the solution space as well as the KeY's proof procedure. This is of great help, as our tested system is a black box and we need such information to improve our framework.

In addition, although the BeamSearch$^{\text{FastMinSet}}$ explored a vast number of nodes in the search space, the actual total coverage is just half of the axiomatisation base. In other words, the approach reaches a local optima. This is because the actual programs in the test suite do not use the logic encoded in the rest of the axioms. Thus, it is obvious that the current test suite needs some updates to include new

| | total successful replacements | total unsuccessful replacements | total | % of successful replacements |
|---|---|---|---|---|
| Depth-First search | 7,973 | 135,655 | 143,628 | 6% |
| Random Depth-First search | 4,034 | 39,015 | 43,049 | 9% |
| Greedy | 2,544 | 81,171 | 83,715 | 3% |
| Breadth-First search | 9,995 | 31,299 | 41,294 | 24% |
| Random Breadth-First search | 11,008 | 30,598 | 41,606 | 26% |
| BeamSearch | 36,084 | 60,646 | 96,730 | 37% |
| BeamSearch$^{\text{FastMinSet}}$ | 58,982 | 244,506 | 303,488 | 19% |

Table 5.7: Successful vs. unsuccessful replacements: non-unique sets of axioms

test cases. In other words, it is essential to manually create test cases that carry different logical properties to the existing ones.

#### 5.3.3.4 BeamSearch$^{\text{FastMinSet}}$ Analysis

Now let us look to how the BeamSearch$^{\text{FastMinSet}}$ approach explores the search space. Figure 5.4 illustrates a small part of a medium size test case's search space. The runtime for executing this test case is approximately 13.5 hours.

At the beginning the KeY system starts the proof procedure using the 1,520 axioms. It successfully finds a proof for the given test case using 38 axioms. However, only 31 axioms of them are mandatory axioms. Therefore, the first minimal set $M_1$ contains only these axioms.

Initially, the approach identifies four axioms to be dropped *a2, a4, a5* and *a10*. After excluding each axiom, the verification tool successfully manages to prove the same test case. As a result, the tool finds four more *Ms*. Three *Ms* of them result in an increase in the axiomatisation coverage, whereas the fourth one contain

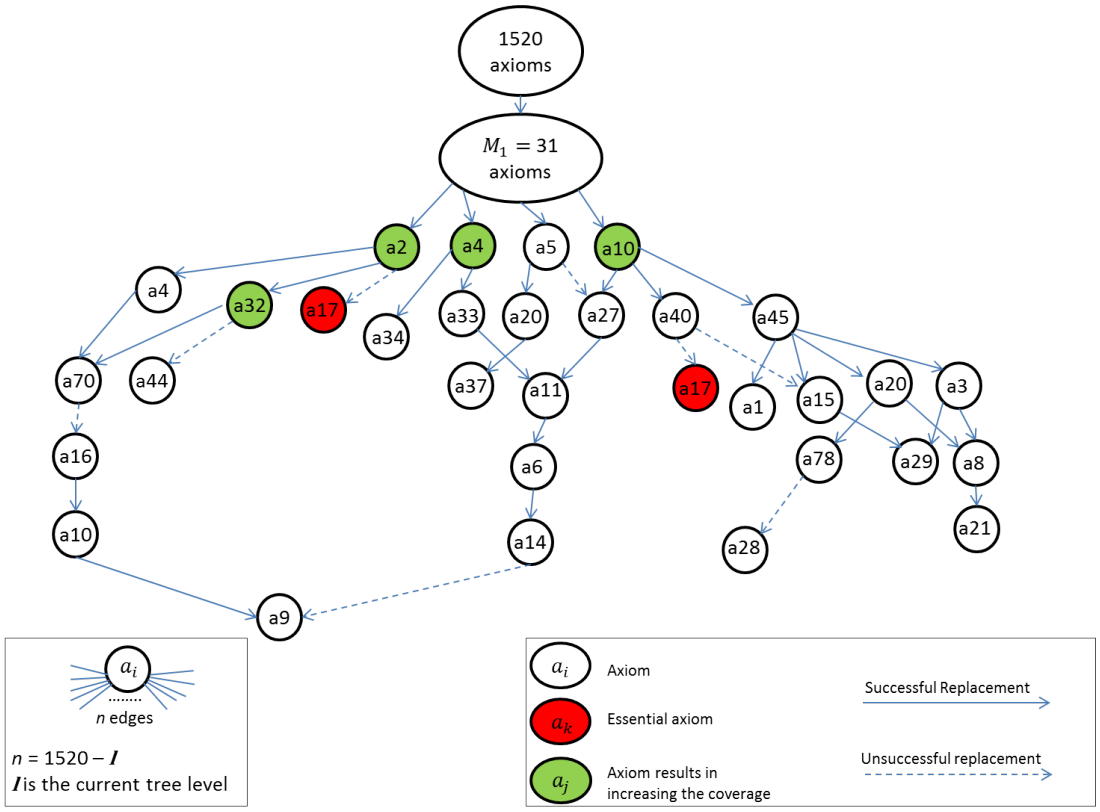



Figure 5.4: Search Space Example: exploring the search space by the beam search approach. Each node represents an axiom. Solid arrows indicate successful replacements while dashed arrows are used for unsuccessful replacements.

previously covered axioms. In Figure 5.4, the axioms in green shows the instances where replacing these axioms increases the coverage. For example, after replacing *a2*, a new $M$ is found. It contains 32 axioms. Amongst these axioms, there is only one new axiom *a32*. It is worth mentioning that in this example, one may consider $(a2 = a32)$, this is only true with the current available axiomatisation base that KeY uses in its proving procedure, and it is capability to prove the given test case in different ways. In addition, replacing *a4* and *a10* increases the coverage by three more axioms. As a result, of replacing the three axioms, the coverage increases from 31 to 35 axioms.

In the subsequent stage, the approach tries to identify new promising nodes to drop together with the previous ones, i.e. it drops sets of axioms. It chooses seven axioms in total from the second level; as a consequence, it creates eight different sets. Only two sets are unsuccessfully replaced {*a2, a17*} and {*a5, a27*}. The red node in Figure 5.4 represents an essential node that has not been replaced in all test cases, i.e. its total successful replacement in the $GT$ equals to zero.

On the other hand, two sets of the six successful replaced sets increase the coverage. For instance, excluding both axioms {*a2, a32*} from the axiomatisation base, forces the tool to use a different set of axioms; therefore another new $M_5$ is found. In addition, forbidding the tool from using the set of axioms {*a2, a4*} results in a new $M_6$, however, this time all its axioms are already covered in the previous $Ms$.

It is worth mentioning that our approach identifies the first four nodes without trying to drop the 1520 axioms, which significantly reduces the time complexity. Furthermore, in total, it tries only 35 replacements where 77% of them are successful.

Summarising the results of this section, we make the following conclusions:

1. Through the use of the guidance table, BeamSearch$^{\text{FastMinSet}}$ and BeamSearch operates more effectively. This allows us to identify more relationships among the axioms to improve our framework for future runs.

2. Moreover, our results clearly show that even though BeamSearch$^{\text{FastMinSet}}$ uses heuristic information and has the ability to decrease the reduction time, the problem of finding potential candidates within such a difficult search space makes it increasingly difficult to cover further axioms. Therefore, we conjecture that we are getting increasingly close to the local optimum that

we can achieve with our current approach.

## 5.4 Conclusions and Future Work

In this research, we presented BeamSearch and BeamSearch$^{\text{FastMinSet}}$ for increasing the axiomatization coverage in deductive verification systems, where a set of axioms—logical rules that capture the semantics of a programming language—is used to find a proof that a program satisfies its formal specifications. Our approaches automatically creates test cases by preventing the verification tool from using previously covered axioms. Therefore, the system tries to find alternative axioms to prove the program. In our situation, a test case consists of the verifiable program, its requirements, and the allowed set of axioms. Our heuristic approach involves a learning process where the beam search method uses a guidance table that contains special historical data from previous runs. As a result, it explores the search space more effectively than previous approaches that use uninformed breadth-first and depth-first variants. Whilst successful in increasing the coverage of our tested verification system, these uninformed techniques often generated infeasible solutions during their search, and they are not much directed towards an actual increase of the coverage.

The experiments reveal several interesting insights. First, our approach achieves a coverage comparable to that of the union of five previous approaches, when given the same computation budget. Furthermore, the overall coverage has been improved over the starting point by 24%. Second, the high number of unsuccessful replacement attempts by our fast approach strongly indicates that we are getting increasingly close to the local optimum of "maximum coverage" that we can reach with our test case reuse.

Finally, we found there is no correlation between code and axiomatization coverage. Although the code coverage reaches more than 34%, the axiomatisation coverage is nearly 10%. Thus, it is essential to focus on maximizing the axiom coverage to uncover hidden defects in the axiomatisation coverage.

We will continue our research in the following areas:

1. We plan to investigate the reasons why some axioms are not covered, amongst others, using the help of developers of the verification systems. We will systematically write specific test cases aimed to increase the axiomatization coverage for specific axioms.

2. Once we reach a satisfactory axiomatization coverage, we will need to focus on combinations of axioms. Failures in a variety of domains are often caused by combinations of several conditions (see studies like [17]). We plan to combine combinatorial testing with combinatorial search techniques. Then, combinations of language features and axioms will be used to form complex test cases. The knowledge gained from the work presented here will help us to focus our efforts in comprehensive testing.

# Appendices

# Appendix A

# Log File Example

In the following, an example of test log is presented. The used approach is the breadth-first search. When operating the BeamSearch$^{\text{FastMinSet}}$ on the same test case, the steps for the reduction is eliminated. Descriptive comments are added to explain parts of the test case, these comments are in italics font style.

Axiomatisation Coverage
parameters: [standard_key/java_dl/arrayUpdateSimp.key, 1] **case name, approach number**
reduceWithBlackList (0):[]
**no excluded axioms (0)**
<result>timeReportedFromKeY=188
**time for finding the initial set of axioms in ms**
new minimalSet can be found, UsedInProof.size=3 **size of the initial set of axioms. the following lines contain the output of the reduction phase. y for needed, n for unneeded, t is the time needed for checking the axiom .**

<result>Last checked: [closeTrue](needed=y, t=176); Needed: [closeTrue];
Remaining: [...]
<result>Last checked: [eq_imp](needed=y, t=92); Needed: [closeTrue, eq_imp]; Remaining: [...]
<result>Last checked: [simplifyUpdate2](needed=y, t=44); Needed: [closeTrue, eq_imp, simplifyUpdate2]; Remaining: [...]
<result>Last checked: [closeTrue](needed=y, t=176); Needed: [closeTrue]; Remaining: [...]
<result>Last checked: [eq_imp](needed=y, t=92); Needed: [closeTrue, eq_imp]; Remaining: [...]
<result>Last checked: [simplifyUpdate2](needed=y, t=44); Needed: [closeTrue, eq_imp, simplifyUpdate2]; Remaining: [...]

**here the reduction phase finishes, the output (M) is:**
minimalSet number 1 found, total covered taclets = 3, Remaining (3):[closeTrue, eq_imp, simplifyUpdate2] reduceWithBlackList (0):[] time needed t=188
reduceWithBlackList (1):[closeTrue] **one axiom is dropped from the whitelist**
<result>timeReportedFromKeY=191
reduceWithBlackList (1):[eq_imp] **another axiom is dropped**
<result>timeReportedFromKeY=209
**dropping eq_imp forced KeY to find another proof.**
new minimalSet can be found, UsedInProof.size=5
... some skipped line for reduction ...
minimalSet number 2 found, total covered taclets = 5, Remaining (3):[close, impRight, simplifyUpdate2] reduceWithBlackList (1):[eq_imp] time needed t=98
reduceWithBlackList (2):[close, eq_imp] **Now dropping set of axioms**
<result>timeReportedFromKeY=213
new minimalSet can be found,
UsedInProof.size=6

... skipped lines ...

Minimal sets found:4

[close, impRight, simplifyUpdate2]

[closeFalse, impRight, replace_known_right, simplifyUpdate2]

[closeTrue, eq_imp, simplifyUpdate2]

[closeTrue, impRight, replace_known_left, simplifyUpdate2]

... end of the file ...

# Bibliography

[1] Wolfgang Ahrendt, Bernhard Beckert, Daniel Bruns, Richard Bubel, Christoph Gladisch, Sarah Grebing, Reiner Hähnle, Martin Hentschel, Mihai Herda, Vladimir Klebanov, Wojciech Mostowski, Christoph Scheben, Peter H. Schmitt, and Mattias Ulbrich. The key platform for verification and analysis of java programs. In Dimitra Giannakopoulou and Daniel Kroening, editors, *Verified Software: Theories, Tools, and Experiments (VSTTE 2014)*, number 8471 in Lecture Notes in Computer Science, pages 1–17. Springer-Verlag, 2014.

[2] Thomas Back, David B Fogel, and Zbigniew Michalewicz. *Handbook of evolutionary computation*. IOP Publishing Ltd., 1997.

[3] Bernhard Beckert, Thorsten Bormer, and Markus Wagner. Heuristically creating test cases for program verification systems. In *Metaheuristics International Conference (MIC)*, 2013.

[4] Bernhard Beckert, Reiner Hähnle, and Peter H. Schmitt, editors. *Verification of Object-Oriented Software: The KeY Approach*, volume 4334 of *LNCS*. Springer, 2007.

[5] Bernhard Beckert, Markus Wagner, and Thorsten Bormer. A metric for test-

ing program verification systems. In *Tests and Proofs (TAP)*, volume 7942 of *LNCS*, pages 56–75, 2013.

[6] Mahmoud Bokhari. Software testing a verification system, mini thesis. Master's thesis, Computer Science, University of Adelaide, 2014.

[7] Mahmoud Bokhari, Thomson Bomer, and Markus Wagner. An improved beam-search for testing formal verification systems. In *Proceedings of the 2015 Symposium on Search-Based Software Engineering*, SSBSE '15. Springer, 2015. (to be published).

[8] Mahmoud Bokhari and Markus Wagner. Improving test coverage of formal verification systems via beam search. In *Companion of the 2015 Conference on Genetic and Evolutionary Computation*, GECCO '15. ACM, 2015. (to be published).

[9] Thorsten Bormer and Markus Wagner. Towards testing a verifying compiler. In *Int. Conference on Formal Verification of Object-Oriented Software (FoVeOOS). Pre-Proceedings*, pages 98–112. Karlsruhe Institute of Technology, 2010.

[10] Daniel Bruns, Wojciech Mostowski, and Mattias Ulbrich. Implementation-level verification of algorithms with key. *International Journal on Software Tools for Technology Transfer*, pages 1–16, 2013.

[11] P.J. Deitel and H.M. Deitel. *Java: How to Program*. How to program series. Pearson Prentice Hall, 2010.

[12] Marco Dorigo, Mauro Birattari, and Thomas Stutzle. Ant colony optimization. *Computational Intelligence Magazine, IEEE*, 1(4):28–39, 2006.

[13] Vijay D'Silva, Daniel Kroening, and Georg Weissenbacher. A survey of automated techniques for formal software verification, 2008.

[14] Stefan Edelkamp and Stefan Schroedl. *Heuristic search: theory and applications*. Elsevier, 2011.

[15] Jean-Christophe Fillitre. Deductive software verification. *International Journal on Software Tools for Technology Transfer*, 13(5):397–403, 2011.

[16] Marc R. Hoffmann. *Jacoco*, 2014. accessed September 2014.

[17] D. Richard Kuhn, Dolores R. Wallace, and Albert M. Gallo. Software fault interactions and implications for software testing. *IEEE Transactions on Software Engineering*, 30(6):418–421, 2004.

[18] R. Lingampally, A. Gupta, and P. Jalote. A multipurpose code coverage tool for java. In *System Sciences, 2007. HICSS 2007. 40th Annual Hawaii International Conference on*, pages 261b–261b, Jan 2007.

[19] Aditya P. Mathur. *Foundations of Software Testing*. Addison-Wesley Professional, 1st edition, 2008.

[20] Glenford J. Myers, Corey Sandler, and Tom Badgett. *The Art of Software Testing*. Wiley Publishing, 3rd edition, 2011.

[21] Salvatore Ruggieri. On computing the semi-sum of two integers. *Inf. Process. Lett.*, 87(2):67–71, July 2003.

[22] Agitar Development Team. *Agitar*. Agitar Technologies, Cranston, USA, 2014. accessed September 2014.

[23] Cobertura Development Team. *Cobertura*, 2014. accessed September 2014.

[24] EMMA Development team. *EMMA*, 2006. accessed September 2014.

[25] Jcover Development team. *Jcover*. Man Machine System, 2009. accessed September 2014.

[26] Jtest Development Team. *Jtest Software*. Parasoft, california, USA, 2012. accessed September 2014.

[27] Purify Plus Development Team. *Purify Plus*. IBM, New York, USA, 2012. accessed September 2014.

[28] Mustafa M. Tikir and Jeffrey K. Hollingsworth. Efficient instrumentation for code coverage testing. *SIGSOFT Softw. Eng. Notes*, 27(4):86–96, July 2002.

[29] M. Wagner. Maximising axiomatization coverage and minimizing regression testing time. In *IEEE Congress on Evolutionary Computation (CEC)*, pages 2885–2892, July 2014.

[30] Qian Yang, J. Jenny Li, and David Weiss. A survey of coverage based testing tools. In *Proceedings of the 2006 International Workshop on Automation of Software Test*, AST '06, pages 99–103, New York, NY, USA, 2006. ACM.

[31] Hong Zhu, Patrick A. V. Hall, and John H. R. May. Software unit test coverage and adequacy. *ACM Computing Surveys*, 29(4):366–427, December 1997.