An Improved Beam-Search for the Test Case Generation for Formal Verification Systems

Mahmoud A. Bokhari[1,2], Thorsten Bormer[3], and Markus Wagner[4(✉)]

[1] Computer Science School, University of Adelaide, Adelaide, Australia
[2] Computer Science Department, Taibah University, Medina, Kingdom of Saudi Arabia
[3] Institute for Theoretical Informatics, Karlsruhe Institute of Technology, Karlsruhe, Germany
[4] Optimisation and Logistics, University of Adelaide, Adelaide, Australia
markus.wagner@adelaide.edu.au

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 verification systems need to understand their target software, which can be done by using an axiomatization base. It captures the semantics of the programming language used for writing the target software. To ensure their correctness, it is necessary to validate both parts: the implementation and the axiomatization base. As a result, it is essential to increase the axiom coverage in order to verify its correctness. However, creating test cases manually is a time consuming and difficult task even for verification engineers. We present a beam search approach to automatically generate test cases by modifying existing test cases as well as a comparison between axiomatization and code coverage. Our results show that the overall coverage of the existing test suite can be improved by more than 20 %. In addition, our approach explores the search space more efficiently than existing ones.

Keywords: Beam search · Formal verification · System testing

1 Introduction

Formal verification is the act of proving or disproving that an algorithm or its implementation is correct with respect to its formal specification. The formal mathematical approaches include, amongst others, model checking, deductive verification, and program derivation [4,7,12].

The correctness of the program verification systems themselves is imperative if they are to be used in practice. In principle, instead of or in addition to testing, parts of verification tools (in particular the axiomatization and the calculus) can be formally verified. For example, the Mobius project [2], the LOOP project [15], and the Bali project [18], all aimed at the development of fully verified verification systems. One may employ formal methods to prove a system or

M. Barros and Y. Labiche (Eds.): SSBSE 2015, LNCS 9275, pp. 77–92, 2015.
DOI: 10.1007/978-3-319-22183-0_6

its calculus to be correct. But—as for any other type of software system—testing and cross-validation are of great importance [3,10].

In our situation of testing formal verification systems, all tests have to be programs (along with their formal specifications) that can be verified successfully, whether it is with or without human interaction. Due to their inherent complexity, creating such test cases by hand is already a challenging problem for experienced verification engineers. Currently, it is unknown how tests can be generated automatically from scratch using existing methods.

A verification system consists of many testable components (e.g., parsers, the user interface, proof procedures), and the so-called *axiomatization* is one of them. It carries the formal definitions of the target program language, which makes it a core component of the systems. The correctness of this component is of outmost importance, *especially* when safety- and security-critical programs are to be verified.

To ensure the correctness of program verification tools, it is necessary to validate both parts: the implementation, as well as the axiomatization. Only testing the implementation is not sufficient, even if a high code coverage is achieved. For example, it was noted in [6] that there is a certain amount of "core code" exercised by all tests, while there is only a small number of "core axioms" used by many tests. Some logical defects stay hidden within the axiomatization unless it is fully exercised. The work in [6] discovered two bugs in the axiomatization as a result of the coverage maximization research.

Our goal is to increase the proportion of the axiomatization that is actively used in successful verification attempts [6]. As a consequence, new bugs ("regressions") are more likely to be found in regression testing, when the implementation of the verification system (and its axiomatization) is changed. The problem is challenging for iterative search approaches due to the large number of axioms (typically 100's) and due to the time consuming verification process (sometimes minutes), which also makes it unsuitable for population-based evolutionary algorithms or ant-colony optimization as they require many evaluations [1,13]. 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 [19] a collection of various breadth-first and depth-first approaches with randomized components to this problem was investigated. These approaches were not problem-specific in the sense that the search for the next test case was completely uninformed. In contrast to this, we are using in this article a beam search approach [8,9] that is informed by previous runs. We do this in order to achieve two goals: (1) reduction of the likelihood to generate infeasible solutions, and (2) increase of the likelihood to cover previously uncovered axioms.

First, we outline the specific problem in Sect. 2, and in Sect. 3 we formulate it as an optimization problem. In the subsequent Sect. 4, we describe our informed approach. We analyze our results and compare them with existing approaches in Sect. 5. The paper concludes in Sect. 6 with a summary of key findings and a description of potential future research.

2 Target of Optimization: Program Verification Systems

In this article, we concentrate on modern verification systems that allow for *auto-active verification*. In auto-active verification, the requirement specification together with all relevant information to find a proof (e.g., loop invariants) is given to the verification tool right from the start of the verification process—interaction hereafter is not possible. While some tools such as VCC [11] and Caduceus [14] allow only this type of interaction, other such as the KeY tool [4], offer in addition an interactive mode for the proof construction.

Program verification tools have to capture the program language semantics of the programs to be verified. In some tools (e.g., as with logical frameworks like Isabelle/HOL [17]) these semantics are mostly stored as one huge axiomatization or a set of calculus rules and separate from the actual proof system. At this end of the spectrum of program verification systems, (at least) one rule is defined per program language construct (e.g., control flow statements or evaluation of arithmetic expressions) in order to conduct proofs about program correctness. The task of the actual implementation part of the verification tool is then mostly to apply these rules, respectively axioms.

We consider in this article system tests, i.e., the verification tool is tested as a whole. Though the correctness of a tool, of course, 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 possible (and useful) to unit-test an SMT solver that is used by a tool. But the verification condition generator is hard to test separately as it is very difficult to specify its correct behaviour. In the following, we concentrate on functional tests that can be executed automatically, i.e., user-interface properties are not considered.

As is typical for verification tools following the auto-active verification paradigm, we assume that a verification problem consists of a program to be verified and a requirement specification that is added in form of annotations to the program. Typical annotations are, e.g., invariants, pre-/postcondition pairs, and assertions of various kinds. If P is a program and A is a set of annotations, then we call the pair $P{+}A$. Besides the requirement specification, a verification problem usually contains additional auxiliary annotations that help the system in finding a proof. We assume that all auxiliary input (e.g., 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).

In the following, we are only considering test cases for which the intended outcome is that KeY finds a proof given the allocated computational resources.

3 Problem Formulation

In this section, we present how we determine the amount of testing done, and how we intend to improve it.

3.1 Axiomatization Coverage

Measuring code coverage is an important method in software testing to judge the quality of a test suite. This is also true for testing verification tools. However, code coverage is not an indicator for how well the declarative logical axioms and definitions—that define the semantics of programs and specifications and that make up an important part of the system—are tested.

To solve this problem, we use the notion of axiomatization coverage [6]. It measures the extent to which a test suite exercises the axioms (that capture the program language semantics) used in a verification system. The idea is to compute the percentage of axioms that are actually used in the proofs for the verification problems that make up a test suite. The higher the coverage of a test suite is, the more likely it is that a bug that is introduced in a new version of the verification system is discovered.

We use the following version of axiomatization coverage: the percentage of axioms needed to successfully verify correct programs. An axiom is defined to be *needed* to verify a program, if it is an element of a *minimal axiom subset*, using which the verification system is able to find a proof. That is, if the axiom is removed from this subset, the tool is not able anymore to prove the correctness of the program.

Definition 1 ([6]). *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* $Th' \subsetneq Th$.

Note that, in general, the minimal set of axioms covered by a given verification problem is not unique.

To compute an approximation of the axiom coverage for a test case $P{+}(REQ \cup AUX)$, the procedure is as follows. In a first step, we verify the test case with the verification tool using the complete axiom base available. Besides gathering information on resource consumption of this proof attempt, information on which axioms are actually used in the proof are recorded as set T.[1] Then, the iterative reduction phase starts. In a reduction step, we start from the empty set C of *covered* axioms. For each axiom t in the set of initially used axioms T, an attempt to prove the test case using axioms $C \cup (T \setminus \{t\})$ is made. If the proof does not succeed, we consider t to be necessary and we add t to the set C. Then, we remove axiom t from T and start the next proof iteration until

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

$T = \varnothing$. After a single iteration of this computation, we repeat these operations but this time on C, until no more axiom removal is possible without affecting the ability to find a proof. As a result, this fixed-point algorithm finds a true minimal set of axioms to construct the proof.

3.2 Maximizing Axiomatization Coverage

We increase the amount of testing done by generating additional tests from existing ones. We achieve this by preventing the verification system to use certain parts of the axiomatization. Thus, we force the system to find alternative ways of constructing a correctness proof for a given test case $P+(REQ \cup AUX)$, while using only a subset of the total set of axioms. We will refer to this subset of allowed axioms as the whitelist WL. Now, the notion of what a test case constitutes actually changes: it becomes 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 the whitelists allows us to reuse existing test cases, 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. On the other hand, our approach cannot fully replace the need to extend test suites through additional test cases. For example, take axioms for bitwise XOR-operations or for certain simplifications of inequalities. Even though many parts of the axiomatization will be reused over and over, it may not be possible to cover these, if the corresponding characteristics are never found in any of the existing test cases.

4 Metaheuristic Approach

In the following, we describe the verification system that is the subject of our study. Subsequently, we present our heuristic approaches to the problem. Note that our approaches can be applied to the testing of further verification systems, if these can provide information on which axioms were used during the construction of the proof; for example, this is the case for Microsoft's VCC [11].

4.1 The KeY System

As the target for our case study we have chosen the KeY tool [4], a verification system for sequential Java Card programs. In KeY, the Java Modeling Language (JML) is used to specify properties about Java programs with the common specification constructs like pre- and postconditions for methods and object invariants. Like in other deductive verification tools, the verification task is modularized by proving one Java method at a time.

In the following, we will briefly describe the workflow of the KeY system. Let us assume the user has chosen one method to be verified against a single pre-/postcondition pair. First, the relevant parts of the Java program, together

with its JML annotations are translated to a sequent in Java Dynamic Logic, a multimodal predicate logic. Validity of this sequent implies that the program is correct with respect to its specification. Proving the validity is done using automatic proof strategies within KeY, which apply sequent calculus rules implemented as so-called *taclets*. For an in-depth introduction, we refer the interested reader to [4]. The set of taclets provided with KeY captures the semantics of Java. Additionally, it contains taclets that deal with first order logic formulas. The development version of KeY as of 16 August 2012, contains 1520 taclets and rules; we will call these *axioms* in the remainder of this article to facilitate reading. Note that not all axioms are always available when performing a proof, as some exist in several versions, depending on proof options chosen.

4.2 Algorithms

As stated above, we are aiming at maximizing the axiomatization coverage through the creation of test cases $\langle P{+}(REQ \cup AUX), WL\rangle$. The test suite that we will consider already contains pairs $P{+}(REQ{\cup}AUX)$, such that we can focus on the search for whitelists. This process can be very time consuming (several hours) due to the reduction phases. Furthermore, it is very often the case that infeasible whitelists are created, as they miss elements that are crucial for the construction of the eventual proof. Even a very "careful" random generation of whitelists is rarely successful. Therefore, we use an informed conservative approach in which we attempt to use the knowledge gained so far.

Guidance Table. To efficiently navigate the search space, we propose a new approach that uses a guidance table GT to guide and inform the search. This GT consists of the following: each found axiom encountered in any 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 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 it was not possible to replace {*ax3*}.

Table 1. Guidance Table example

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

The major purpose of using the guidance table is to find equivalences between axioms.[2] In other words, it lists equivalent sets of axioms for each axiom or axiom sets. For example, Table 1 depicts that *ax3* has two equivalent sets of axioms which are $\{ax5, ax6\}$ and $\{ax7\}$. It is worth mentioning that these two sets are only equivalent to that axiom in four cases in total. However, as they are not completely equal to *ax3*, the first set could not replace *ax3* in one instance.

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 such a table that can be used to inform and guide it towards promising nodes. Moreover, the guidance table can identify irreplaceable axioms that have not been replaced successfully. Avoiding these axioms improves the performance of the search process and the framework.

BeamSearch 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. If a proof can be constructed, then 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, however, in some cases an axiom may not found in the GT which means it is a newly covered axiom. Furthermore, these axioms must have relatively high successfully replacement rates. Lastly, axioms that have not been 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 overall axiomatization coverage.

In the final stage, the promising list is sorted by each axiom's successful replacements in a descending order, then the method starts exploring the promising axioms. We do this by dropping one at a time from the axiomatization base WL and then re-running the proving procedure using the shorter WL (Step 1). As a consequence a new test case might be generated, but this time an axiom which has several logically equivalent sets of axioms has been removed from it, which increases the chances of forcing the verification tool to use different axioms.

BeamSerch$^{\text{FastMinSet}}$. In order to reduce the computation time, we use some additional information obtained by the GT tool from the previous test runs.

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

Algorithm 1. BeamSearch

```
Data: GT: guidance table (sorted by successful replacements)
Data: TC: test case ⟨P+(REQ ∪ AUX), WL⟩
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 discarded nodes
Result: union of all minimal lists
T = Run(TC) ;                                          /* run the verification tool */
if TC is not Proved OR Timeout then
    Stop;
else
    M = Reduce(T);
    Add M to result;                                   /* add the newest minimal list */
    foreach axiom in M do
        if GT contains axiom then
            if axiom in GT has total successful time greater than 0 then
                Add axiom to PromisingNodeList;
            else
                Add axiom to DiscardedList;
            end
        else
            Add axiom to PromisingNodeList;      /* adding new axiom, since it has not
                                                    been in the GT so far */
        end
    end
    sort(PromisingNodeList);                       /* by total replacements (descending) */
    foreach axiom in PromisingNodeList do
        Drop axiom from current WL;
        Repeat from Step 1;
    end
end
return result;
```

For each test case TC, the GT tool collects and stores all of the initially used axiom sets T and their reduced minimal 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 a set of M. As a result, the tool generates a hash table where the keys are pairs of $\langle TC, T\rangle$ and the values are sets of M.

The approach BeamSearch$^{\text{FastMinSet}}$ has two main parts: (1) it effectively tries to quickly re-discover the previously found minimal sets M, and (2) 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) is already discussed in BeamSearch Approach in Sect. 4.2. As can be seen, the set T of axioms used in proofs is obtained by running the verification tool on the test case TC using the whole white list WL. Additionally, the approach looks for the corresponding minimal set M from the hash table HT.

In the next stage, when M is found, the BeamSearch$^{\text{FastMinSet}}$ reruns the tool again, but this time the WL is replaced by the corresponding M (which was a successful reduction at least once before), to ensure the validity of M. It is worth mentioning that we add this step, since the verification tool may undergo some modifications that affect the proof procedure. Then the BeamSearch$^{\text{FastMinSet}}$ uses the valid M for building the promising list. However, in case the HT does

Algorithm 2. BeamSearch$^{\text{FastMinSet}}$

```
   Data: HT: hash table ((TC, T), M)
   Data: TC: test case ⟨P+(REQ ∪ AUX), WL⟩
   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
   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;
```

not contain such T (i.e., it is new, or M is not valid), the BeamSearch$^{\text{FastMinSet}}$ continues its job as BeamSearch, by reducing T to a new minimal set M and then constructs the promising list.

As can be noted, 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, still it is 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.

5 Experimental Investigations

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.

5.1 Experimental Setup

Our testing framework automatically executes the 319 test cases mentioned above and measures the axiomatization coverage[3]. We also use Emma tool version 2.0 to measure the code coverage[4]. It worth mentioning that we run 2 separated experiments, one for each coverage criterion. For code coverage the

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

[4] www.emma.sourceforge.net (last accessed: 5 April 2015).

reduction phase is disabled during the test, and therefore the number of covered axioms is slightly different to those in the axiomatization experiment.

This test and all subsequent runs are performed on Intel Xeon E5430 CPUs (2.66 GHz), 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 takes typically minutes, and in a few cases even hours. Therefore, we limit the computation time for each of the 319 test cases to 24 h for each approach, which means an investment of 0.87 CPU years per approach. We compare our informed beam-search approach to the different variant of uninformed breadth-first and depth-first approaches reported in [19]. We observed in preliminary testing that even the approaches with random selectivity produced the same results in independent runs with negligible deviations (± 1 covered axiom), which is why (in addition to the computational cost) we limit our investigations to only one run per approach.

Before our experiments, we build the guidance table for our beam search based on re-runs of the Approaches 1–5 in [19]. This guidance table contains, amongst others, the following interesting information. This step is mandatory, as otherwise our approach default to a simple breadth-first search.

First, let us look at individual axioms. In Fig. 1, 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 straight-forward, 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 the promising list, as they appear to be dead ends more often than not. On the end of the spectrum, 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 2 shows how many different single axioms

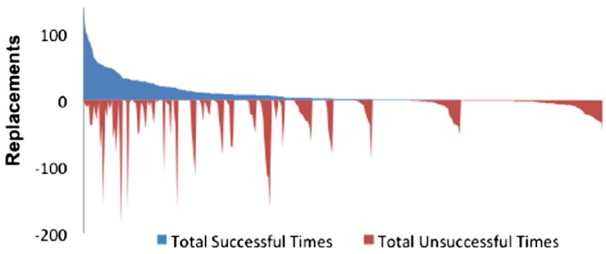

Fig. 1. Number of successful vs. unsuccessful replacements for each single axiom, shown as positive and negative values. The axioms (along the x-axis) are sorted in a decreasing order according to the number of successful replacements.

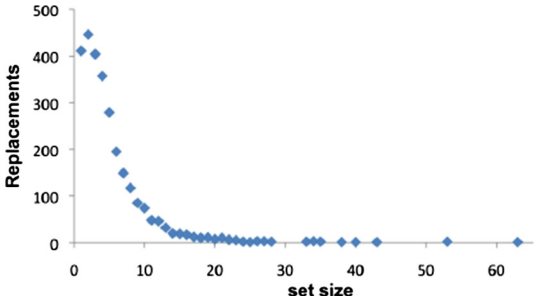

Fig. 2. Replacement set's sizes and number of replacements for single axioms BEAM-SEARCH$^{\text{FastMinSet}}$

are replaced by a set of axioms. For example, 446 pairs of axioms successfully replace a single axiom, and there is one case where one axiom is replaced by an enormous set of 63 axioms. In the future, 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, since it may increase the execution time for the proof procedure.

5.2 Code Coverage Results

Achieving high code coverage in software testing is of great importance to judge the test suite. Nevertheless, in verifying deductive verification systems, our results show axiomatization coverage is essential. This is because although in 295 test cases the lines of code (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 2 shows a summary of the coverage details for the test cases.

As can be seen, the average LOC coverage is 37 %, in contrast, it is only 4.43 % for the axiomatization coverage. Most of test cases exercised nearly the same proportion of the code as the standard deviation for LOC is 2.5 %. On the other hand, there are fluctuations in the amount of covered axioms 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 used axioms.

The overall axiomatization coverage—as it is expected—is low with only 45 %. Additionally, the LOC coverage is only 51 %, which is significantly less than the 85 % recommended by the software testing (see e.g. [20]). Though the class coverage reached 90 %, after we analyzed the EMMA outputs, we find 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 3 for some examples).

Table 2. Code Coverage vs. Axiomatization Coverage (excerpt). Sorted by axiom percentage in descending fashion.

Test Cases	Code Coverage		Axiomatization Coverage	
	Class Coverage	Line Coverage	Number of Axioms	Axiom Percent
standard_key-java_dl-arrayUpdateSimp .... 182 test cases	87 %	35 %	3	0.20 %
heap-SmansEtAl-Iterator_list ... 134 test cases	83 %	40 %	62	4.08 %
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 3. Some classes within KeY tool.

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

5.3 Axiomatization Coverage Results

The coverage statistics of the different approaches are listed in Table 4. The number 611 represents the result of the naive approach, where the full set of 1520 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 12–15 % each. The highest individual improvements are made by our BeamSearch$^{\text{FastMinSet}}$.

When considering all approaches together, then the initial coverage of about 611 axioms increases to a total of 755 axioms through the use of whitelists. This means that all approaches together improve 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 Approaches 1–5 together. As we can see via the 755 axioms that are covered by the union of all seven approaches, our beam search does not only cover most of what Approaches 1–5 do, but it also covers additional 33 axioms. It appears that the combination of guidance table and the fast reduction (when available) allows it to search more effectively than the previous approaches.

Table 4. Coverage statistics. The *first minimal sets* refer to those found first by the approaches, which initially use all 1520 axioms. The results for Approaches 1–5 are based on reruns from [5].

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. Successful vs. unsuccessful replacements: unique single axioms.

	Successfully replaced single axioms	Unsuccessfully replaced single axioms	Total	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 6. Analysis: equivalent sets found by each approach.

	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 %

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 Table 5, and they clearly show the structural differences between the approaches. For example, the depth-first Approaches 1–3 have the smallest number of replacements of single axioms, which is expected given their nature: they will explore shorter and shorter white lists first. The breadth-first Approaches 4/5 on the other hand achieve significantly higher single replacements, since they explore the replacement of single axioms first. Our beam search achieves the highest number of replacements here, since it prefers the replacements of single axioms, and it also considers single axioms when it comes across new ones in the search. This has the big advantage for the use of the guidance table that 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 6 presents the total *attempts*, as well as the amount of equivalent sets and their percentages. As we can see, the success rate is in favor of BeamSearch with 44 %. On the other hand, among all BeamSearch$^{\text{FastMinSet}}$ is the fourth with 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 more time exploring the search space. In addition, it is worth mentioning that the numbers of the total attempts represent the sizes of each approach's generated guidance table, which shows that a large amount of information for the beam search is extracted.

In contrast to this, 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, that achieve the fourth and fifth highest number 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.

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

1. Through the use of the guidance table, BeamSearch$^{\text{FastMinSet}}$ and BeamSearch search more efficiently. This also allows us to identify more logical relationships among the axioms to improve our framework for future runs.
2. Moreover, our results clearly show that even though BeamSearch$^{\text{FastMinSet}}$ is using heuristic information and it has the ability to decrease the reduction time, the problem of finding potential candidates within such a difficult search space makes it increasingly hard 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.

6 Conclusions and Future Work

In this article, we present a beam search approach 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 approach automatically creates test cases by preventing the verification tool from using previously covered axiom. Therefore, the system tries to find alternative axioms to prove the program. 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 and therefore it is essential to focus on maximizing the axiom coverage to uncover hidden defects.

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 will have reached 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 [16]). 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.

References

1. Back, T., Fogel, D.B., Michalewicz, Z.: Handbook of evolutionary computation. IOP Publishing Ltd., (1997)
2. Barthe, G., et al.: MOBIUS: mobility, ubiquity, security. In: Montanari, U., Sannella, D., Bruni, R. (eds.) TGC 2006. LNCS, vol. 4661, pp. 10–29. Springer, Heidelberg (2007)

3. Beckert, B., Klebanov, V.: Must program verification systems and calculi be verified? In: Verification Workshop (VERIFY), Workshop at Federated Logic Conferences (FLoC), pp. 34–41 (2006)
4. Beckert, B., Hähnle, R., Schmitt, P.H. (eds.): Verification of Object-Oriented Software. The KeY Approach. LNCS, vol. 4334. Springer, Heidelberg (2007)
5. Beckert, B., Bormer, T., Wagner, M.: Heuristically creating test cases for program verification systems. In: Metaheuristics International Conference (MIC) (2013)
6. Beckert, B., Bormer, T., Wagner, M.: A metric for testing program verification systems. In: Veanes, M., Viganò, L. (eds.) TAP 2013. LNCS, vol. 7942, pp. 56–75. Springer, Heidelberg (2013)
7. Bérard, B., Bidoit, B., Finkel, M., Laroussinie, F., Petit, A., Petrucci, L., Schnoebelen, P.: Systems and Software Verification: Model-checking Techniques and Tools. Springer, Heidelberg (2010)
8. Blum, C., Blesa, M.J.: Probabilistic beam search for the longest common subsequence problem. In: Stützle, T., Birattari, M., H. Hoos, H. (eds.) SLS 2007. LNCS, vol. 4638, pp. 150–161. Springer, Heidelberg (2007)
9. Bokhari, M., Wagner, M.: Improving test coverage of formal verification systems via beam search. In: Companion of the 2015 Conference on Genetic and Evolutionary Computation, GECCO 2015. ACM (2015) (to be published)
10. Bormer, T., Wagner, M.: Towards testing a verifying compiler. In: International Conference on Formal Verification of Object-Oriented Software (FoVeOOS) Pre-Proceedings, pp. 98–112. Karlsruhe Institute of Technology (2010)
11. Cohen, E., Dahlweid, M., Hillebrand, M., Leinenbach, D., Moskal, M., Santen, T., Schulte, W., Tobies, S.: VCC: a practical system for verifying concurrent C. In: Berghofer, S., Nipkow, T., Urban, C., Wenzel, M. (eds.) TPHOLs 2009. LNCS, vol. 5674, pp. 23–42. Springer, Heidelberg (2009)
12. Dijkstra, E.W.: Guarded commands, nondeterminacy and formal derivation of programs. Commun. ACM **18**, 453–457 (1975)
13. Dorigo, M., Birattari, M., Stutzle, T.: Ant colony optimization. IEEE Comput. Intell. Mag. **1**, 28–39 (2006)
14. Filliâtre, J.-C., Marché, C.: Multi-prover verification of C programs. In: Davies, J., Schulte, W., Barnett, M. (eds.) ICFEM 2004. LNCS, vol. 3308, pp. 15–29. Springer, Heidelberg (2004)
15. Jacobs, B., Poll, E.: Java program verification at nijmegen: developments and perspective. In: Futatsugi, K., Mizoguchi, F., Yonezaki, N. (eds.) ISSS 2003. LNCS, vol. 3233, pp. 134–153. Springer, Heidelberg (2004)
16. Kuhn, D.R., Wallace, D.R., Gallo, A.M.: Software fault interactions and implications for software testing. IEEE Trans. Softw. Eng. **30**, 418–421 (2004)
17. Nipkow, T., Paulson, L.C., Wenzel, M. (eds.): Isabelle/HOL. LNCS, vol. 2283. Springer, Heidelberg (2002)
18. von Oheimb, D.: Hoare logic for Java in Isabelle/HOL. Concurrency Comput. Pract. Experience **13**, 1173–1214 (2001)
19. Wagner, M.: Maximising axiomatization coverage and minimizing regression testing time. In: IEEE Congress on Evolutionary Computation (CEC), pp. 2885–2892 (2014)
20. Zhu, H., Hall, P.A.V., May, J.H.R.: Software unit test coverage and adequacy. ACM Comput. Surv. **29**, 366–427 (1997)