Heuristically Creating Test Cases for Program Verification Systems

Bernhard Beckert[1], Thorsten Bormer[1], Markus Wagner[2]

[1] Department of Informatics, Karlsruhe Institute of Technology
beckert@kit.edu, bormer@kit.edu

[2] Evolutionary Computation Group, School of Computer Science, The University of Adelaide
markus.wagner@adelaide.edu.au

Abstract

The correctness of program verification systems is of great importance, as they are used to formally prove that safety- and security-critical programs follow their specification. This correctness needs to be checked and demonstrated to users and certification agencies. One of the contributing factors to the correctness of the whole verification system is the correctness of the background axiomatization, which captures the semantics of the target program language. We present an optimisation framework for the maximization of the proportion of the axiomatization that is used ("covered") during testing of the verification tool. We show how test cases for regression tests can be created based on existing ones, as the initial creation of test cases is a very time consuming process. Our study shows that the combination of different heuristics leads to a diverse set of test cases, which in turn increases the axiomatization coverage. This translates into a significant increase in trust in the program verification system.

1 Introduction

Motivation. 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, 9].

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 Bali project [16], the LOOP project [12], and the Mobius project [2], all aimed at the development of fully verified verification systems. One may employ formal methods to prove a system or its calculus to be correct. But—as for any other type of software system—testing and cross-validation are of great importance; this is further discussed in [5].

Metaheuristic and other search-based approaches to test case generation have been in use for over a decade (see [14, 17] for an overview). In our situation of verification system testing, 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.

Within the verification systems, the so-called *axiomatization* carries the formal definitions of the target program language, among other things. This 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 formally verified.

Topic. 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 large number of axioms (typically 100's) and the time consuming verification process (sometimes minutes) make this a challenging problem for iterative search approaches.

We present a framework that allows to increase the axiomatization coverage for regression testing of verification systems. We focus on systematic local searches with randomized components, as the time-consuming coverage determination does not allow for approaches that typically require many evaluations,

such as population-based evolutionary algorithms or ant-colony optimization [1, 10]. Furthermore, the vast number of infeasible ways of reusing existing test cases renders the problem inappropriate for disruptive approaches, such as simulated annealing and even the simple (1+1) evolutionary algorithms.

The structure of this paper is as follows: First, Section 2 outlines the specific problem to be solved in detail, with its formulation as an optimisation problem following in Section 3. In the subsequent Section 4, our metaheuristic approaches to the problem are described, with results being presented and discussed in Section 5. The paper concludes in Section 6 with a summary of key findings and a description of potential future areas of work.

2 Target of Optimization: Program Verification Systems

Modern Program Verification Tools. Every program verification system has to perform (at least) two rather separate tasks: (a) handling the program-language-specific and specification-language-specific constructs, and reducing or transforming them to classical logic expressions, (b) theory reasoning and reasoning in classical logics, for handling the resulting expressions and statements over data types. One can either handle these tasks in one monolithic logic/system, or one can use a combination of subsystems.

In this paper, we concentrate on 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 [8] and Caduceus [11] allow only this type of interaction, other such as the KeY tool [4], offer in addition a mode where user interaction is possible also during the proof construction stage. For the rest of the paper, we restrict all test cases to be provable without human interaction, due to practical reasons.

Program verification tools have to capture the program language semantics of the programs to be verified. In some tools this information is mostly stored as one huge axiomatization (e.g., as with logical frameworks like Isabelle/HOL [15]). Then, the actual implementation part of the tool itself can be kept relatively small. Other tools (e.g., some static checkers) implicitly contain most of the programming language semantics in their implementation.

To assure 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 the axiomatization coverage was as low as 1% for some tests (for the given verification system), while code coverage was never less than 25%. This means 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.

Test Cases. We consider in this paper 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. In the following, we concentrate on functional tests that can be executed automatically, i.e., usability tests and 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 other 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 resource (time and space).

Correctness Properties. Verification systems can be tested for soundness (*"The program P indeed satisfies the specification $SPEC$, whenever the output for a verification problem P+$SPEC$ is 'proved'."*) and for completeness (*"If the program is correct with respect to its given requirement specification REQ, then some auxiliary specification AUX or other required user input exists allowing to prove this"* [3]). Interestingly, it is difficult to test for soundness, as a soundness test case should *not* be provable.[1] For completeness testing, it is easier to construct tests, albeit still far from being trivial. To reveal a completeness problem, a test case must consist of a program P with annotations $REQ \cup AUX$ such that (a) P satisfies $REQ \cup AUX$ and (b) the annotations are strong enough to prove this, i.e., the expected output is "proved".

If the observed output is "not provable", then a completeness failure is revealed. The situation is similar with an observed output "timeout". In that case, the system may just be slower as expected or, worse, there may be no proof using AUX or, worst of all, there may be no proof at all for any annotation set AUX'. For both kinds of incorrect output ("not provable" and "timeout") the developer has to further investigate what kind of failure occurred.

One may consider incompleteness of a verification tool to be harmless in the sense that it is noticeable: the user does not achieve the desired goal of constructing a proof and thus knows that something is wrong. In practice, however, completeness bugs can be very annoying, difficult to detect, and time-consuming. A user may look for errors in the program to be verified or blame the annotation AUX when no proof is found for a correct program and try to improve AUX, while in fact nothing is wrong with it. It is therefore very important to systematically test for completeness bugs.

3 Problem Formulation

Our goal is to increase the amount of testing done in a structured way. In this section, we present how we determine the amount of testing done, and how we intend to improve it.

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 to which extent 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 or proof attempts 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.

As mentioned above, we focus on completeness testing, and consequently 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 the subset, the verifier 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 $Th' \subsetneq Th$.*

[1] For an in-depth discussion on how one can test for different kinds of failures and correctness properties we refer the interested reader to [6].

As a consequence, the axiom coverage of a test suite with respect to a system depends on resource constraints (e.g., number of proof steps allowed, timeout or memory limitations) and the implementation of the verification system, most notably the proof search strategy. In addition, axiom coverage of a test suite has to be recomputed not only when the axiomatization or test suite changes but also whenever parts of the implementation of the verification tool relevant for proof search are modified.

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

Computing Axiomatization Coverage in Practice. We have implemented a framework that allows for the automated execution and evaluation of tests for verification systems that computes the completeness version of axiomatization coverage.

To compute an approximation of the axiom coverage for a completeness test case $P{+}(REQ \cup AUX)$, the procedure is as follows. In a first step, $P{+}(REQ \cup AUX)$ is verified with the verification tool using the complete axiom base available. Besides gathering information on resource consumption of this proof attempt (e.g., number of proof steps and time needed), information on which axioms are actually used in the proof are recorded as set T.[2] In a reduction step, we start from the empty set C of covered axioms. For each axiom t in the set of axioms T used in the first proof run, an attempt to prove $P{+}(REQ \cup AUX)$ using axioms $C \cup (T \setminus \{t\})$ is made. If the proof does not succeed, t is added to set C. Axiom t is removed from T and the next proof iteration starts until $T = \varnothing$.

After a single iteration of this computation, the resulting set of axioms C is only an approximation of the coverage of $P{+}(REQ \cup AUX)$, as not every applied axiom was not necessarily crucial in the proof process. This is the approach taken in [6]. In contrast to this, we repeat the above procedure with C as input as long as the result is different from the input. Eventually, this fixed-point algorithm finds a true minimal set of axioms necessary to construct the proof.

It currently takes several minutes to compute a single minimal axiom set for an average test case. This is acceptable if the coverage is not computed too often, but a considerable speed-up should be possible using heuristics for choosing the axioms to remove from the set. Divide and conquer algorithms, e.g., akin to binary search, seem to be suited to reduce computation times at first glance. However, they do not help in practice: as the reduction step does not start from the whole axiomatization but rather from the subset T of axioms actually used in a proof, only relatively few axioms remain that are *not* covered and can be discarded in the iterative proof runs. For divide and conquer algorithms to be successful, large sets of axioms that could be discarded at once are needed.

In our case, where we will iteratively maximize the axiomatization coverage, the computation of a single minimal axiom set is similar to what is often referred to as "an evaluation". As we shall see in Section 5, evaluation times typically take several minutes, but can in very few cases exceed 24 hours (despite adjusted internal timeouts). Consequently, this renders our problem infeasible for many population-based approaches and other iterative approaches that would require large numbers of evaluations.

Maximizing Axiomatization Coverage. We propose to increase the amount of testing done, by generating additional tests from existing tests. 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. 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.

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

With our additional generated test cases, it is for example possible to identify axioms that are still not used at all, for which the reasons can then be investigated separately. Such analysis can help to focus the efforts of manual test creation to parts of the axiomatization that are not exercised.

One could ask whether it is possible to maximize the number of axioms covered in a more direct way. We conjecture that it is either not possible, or just with significant effort. One would need to know in advance which combinations of axioms would "just suffice", and this would require an oracle.

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. The 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; this is typically the case.

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—in our case, we 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 [4]. 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*.

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 that we will call *axioms* to facilitate reading. However, not all of them are available at a time when performing a proof, as some exist in several versions, depending on proof options chosen (e.g., handling integer arithmetic depends on whether integer overflows are to be checked or not).

The automatic proof search is combined with interactive steps of the user, in case a proof is not found automatically. As already mentioned, the interactive part of KeY is irrelevant to us, as we restrict test cases to those that can be proven automatically—otherwise, finding the a minimal set of taclets needed to prove a program correct is infeasible.

Results of a verification attempt in KeY are the following: either the generated Java Card Dynamic Logic formula is valid and KeY is able to prove it; or the generated formula is not valid and the proof cannot be closed; or KeY runs out of resources.

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 contains already pairs $P{+}(REQ \cup AUX)$, such that we can focus on the search for whitelists. In previous experiments, we have found out that 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 choose conservative approaches in which we try to use the the knowledge gained so far. In addition, we introduce varying degrees of randomization, to allow for different search directions.

All approaches have the following idea in common. 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 1520 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. Consequently, a new minimal set M' can be found, which will of course 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.

The approaches differ in the way they shorten the whitelists, when given a minimal set of axioms M:

1. APPROACH 1 "in order enumeration": the elements in M are explored in lexicographical order in a depth-first fashion. This naive approach is structured and the resulting sequence of minimal sets allows for an easy analysis of dependencies.

2. APPROACH 2 "random order enumeration": the elements in M are explored in random order. The axiomatization contains groups of axioms that can be used interchangeably in the proof search. In case several unrelated test cases use the same groups of axioms, then chances are that this approach will not rediscover the same replacements for group of axioms over and over again.

3. APPROACH 3 "random step sizes": up to six randomly picked elements from M are removed from the current whitelist. Iteratively, the step size is reduced by 50% (rounded up) in case the whitelist does not allow for a successful proof, because it is too restrictive. APPROACH 2 is used as a fall-back strategy. Even though verification attempts are more likely to fail, the whitelists should become shorter, thus motivating the verification system to "work around" our artificially imposed restrictions in order to construct a proof.

Our general approach is in stark contrast to shortening the whitelists by randomly picking axioms from the axiomatization: removing a previously unused axiom from the whitelist will result in the very same minimal set over and over again. In contrast to this, we try to remove axioms that have been used before.

Lastly, as we keep track of the generated whitelists, we prevent the repeated computation of minimal sets for a given pair pairs $P+(REQ \cup AUX)$.

5 Experiments

Using our testing framework, we automatically execute the test cases contained in KeY's test suite and measure the taclet coverage. The code is available upon request, and will be made publicly available.

The KeY source distribution provides a test suite containing 335 test cases (as of 16 August 2012) of which 319 test cases testing verification of functional properties—the other 16 are soundness tests or are concerned with the verification of information flow properties and were omitted due to resource constraint. The complexity of the proof obligations ranges from simple arithmetic problems to small Java programs testing single features of Java, up to more complex programs and properties taken from recent software verification competitions.

This test and all subsequent runs are performed on AMD Opteron 250 CPUs (2.4GHz), on Debian GNU/Linux 5.0.8, with Java SE RE 1.7.0. The computation time for each of the 319 test cases is limited to 24h for each approach. 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.

Example. To start our presentation of the results, we exemplarily investigate KeY's original test case `heap/list/ArrayList.ArrayListIterator_inv.key`. It's purpose is to test the iterator functionality of an array list implementation. It is of particular interest, as the shortest successful whitelist across all approaches is found for it. Several statistics are listed in Table 1. Note that the first minimal set contains only 34 elements, and that all approaches have been able to at least double this number.

Interestingly, all approaches rarely backtrack, as we would otherwise see more "branches" in the sequence of whitelist lengths. This is a strong indicator that the search for new minimal sets is not finished yet. Many backtracking points are still waiting to be explored.

The significant drop of the trials that APPROACH 2 performs is due to axioms being removed that (1) cause a timeout (when missing), and (2) have previously improved the performance of the proof strategy.

Approach	successful trials	covered axioms	sequence of whitelist length development
APPROACH 1	43/878	80	$\langle 1520, \ldots, 1481, \mathit{1487}, 1486, 1485 \rangle$
APPROACH 2	45/388	80	$\langle 1520, \ldots, 1514, \mathit{1514}, \ldots, 1484, \mathit{1485}, \ldots, 1481 \rangle$
APPROACH 3	17/864	116	$\langle 1520, 1514, 1511, 1510, 1507, 1504, 1503, 1500, \ldots,$ $1497, 1491, 1488, \ldots, 1486, 1483, \ldots, 1464, 1461, 1460 \rangle$

Table 1: Example test case. Listed are the ratios of successful to failed whitelist trials, the number of axioms covered by the union of all minimal sets, and the sequence of whitelist lengths that were discovered (*backtracking points marked*). Dots indicate decrements by one up to the next shown value.

	APPROACH 1	APPROACH 2	APPROACH 3	Union
axioms covered in the first minimal sets	475 (31%)	473 (31%)	475 (31%)	479 (32%)
axiom usage in the first minimal sets	7,375	7,374	7,370	7,512
axioms covered in all minimal sets	616 (41%)	610 (40%)	594 (39%)	643 (42%)
axiom usage in all minimal sets	15,278	15,855	14,632	18,547
shortest whitelist found	1469	1471	1460	1460

Table 2: Coverage statistics. The *first minimal sets* refer to those found first by the approaches, which initially use all 1520 axioms. Axiom usage is the total count of axioms used by the respective sets. The *Union* is the result of considering all three approaches together.

APPROACH 3 tries out many larger reductions of the current whitelist, and a huge proportion of them is not successful. If it would be possible to establish dependencies between the axioms, and logical groups, then it should be possible to either identify these in advance, or to learn these on the fly. Consequently, the time spent on extensions that are unlikely to work (because "essential" rules are to be left out) may be reduced, thus increasing the efficiency of the framework.

It is obvious that the different approaches exercise the verification system in different ways. Because of their nature, APPROACH 1 and 2 iteratively block out larger and larger parts of the axiomatization. In contrast to that, APPROACH 3 is able to rapidly decrease the lengths of the whitelists.

Axiomatization Coverage Results. The coverage statistics of the different approaches are listed in Table 2. In [6], an approximated coverage of 585 (out of 1520) is reported. This turns out to be drastically overestimating the actual ~ 474 that is achieved by iteratively running the reduction algorithm for the same test files. The number 474 furthermore represents the result of the naive approach, where the full set of 1520 axioms is used and no alternatives are sought. Thus, this is our base value.[3]

The individual approaches improve the total coverage by about 10% each. When considering all approaches together, then the initial coverage of about 474 axioms is increased to 643 axioms through the use of whitelists. This means that the framework is able to improve the achievable coverage autonomously by about 35%, without requiring a verification engineer to write a single new test case.

Figures 1–4 show additional statistics about the frequency of the axioms covered, and about the lengths of the whitelists (from top to bottom: APPROACH 1–3). The overall distribution of the axiom coverage is almost identical (see Figure 1). Often, APPROACH 1 and 2 exhibit very similar statistics (e.g., see Figures 2–4). APPROACH 3 deviates from the other two in certain sections, as different "local shapes" indicate. Such histograms, when split by taclet group, allow us to compare the quality of the test suite with respect to the different groups. In [6] we were able to locate underrepresented taclet groups, e.g., relevant for Java assertions or the bigint primitive type of JML (with coverage of each group below 10%). This coarse classification already allows to focus the effort of writing new test cases on constructing specific tests for seldomly covered taclet groups. Additional data mining, e.g., in the

[3]The used KeY-Version uses certain taclets indeterministically. However, we observed no significant consequences on the overall number of taclets covered in several repetitions of our experiments.

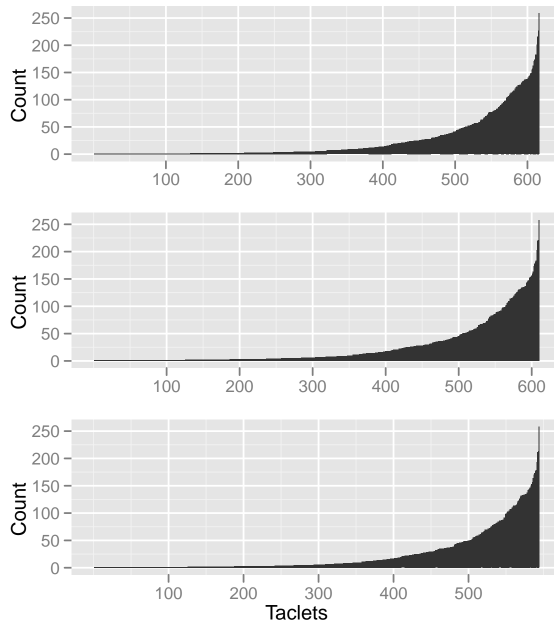

Figure 1: Axiom coverage counts. y-axis: number of test cases an axiom is covered by. On the x-axis: axioms at least covered by one test case, sorted by y values. (f.t.t.b.: APPROACH 1–3)

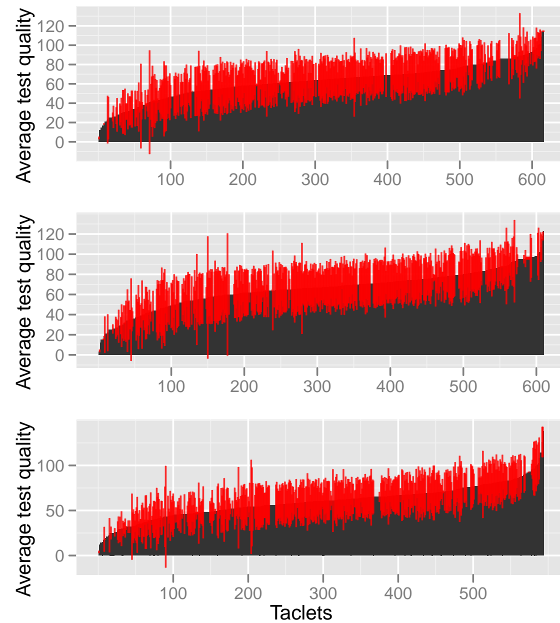

Figure 2: Average test case selectivity by axiom. In black: average selectivity of all test cases covering an axiom. Deviation of this value from the average is shown in red. (f.t.t.b.: APPROACH 1–3)

y covers axioms not covered by:	APPROACH 1	APPROACH 2	APPROACH 3
APPROACH 1	—	25	35
APPROACH 2	19	—	31
APPROACH 3	13	15	—

Table 3: Differences between the minimal sets found. For example, APPROACH 3 covers 15 axioms that are not covered by APPROACH 2.

form of clustering, can be done in order to group similar test cases together to identify commonalities. This is, however, beyond the scope of this paper.

The need for broader test cases, covering several combinations of axioms, is supported by studies (e.g. [13]), which show that software failures in a variety of domains are often caused by combinations of several conditions. Specialized test cases, in comparison, might simplify testing different aspects of one taclet by being able to better control the context an axiom will presumably be applied in the proof. As a measure for this, we define the *selectivity* of a test case as the number of axioms covered by the test. The current state of the KeY test suite with respect to this selectivity criterion is shown in Figure 2. For each axiom, the average selectivity of all test cases covering this taclet is shown, together with the deviation from the average. The leftmost axioms in this diagram are good candidates for which additional test cases might be needed, as they are only covered by specialized test cases. Also axioms with a high selectivity average of the corresponding test cases but low deviation indicate need for improvements, as only broad test cases cover the axioms.

Coming back to the optimisation approaches, we can see in Table 3 that they complement each other. When we combine Approaches 1 and 2, they cover a total of 635 axioms. Thus, one can see that a small degree of randomization contributes to the diversity of the outcomes. When combining the randomized Approaches 2 and 3, then only 625 axioms are covered. It seems that the search directions differ significantly, because APPROACH 2 covers 31 axioms that are not covered by the other approach. On the other hand, even though APPROACH 3 yields the lowest coverage values amongst all, its outcome is still complementary to those of the otherwise relatively similar Approaches 1 and 2.

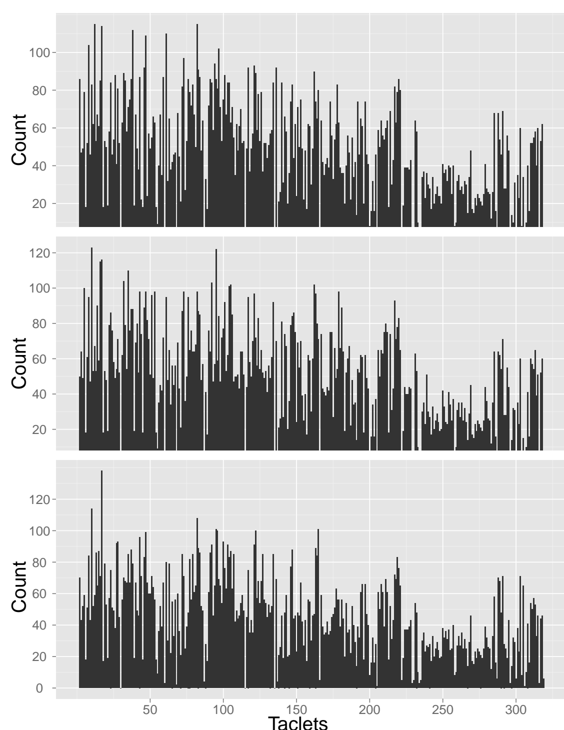

Figure 3: For each of the 319 tests (x-axis) the total number of axioms covered across all minimal sets is shown. (f.t.t.b.: APPROACH 1–3)

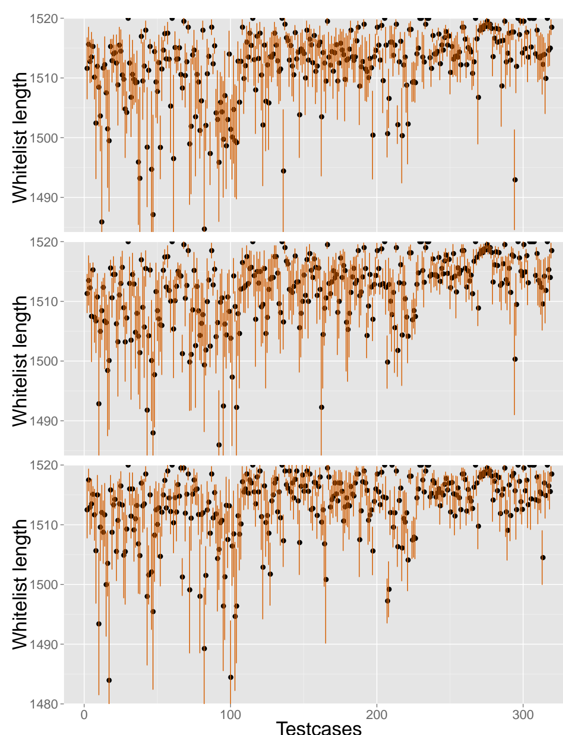

Figure 4: For each of the 319 tests (x-axis) the found whitelists' mean length and the standard deviation is shown. (f.t.t.b.: APPROACH 1–3)

Three test cases never produced a minimal set within the allotted 24 hours. The reason for this is that in all three cases the number of axioms used in the first successful proof is very large (> 140). Then, the iterative reduction process becomes very time consuming, and cannot finish within the allotted time. We will allot additional time to these three in the future, as they have the potential to contribute significantly to the coverage due to the complexity of the tests.

6 Conclusions and Future Work

In this paper, we address the problem of increasing the axiomatization coverage when rigorously testing verification systems. We introduce several approaches that allow us to reuse the existing test cases without generating new cases by hand. The reuse is implemented through varying the restriction on which axioms can be used for proof attempts.

The experiments reveal several interesting insights. It is important to note that it is impractical to manually impose the restrictions on verification systems under which correctness should be proven. Randomly generated restrictions typically are too strong and make it impossible to find a proof. Our local search approaches, however, explore the space of restrictions in a structured way. When generating the restrictions, it is beneficial to allow for random decisions and for a degree of disruption. An algorithm that does both is able to obtain a considerably higher axiomatization coverage. Even though our experiments are computationally expensive, the restrictions found so far can be reused in future coverage computations as seeds, without having to rerun the entire search again.

In the future, we will continue to research in the following areas:

1. The computation of a minimal set of axioms is time consuming. If it would be possible to establish dependencies between the axioms, and logical groups, then it should be possible to either identify these in advance, or to learn these on the fly. Consequently, the time spent on the coverage calculations may be reduced significantly, thus increasing the efficiency of the framework.

2. Failures in a variety of domains are often caused by combinations of several conditions (see studies

like [13]). We plan to combine combinatorial testing with combinatorial search techniques. There, combinations of language features and axioms are 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] Thomas Back, David B Fogel, and Zbigniew Michalewicz. *Handbook of evolutionary computation*. IOP Publishing Ltd., 1997.

[2] Gilles Barthe, Lennart Beringer, Pierre Crégut, Benjamin Grégoire, Martin Hofmann, Peter Müller, Erik Poll, Germán Puebla, Ian Stark, and Eric Vétillard. *MOBIUS: Mobility, Ubiquity, Security*, volume 4661 of *LNCS*. Springer, 2006.

[3] Bernhard Beckert, Thorsten Bormer, and Vladimir Klebanov. Improving the usability of specification languages and methods for annotation-based verification. In Bernhard Aichernig, Frank S. de Boer, and Marcello Bonsangue, editors, *FMCO 2010. State-of-the-Art Survey*, volume 6957 of *LNCS*. Springer, 2011.

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

[5] Bernhard Beckert and Vladimir Klebanov. Must program verification systems and calculi be verified? In *3rd International Verification Workshop (VERIFY), Workshop at Federated Logic Conferences (FLoC)*, pages 34–41, 2006.

[6] Bernhard Beckert, Markus Wagner, and Thorsten Bormer. A metric for testing program verification systems. In *7th International Conference on Tests and Proofs (TAP)*, 2013. Accepted for publication, preliminary version: `http://tinyurl.com/axiomatizationcoverage`.

[7] Béatrice Bérard, Michel Bidoit, Alain Finkel, François Laroussinie, Antoine Petit, Laure Petrucci, and Philippe Schnoebelen. *Systems and software verification: model-checking techniques and tools*. Springer Publishing Company, Incorporated, 2010.

[8] Ernie Cohen, Markus Dahlweid, Mark Hillebrand, Dirk Leinenbach, Michał Moskal, Thomas Santen, Wolfram Schulte, and Stephan Tobies. VCC: A practical system for verifying concurrent C. In *TPHOLs'09*, volume 5674 of *LNCS*, pages 23–42. Springer, 2009.

[9] Edsger W Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *Communications of the ACM*, 18(8):453–457, 1975.

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

[11] Jean-Christophe Filliâtre and Claude Marché. Multi-prover verification of C programs. In *Formal Methods and Software Engineering*, LNCS 3308, pages 15–29. Springer, 2004.

[12] Bart Jacobs and Erik Poll. Java program verification at Nijmegen: Developments and perspective. *LNCS*, 3233:134–153, 2004.

[13] 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.

[14] Phil McMinn. Search-based software test data generation: a survey. *Software Testing, Verification and Reliability*, 14(2):105–156, 2004.

[15] Tobias Nipkow, Lawrence C Paulson, and Markus Wenzel. *Isabelle/HOL: a proof assistant for higher-order logic*, volume 2283. Springer, 2002.

[16] David von Oheimb. Hoare logic for Java in Isabelle/HOL. *Concurrency and Computation Practice and Experience*, 13(13):1173–1214, 2001.

[17] Joachim Wegener, André Baresel, and Harmen Sthamer. Evolutionary test environment for automatic structural testing. *Information and Software Technology*, 43(14):841–854, 2001.