2014 IEEE Congress on Evolutionary Computation (CEC)
July 6-11, 2014, Beijing, China

Maximising Axiomatization Coverage and Minimizing Regression Testing Time

Markus Wagner

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. 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 a framework for the maximization of the proportion of the axiomatization that is used ("covered") during testing of the verification tool. The diverse set of test cases found not only increases the trust in the verification system, but it can also be used to reduce the time needed for regression testing.**

I. 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 [5, 8, 11].

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 [18], the LOOP project [14], and the Mobius project [3], all aimed at the development of fully verified verification systems. Similarly, components of the KeY verification system [5] for Java were verified using the Maude tool [1]. 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 [4].

Metaheuristic and other search-based approaches to test case generation have been in use for over a decade (see [16, 19] 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.

Markus Wagner is an IEEE Member and is with the Optimisation and Logistics Group, School of Computer Science, The University of Adelaide (email: markus.wagner@adelaide.edu.au).

Our goal is to increase the proportion of the axiomatization that is actively used in successful verification attempts [7, 9]. 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 [2, 12]. 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 article is as follows: First, we outline the specific problem in Section II, and in Section III we formulate it as an optimization problem. In the subsequent Section IV, we describe our approaches, and we present and discuss the results in Section V. The paper concludes in Section VI with a summary of key findings and a description of potential future areas of work.

II. Target of Optimization: Program Verification Systems

A. 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 article, 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 [10] and Caduceus [13] allow only this type of interaction, other such as the KeY tool [5], offer in addition a mode where user interaction is possible also during the proof construction stage.

978-1-4799-1488-3/14/$31.00 ©2014 IEEE

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 and it can be kept generic and thus comparatively small.

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 [7] 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.

B. Test Cases

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—more difficult than specifying the correct behaviour of the verification system as a whole. 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. Which annotations are *compatible* to a program, i.e., which annotation types exist and in which program contexts a particular annotation is allowed, depends on the given annotation language. 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).

For example, let us consider the following test case for KeY, with its Java program code in Lines 4-8 and its post-condition in Line 11. The test goal is to check if KeY correctly deals with a *division by zero*:

```
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 }
```

Note that the test case writer chose a particular way to test a single feature, Consequently, KeY needs to reason, for example about variable initialisation and exception handling as well, in addition to integer division.

III. Problem Formulation

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

A. 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 [7]. 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 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 the subset, the verifier is not able anymore to prove the correctness of the program.

Definition 1 ([7]): 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 $Th' \subsetneq Th$.

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. We will exploit this lack of uniqueness later-on.

B. 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.[1] 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 the 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 [7]. 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. See Section VI for further ideas.

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 V, 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.

C. Maximizing Axiomatization Coverage

We 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.

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.

IV. 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.

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

A. The KeY System

As the target for our case study we have chosen the KeY tool [5], 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[2] in Java Dynamic Logic, a multimodal predicate logic [5]. 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 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 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.

B. 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. 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 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 "depth-first 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 "depth-first random order enumeration": the elements in M are explored in random order. The motivation for this slight different to APPROACH 1 is the following: (1) the axiomatization contains groups of axioms that can be used interchangeably in the proof search, and (2) 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 "depth-first 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. We use APPROACH 2 as a fall-back strategy, in case all larger "jumps" are unsuccessful. 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.
4) APPROACH 4 "breadth-first in order enumeration": in contrast to APPROACH 1, the elements in M are explored in a breadth-first fashion. Here, the focus is on finding alternative groups of individual axioms first, whereas APPROACH 1 focusses on the creation of shorter and shorter whitelists.
5) APPROACH 5 "breadth-first random order enumeration": analogous to APPROACH 2, the elements are explored in a random order to reduce the number of rediscoveries of equivalent groups of axioms.

Note that APPROACH 1–3 were already presented in [6], and they are now compared with two depth-first approaches.

By construction, all five methods are de facto complete as they perform either depth-first or breadth-first searches. They will exhaustively enumerate the feasible whitelists that can

[2]A sequent has the form $\Gamma \vdash \Sigma$. Both Γ and Σ are sets of logical formulae, where Γ is called antecedent and Σ called succedent. The intuitive semantics of a sequent is that the conjunction of the formulae in the antecedent imply the disjunction of the formulae in the succedent.

Approach	successful trials	covered axioms	sequence of whitelist length development
APPROACH 1	28/651	120	$\langle 1520, \ldots, 1493 \rangle$
APPROACH 2	12/110	91	$\langle 1520, \ldots, 1509 \rangle$
APPROACH 3	8/158	123	$\langle 1520, 1517, 1516, 1514, 1513, 1512, 1509 \rangle$
APPROACH 4	19/42	92	$\langle 1520, 1519 \rangle$
APPROACH 5	21/41	94	$\langle 1520, 1519 \rangle$

TABLE I: 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 (immediate repetitions omitted). Dots indicate decrements by one up to the next shown value.

be reached when starting with the entire whitelist from the initial minimal set.

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 remove axioms that have been used successfully before.

It is important to note that the search space is very "brittle": given a valid solution, only about 5-10% of its neighbours turn out to be valid solutions again. We therefore chose very conservative heuristics that would change very little at a time. If one increases the randomness in an unstructured way, the chances diminish very quickly of (1) hitting a valid solution and (2) guiding the search towards a higher coverage. Actually, as we shall later-on see, our slightly more exploring heuristic APPROACH 3 can in fact contribute to the overall coverage, however, this comes at the cost of a significantly increased failure rate.

Lastly, as we keep track of the generated whitelists, we prevent the repeated computation of minimal sets for a given pair $P{+}(REQ \cup AUX)$. As a side-effect, this collection can later-on be easily reused for regression testing without the need to do the expensive searches again from start.

V. EXPERIMENTS

Using our testing framework, we automatically execute the test cases contained in KeY's test suite and measure the axiomatization coverage.[3]

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 Intel Xeon E5430 CPUs (2.66GHz), 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.

[3]The code is available upon request, and will be made publicly available.

A. Example

To start our presentation of the results, we examplarily investigate KeY's original test case `heap/list/ArrayList.ArrayListIterator_inv.key`. Its purpose is to test the iterator functionality of an array list implementation. Several statistics are listed in Table **??**. Note that the first minimal set contains ("covers") only 34 elements, and that all approaches have been able to at least double this number.

Interestingly, APPROACHES 1–3 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, which is why we included their breath-first variants APPROACH 4 and APPROACH 5. 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 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. APPROACH 3 is able to rapidly decrease the lengths of the whitelists. In contrast to this, APPROACHES 4 and 5 hardly reduce the lengths of the whitelists at all, but explore just the very first branching.

B. Axiomatization Coverage Results

The coverage statistics of the different approaches are listed in Table II. 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. [4]

The individual approaches improve the total coverage by about 6% each. When considering all approaches together, then the initial coverage of about 611 axioms increases to a total of 722 axioms through the use of whitelists. This means that the framework improve the achievable coverage

[4]The used KeY-Version still uses certain taclets indeterministically, despite our introduction of deterministic data structures. However, we observed no significant consequences on the overall number of taclets covered in several repetitions of our experiments.

	APPROACH 1	APPROACH 2	APPROACH 3	APPROACH 4	APPROACH 5	Union
axioms covered in the first minimal sets	611 (40%)	611 (40%)	610 (40%)	613 (40%)	609 (40%)	615 (40%)
axiom usage in the first minimal sets	13,978	14,028	14,081	13,900	13,998	69,982
axioms covered in all minimal sets	701 (46%)	699 (46%)	688 (45%)	687 (45%)	684 (45%)	722 (48%)
axiom usage in all minimal sets	21,358	22,228	21,257	21,823	22,028	108,691
shortest whitelist found	1,480	1,467	1,446	1,512	1,512	1,446

TABLE II: 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 five approaches.

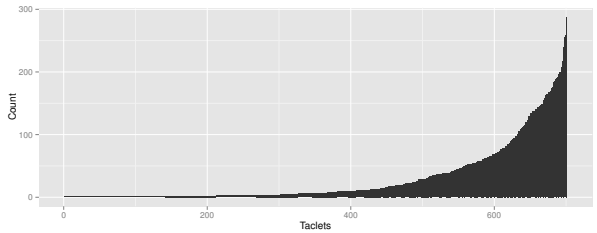

Fig. 1: Axiom coverage counts. y-axis: number of test cases an axiom is covered by. On the x-axis: axioms at least covered by one test case, sorted by y values.

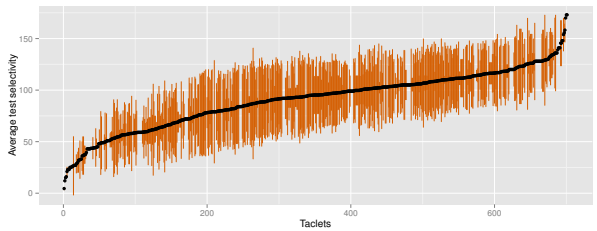

Fig. 2: Average test case selectivity (*"if covered then by how many tests"*) by axiom. In black: average selectivity of all test cases covering an axiom. Deviation of this value from the average is shown in red.

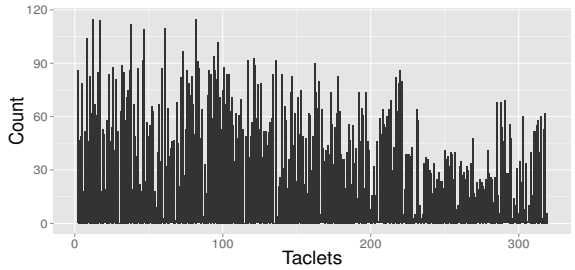

Fig. 3: For each of the 319 tests (x-axis) the total number of axioms covered across all minimal sets is shown.

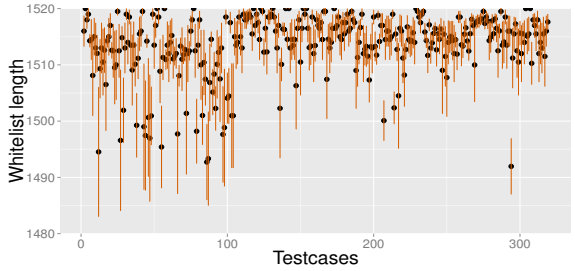

Fig. 4: For each of the 319 tests (x-axis) the found whitelists' mean length and the standard deviation is shown.

autonomously by about 18%, 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. Due to space contraints, we limit ourselves to the results of APPROACH 1. The figures for the other approaches are fairly similar to a human viewer, with slightly different "local shapes". Such histograms, when split by taclet group, allow us to compare the quality of the test suite with respect to the different groups. For example, in [7] underrepresented taclet groups are located, 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 form of clustering, can 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. [15]), 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 the testing of 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 use 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 these axioms.

Coming back to the optimisation approaches, we can see in Table III that they complement each other. For example, when we combine APPROACHES 1 and 2, they cover a total of 719 axioms. Thus, one can see that a small degree of

covered by ↓ but not covered by →	APPROACH 1	APPROACH 2	APPROACH 3	APPROACH 4	APPROACH 5
APPROACH 1	—	20	21	24	28
APPROACH 2	18	—	22	23	24
APPROACH 3	8	11	—	12	17
APPROACH 4	10	11	11	—	8
APPROACH 5	11	9	13	5	—

TABLE III: Differences between the minimal sets found. For example, APPROACH 5 covers 11 axioms that are not covered by APPROACH 1.

randomization contributes to the diversity of the outcomes. When combining the randomized APPROACHES 2 and 3, then only 710 axioms are covered. It seems that the search directions differ significantly, because APPROACH 2 covers 22 axioms that are not covered by the other approach. On the other hand, even though APPROACH 3 yields the lowest coverage values amongst the depth-first approaches, its outcome is still complementary to those of the otherwise relatively similar APPROACHES 1 and 2. Similar observations hold for APPROACHES 4 and 5.

Even though the runtimes are capped at 24 hours, it is possible to enumerate all minimal sets for a total of 41 files of the test suite. Consequently, the differences that we can observe in Table II stem from the remaining 278 files. As a side-effect, we do not have to consider these files again in the future, when searching for additional whitelists —unless the axiomatization itself changes.

In contrast to these 41 test cases, seven other test cases never produced a minimal set within the allotted 24 hours. The reason for this is that in all these 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 in the future, as they have the potential to contribute significantly to the coverage due to the complexity of the tests.

C. Minimization of Regression Testing Time

The goal of regression testing is to uncover new bugs when the system changes. As a result of our previous computations, we extended the number of test cases from 319 to several 10,000's of test cases. However, running all these on a regular basis is very impractical due to the long overall runtime.

Each axiom is used about 150 times on average, albeit with significant deviations from this (see column *Union* in Table II), when the entire set of test cases is run. As this high level of redundancy is not necessary for a fast regression test, we can try to contruct a subset of tests that runs in short time and that achieves the same overall axiomatization coverage. Even though this new set of tests will not exercise the verification system as comprehensively, it is very useful as a quick check if a system change has a significant impact on the axiomatization.

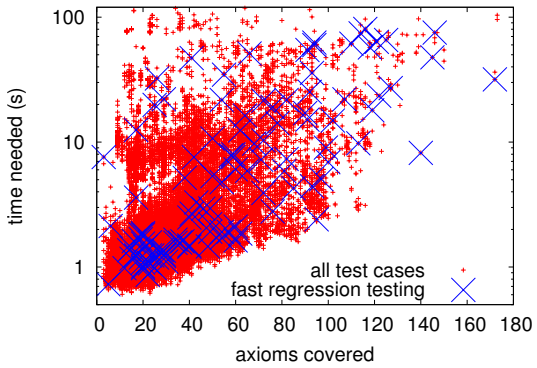

Fig. 5: Test cases: axioms covered and time needed. Shown are (1) all test cases created by our search, and (2) the 115 test cases for the fast regression testing.

In the following, let us consider APPROACH 5—similar statements hold for the other approaches. The computations presented in the previous section resulted in 29,323 minimal sets and consequently the same number of actual test cases for KeY, which is a significant increase from the original 319. In a regression test, these would take 24.2 hours to run and they would cover 45% of KeY's axiomatization.

For a fast regression test, we search for a subset of the 29,323 test cases that still achieves the same overall coverage. Our approach to this problem of runtime minimization works as follows. We start with the test case that covers the most axioms. Then, until the desired coverage is achieved, we iteratively add the test case that results in the maximum increase in totally covered axioms.[5]

The runtime of our subset creation of about 1.5 minutes is a good investment: the resulting reduced set of only 113 test cases finishes in about 27 minutes, which is a significant improvement over the original 24.2 hours.

Finally, in order to achieve the overall achieved coverage of 722 axioms, we consider the entire set of test cases produced by all approaches. The set of all test cases and the constructed subset for fast regression testing are shown in Figure 5. The subset of 115 test cases takes 53 minutes to run and covers 722 axioms. Several of the included test cases are very time consuming, but they also help to quickly achieve the desired coverage. Compared to the starting point of our investigations, this is a significant increase in test quality (according to our measurement) and a significant decrease in regression testing time.

VI. CONCLUSIONS AND FUTURE WORK

In this article, we address the problem of increasing the axiomatization coverage when rigorously testing verification systems. We compare 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

[5]Other investigated approaches have been unsuccessful: either the final subset's runtime was too long, or the runtime of the heuristic was too long.

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 we generate the restrictions, we allow for random decisions and for a degree of disruption. An algorithm that does both is able to obtain a considerably more diverse set of tests. 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.

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. Afterwards, an experiment will be conducted where we systematically write specific test cases aimed to increase the axiomatization coverage for specific axioms. If our assumption that axiomatization coverage is a useful measure is right, we should be able to find further bugs with these tests.
2) The computation of a minimal set of axioms is time consuming. If it is possible to establish dependencies between the axioms, and logical groups, then it will 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.
3) Failures in a variety of domains are often caused by combinations of several conditions (see studies like [15]). 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.

ACKNOWLEDGEMENTS

We thank Mojgan Pourhassan for her very valuable feedback on APPROACH 1–3 that lead to APPROACH 4 and APPROACH 5.

REFERENCES

[1] W. Ahrendt, A. Roth, and R. Sasse. Automatic validation of transformation rules for Java verification against a rewriting semantics. In *LPAR'05*, Vol. 3835, pp. 412–426. Springer, 2005.

[2] T. Back, D. B. Fogel, and Z. Michalewicz. *Handbook of evolutionary computation*. IOP Publishing Ltd., 1997.

[3] G. Barthe, L. Beringer, P. Crégut, B. Grégoire, M. Hofmann, P. Müller, E. Poll, G. Puebla, I. Stark, and E. Vétillard. *MOBIUS: Mobility, Ubiquity, Security*, Vol. 4661 of *LNCS*. Springer, 2006.

[4] B. Beckert and V. Klebanov. Must program verification systems and calculi be verified? In *3rd Int. Verification Workshop (VERIFY), Workshop at Federated Logic Conferences (FLoC)*, pp. 34–41, 2006.

[5] B. Beckert, R. Hähnle, and P. H. Schmitt, editors. *Verification of Object-Oriented Software: The KeY Approach*, Vol. 4334 of *LNCS*. Springer, 2007.

[6] B. Beckert, T. Bormer, and M. Wagner. Heuristically creating test cases for program verification systems. In *10th Metaheuristics Int. Conference (MIC)*, Singapore, 2013.

[7] B. Beckert, M. Wagner, and T. Bormer. A metric for testing program verification systems. In *7th Int. Conference on Tests and Proofs (TAP)*, Vol. 7942 of *LNCS*, pp. 56–75, 2013.

[8] B. Bérard, M. Bidoit, A. Finkel, F. Laroussinie, A. Petit, L. Petrucci, and P. Schnoebelen. *Systems and software verification: model-checking techniques and tools*. Springer, 2010.

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

[10] E. Cohen, M. Dahlweid, M. Hillebrand, D. Leinenbach, M. Moskal, T. Santen, W. Schulte, and S. Tobies. VCC: A practical system for verifying concurrent C. In *Int. Conference on Theorem Proving in Higher Order Logics (TPHOLs)*, Vol. 5674 of *LNCS*, pp. 23–42. Springer, 2009.

[11] E. W. Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *Communications of the ACM*, 18:453–457, 1975.

[12] M. Dorigo, M. Birattari, and T. Stutzle. Ant colony optimization. *Computational Intelligence Magazine, IEEE*, 1:28–39, 2006.

[13] J.-C. Filliâtre and C. Marché. Multi-prover verification of C programs. In *Formal Methods and Software Engineering*, LNCS 3308, pp. 15–29. Springer, 2004.

[14] B. Jacobs and E. Poll. Java program verification at Nijmegen: Developments and perspective. *LNCS*, 3233: 134–153, 2004.

[15] D. R. Kuhn, D. R. Wallace, and A. M. Gallo. Software fault interactions and implications for software testing. *IEEE Transactions on Software Engineering*, 30:418–421, 2004.

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

[17] T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL: a proof assistant for higher-order logic*, Vol. 2283. Springer, 2002.

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

[19] J. Wegener, A. Baresel, and H. Sthamer. Evolutionary test environment for automatic structural testing. *Information and Software Technology*, 43:841–854, 2001.