A Metric for Testing Program Verification Systems*

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

[1] Department of Informatics, Karlsruhe Institute of Technology
beckert@kit.edu, bormer@kit.edu
[2] School of Computer Science, The University of Adelaide
markus.wagner@adelaide.edu.au

Abstract. The correctness of program verification systems is of great importance, and it 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, respectively the correctness of calculus rules. In this paper, we examine how testing verification systems is able to provide evidence for the correctness of the rule base or the axiomatization. For this, we present a new coverage criterion called axiomatization coverage, which allows us to judge the quality of existing test suites for verification systems. We evaluate this coverage criterion at two verification tools using the test suites provided by each tool.

1 Introduction

Motivation. Correctness of program verification systems is imperative if they are to be used in practice. 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 is of great importance.

In this paper, we bring together proofs and tests not as a combination of both used on a program to be validated, but rather to increase software quality by improving conclusiveness of the verification tool itself. Traditional testing techniques alone are insufficient for this purpose—the typical properties and particularities of program verification systems have to be taken into consideration when designing test suites. It is relevant, for example, that verification systems usually do not just consist of an implementation in an imperative programming language but also include axioms and rules written in a declarative language.

The testing process employed must be systematic and the quality of test suites has to be evaluated. Objective criteria, such as coverage measures, are needed to demonstrate the dependability of verification systems to users and certification agencies.

* Work partially funded by the German Federal Ministry of Education and Research (BMBF) in the framework of the Verisoft XT project under grant 01 IS 07 008. The responsibility for this article lies with the authors.

Topic and Structure of this Paper. In this paper, we present a new coverage criterion for testing program verification systems, called axiomatization coverage. We conducted experiments on two verification tools that measure axiomatization coverage of existing test suites, in order to assess the significance of the proposed coverage metric. Our focus is on system tests (as opposed to testing components of the tools); and we test for functional correctness (not usability etc.). Moreover, we only consider tests that can be executed automatically.

The structure of this paper is as follows: First, in Section 2, we clarify what verification systems we consider and discuss their relevant properties. Section 3 is concerned with the test cases we use and the general set-up for testing. In Section 4, we examine the different correctness properties for which we test and explain which kinds of tests relate to what properties. In Section 5, we define a new notion of test coverage for the declarative (axiomatic) part of verification systems. We report on two case studies in Section 6, in which we have evaluated the new test coverage criterion using test suites for two verification systems. Then, in Section 7, we put our work into the context of related work. Finally, in Section 8, we draw conclusions and discuss future work.

2 Target of Evaluation: 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 logical expressions, (b) theory reasoning and reasoning in 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 a paradigm of user interaction with the verification tool termed *auto-active verification*, which is used by tools such as VCC [9], or the Jessie plug-in of Frama-C [16] (if automatic provers are used as backend). 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.

Both tools mentioned above use several subsystems. A more monolithic approach is taken by the Java Card verification tool KeY [6]; in addition, in KeY, user interaction is possible also during the proof construction stage. In many cases however, KeY can be used in an auto-active manner without relying on user input during proof construction. For the rest of the paper, we restrict all test cases to be provable without interaction. This allows us to treat VCC and KeY in the same way.

Imperative Versus Declarative System Components. 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 (as,

e.g., with logical frameworks like Isabelle/HOL) and the implementation part 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.

3 Test Cases for Program Verification Systems

As said above, the tests we consider in this paper are 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, such unit tests cannot replace testing the integrated verification system. Moreover, not all components are easy to test individually. For example, it is possible (and useful) to unit-test an SMT solver that is used as part of a verification system. But the verification condition generator is hard to test separately as it is very difficult to specify its correct behavior—more difficult, in fact, than specifying the correct behavior of the verification system as a whole. Also, 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* with a program, i.e., which annotation types exist and in which program contexts a particular annotation is allowed, depends on the given annotation language.

If P is a program and A is a set of annotations compatible with P, then we call the pair $P{+}A$.

Definition 1 (Annotation satisfaction). *We assume that there is a definition of when a program P satisfies a set $SPEC$ of annotations, denoted by $\models P{+}SPEC$.*

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.

The possible outcomes of running a verification system on some test case $P{+}(REQ \cup AUX)$, i.e., a verification problem consisting of a program P, a requirement specification REQ, and auxiliary annotations AUX, are:

proved: The system finds a proof, showing that P satisfies $REQ \cup AUX$.

not provable: The system is able to show that 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 counterexample or by showing the current proof state.

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

In order to evaluate the obtained results of a test run, we require for the case studies presented in Sect. 6 that the expected outcome of each test case is provided by the author of the test case.

4 Testing Different Properties

Testing a verification system can exhibit various kinds of failures related to different correctness properties. A verification system can be unsound, i.e., verify an incorrect program to be correct. There can be completeness failures, where the system fails to find a proof for the correctness of a program. There also can be performance failures, where the system's performance is too low either in terms of resource consumption or in terms of the amount and complexity of auxiliary input the user has to provide to be able to prove a program correct. And, of course, the system may crash. While abnormal termination of software is usually a serious concern, a bug that makes the verification system crash is in general less serious than a completeness failure because it cannot be confused with bugs in the program to be verified (the verification target).

How one can test for different kinds of failures, and thus, correctness properties, is discussed in the following subsections. Table 1 summarizes the different test results and the failures they indicate.[3]

Table 1. The different test results and the failures they indicate.

	Intended		
Observed	proved	not provable	timeout
proved	—	unsoundness	unsoundness or positive performance anomaly
not provable	incompleteness or performance failure w.r.t. required annotations	—	incompleteness or positive performance anomaly
timeout	incompleteness or performance failure w.r.t. required annotations or resources	unsoundness or performance failure w.r.t. resources	—

[3] The possibility of abnormal termination exists in all cases and is not included in the table.

4.1 Testing Soundness

The most important property of verification tools is soundness. This means that whenever the output for a verification problem P+$SPEC$ is "proved", then the program P indeed satisfies the specification $SPEC$.

To reveal a soundness bug, a test case must consist of a program P and a specification $SPEC$ such that P does *not* satisfy $SPEC$. The correct answer for such a verification problem is "not provable" or "timeout", while "provable" indicates a soundness failure.

Programs that satisfy their specification cannot reveal soundness bugs—at least not directly. The exception are cases where the expected answer is "timeout" but the system answers "proved". Such an anomaly—that needs to be investigated by the developer—can either stem from an unexpected good performance or from a short but incorrect proof (i.e., a soundness problem).

4.2 Testing Completeness

As all sound verification systems must be incomplete (Rice's theorem), sound and complete software verification tools cannot exist. Instead the notion of *relative completeness* is used, i.e., completeness in the sense that the system (respectively its calculus) would be complete if it had an oracle for the validity of formulas over arithmetics [10].

In practice, however, all of today's program verification systems are not even relatively complete. This is not only due to resource limitations. Verification systems presuppose auxiliary annotations or other user input. Auto-active verification tools do not attempt to generate all missing auxiliary annotations. Such an "annotation generator" would give auto-active verification systems the (theoretical) property of being relatively complete but would be useless in practice (although in theory it can be built). Thus, it is neither given nor expected that a program verification system is relatively complete. In practice, completeness of verification tool means that if the program is correct w.r.t. its *given* requirement specification REQ, then some auxiliary specification AUX or other required user input *exists* allowing to prove this [5].

Definition 2 (Annotation completeness). *A verification system S is* annotation complete *if for each program P and specification REQ with*

$$\models P{+}REQ \ ,$$

there is a set AUX of annotations such that

$$\vdash_S P{+}(REQ \cup AUX) \ ,$$

i.e., S finds a proof for $\models P{+}(REQ \cup AUX)$.

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. In that case, a proof may exist using a different (stronger) set AUX' of auxiliary annotations. That is, the system may or may not be annotation complete. The failed test only shows that the expectation that a proof can be found using the annotation set AUX does not hold. The situation is similar with an observed output "timeout". In that case, the system may just be slower than 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.

5 A New Coverage Criterion: 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 are tested that define the semantics of programs and specifications and make up an important part of the system.

To solve this problem, we define the notion of axiomatization coverage. It measures to which extent a test suite exercises the axioms 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. We distinguish two versions: (a) the percentage of axioms needed to successfully verify correct programs (completeness coverage), and (b) the percentage of axioms used in failed proof attempts for programs not satisfying their specification (soundness coverage).

An erroneous axiom may lead to unsoundness or incompleteness or both. The latter effect, where something incorrect can be derived and something correct cannot, is actually quite frequent. Because of that, completeness tests can reveal soundness bugs and vice versa. Nevertheless, one should use both kinds of tests (soundness and completeness) and, thus, both kinds of coverage.

For the remainder of this paper, especially in the two case studies, we concentrate on the completeness version of axiomatization coverage. If not explicitly stated otherwise, "coverage" will stand for "completeness coverage" in the following.

5.1 Completeness Coverage

For the completeness version of axiomatization coverage, we define an axiom 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 3. *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* $Th' \subsetneq Th$.

According to this definition, not all axioms used in a successful proof for a test case are covered by that test. Some axioms may be redundant, i.e., they (a) can be replaced by (a combination of) other axioms used in the same proof, or (b) do not contribute to the proof because they were applied in "dead ends" of the proof search. For case (b), we argue that it is unlikely that the test case makes a relevant statement about the correctness of the axiom (other than that the axiom does not lead to inconsistencies in this proof). For case (a), we expect that there are different use cases in which to apply one or the other set of axioms and thus there should be test cases able to cover one specific set.

If we consider particular verification systems with a fixed axiomatization, we can define an axiom to be *strongly* covered, if it is needed in all proofs of the test case that the verification system is able to find using the given axiomatization.

Definition 4. *A test case* $P{+}(REQ \cup AUX)$ strongly *covers the axioms in a set* Th *w.r.t. an axiomatization* Ax*, if* $Th \subseteq Th'$ *for all sets* $Th' \subseteq Ax$ *such that* $P{+}(REQ \cup AUX)$ *covers* Th'.

Our notion of completeness coverage is rather coarse in that it does not take the structure of the covered axioms (resp. the inference rules in case of the KeY system) into account. One obvious improvement would be to examine in the coverage analysis which part of an axiom is actually needed in a proof—e.g., in case of an implication $A_1 \vee \ldots \vee A_n \rightarrow B_1 \wedge \ldots \wedge B_m$, which A_k establishes the premiss and which of the B_k of the conclusion are actually needed further in the proof. Precise definitions of more fine-grained coverage metrics and their evaluation is part of future work.

Ideally, the coverage definitions would use logical entailment instead of inference. However, as we want to quantify axiom coverage in practice and verification tools are inherently incomplete, the coverage metric is based on the inference relation.

As a consequence, the axiom coverage of a test suite w.r.t. 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. This implies that when calculating axiom coverage, to get reproducible results, performance of the computer that the verification tool runs on has to be taken into account. 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.

In general, the minimal set of axioms covered by a given verification problem is not unique. So, the question arises of what to do if a test case covers several different axiom sets.

We chose to follow the more conservative approach to consider only one (non-deterministically chosen) axiom set to be covered by any given test case. A pragmatic reason for that choice is that it is very costly to compute all minimal axiom sets covered by a test case. But our conservative choice has another important advantage: If, for example, there is a logically redundant axiom A' that is an instance of some other axiom A, and there are verification problems that can be solved either using the more general axiom A or the more special A', then there are test cases that cover A and A' separately but not at the same time. We will count only one of A and A' to be covered by such a single test case. Now there are two situations to consider: (1) A' is included in the axiom set for good reason as it leads to better performance (shorter proofs) for a certain class of problems. Then there should be a test case in the test suite that can only be proved using A'. This test should have sufficiently low resource bounds so that it cannot be verified using A, and thus demonstrates the usefulness of A'. And there should be a different test case that does not fall in the special class to which A' applies and that can only be solved using A. Then, with the two test cases, both A and A' are covered. (2) A' is really redundant, i.e., it is not possible to construct a test case that can be verified with A' but not with A. Then A' is indeed akin to dead code (which also cannot be covered by test cases) and should be removed.

The way in which we compute the axioms covered by a test case ensures that the non-deterministic choice of the covered axioms is done in a useful way in case there is more than one possibility (see Sect. 6).

5.2 Soundness Coverage

For the soundness version of axiomatization coverage, the above definition of *needed axioms* based on minimal axiom sets is not useful. Instead, an axiom is *used* in a failed proof attempt, if it occurs in the proof search, i.e., the verification system actively used the axiom for proof construction. What "used" means depends on the particular verification system and its calculus.

6 Case Studies

To evaluate the usefulness of our notion of the completeness version of axiomatization coverage, we conducted two case studies. As verification systems to be tested, we chose the VCC tool, as well as the KeY system. For both tools, we chose to evaluate the test suite that is part of the corresponding distribution. In addition, for the KeY tool, we examined a third-party test suite.

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

To compute an approximation of the axiomatization coverage for a completeness test case $P{+}SPEC$, the procedure is as follows: in a first step, $P{+}SPEC$ 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 resp. time needed), information on which axioms are actually used in the proof are recorded as set T (e.g., by leveraging Z3's option to generate unsatisfiability cores in case of VCC[4], resp. parsing KeY's explicit proof object). This set T is a first approximation of the completeness coverage of the test case but has to be narrowed down in a subsequent step to yield the actual axiomatization coverage.

For this, 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 $P{+}SPEC$ 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 $T = \varnothing$.

In all these subsequent proof runs, resource constraints are set to twice the amount of resources needed for the first proof run recorded initially. This allows us to calculate axiom coverage in reasonable time and ensures comparability of coverage measures between computers of different processing power.

The resulting set of axioms C is only an approximation of the coverage of $P{+}SPEC$. For precise results the above procedure would have to be repeated with C as input as long as the result is different from the input. For practical reasons we compute axiomatization coverage using only one iteration for the following case studies.

Note that it currently takes several minutes to compute the 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 which could be discarded at once are needed.

6.1 Testing the Axiomatization of VCC

The Architecture and Workflow of VCC In the following we give a short overview of the verification workflow and give a description of the architecture of the VCC tool. For a thorough introduction to the VCC methodology, see [9].

[4] The coverage experiments in this paper have been produced by an older version of our framework without this feature—however, this only impacts performance of the framework.

Table 2. Coverage measures for the first experiment

earlier version of axiomatization			
	total	covered	percentage
axioms for C language features	212	84	40%
axioms for specification language features	166	102	61%
all axioms	378	186	49%
later version of axiomatization			
all axioms	384	139	36%

The VCC tool chain allows for modular verification of C programs using method contracts and invariants over data structures. Method contracts are specified by pre- and postconditions. These contracts and invariants are stored as annotations within the source code in a way that is transparent to the regular C compiler. The tool chain translates the annotated C code into first-order logic. Subsequently, the formulas are given to the SMT solver Z3 [17] together with the prelude (the axiomatization) capturing the semantics of C's built-in operators, etc. Z3 checks whether the verification conditions are entailed by the axiomatization. Entailment implies that the original program is correct w.r.t. its specification.

Axiomatization Coverage Results Using our testing framework, we automatically executed the completeness test cases contained in VCC's test suite and measured both the axiomatization and the code coverage achieved with these tests.

First Experiment. First, we used the test suite shipped with the binary package of VCC version 2.1.20731.0. It consists of 400 test cases, 202 of which are completeness tests. For comparison, we measured axiomatization coverage for two versions of VCC: version 2.1.20731.0, from which the test suite was taken, and version 2.1.20908.0, which is a version about six weeks further into development. The earlier version of the axiomatization contains 378 axioms out of which 186 were covered (49%). A classification of these axioms and the different degrees of coverage for different types of axioms is shown in Table 2.

The later version of the axiomatization contains 384 axioms, of which only 139 were covered (36%), i.e., axiomatization coverage decreased. Investigations revealed that the reason for this decrease is that axioms were modified, e.g., by removing requirements or by adding predicates. Therefore, the old test suite was less adequate to the newer version of VCC.

Second Experiment. In a second experiment, we used VCC version 2.1.30820.1 (one year after the VCC version used in the first experiment), and the accompanying test suite. The examined part of the new test suite that corresponds to the test suite used in the first experiment had been updated and now contains a total of 698 test cases, 417 of which are completeness tests (the rest are

soundness tests and tests checking for parser errors). The axiomatization now consists of 439 axioms. Of these 211 were covered by the completeness tests (48%), i.e., axiomatization coverage has increased again to the level of VCC version 2.1.20731.0, due to the updated test suite.

Further, this new version of the test suite contains an additional directory of test cases we did not consider here to be able to compare coverage results with the first experiment. We thus expect even better coverage when taking all tests into consideration.

For this second experiment, we additionally computed the code coverage[5] for the part of VCC that is related to the semantics of the programming and the specification language, i.e., the verification condition generator. Using the same test suite of 417 completeness tests, the resulting code coverage turned out to be 70%. This is an interesting result as it shows that axiomatization coverage (48%) can be quite a bit lower than code coverage (70%). And it is evidence that axiomatization coverage is independent of code coverage. Therefore, axiomatization coverage should indeed be considered in addition to code coverage to judge the quality of a test suite.

Other Insights. Additional investigations showed a further difference between code coverage and axiomatization coverage: The code coverage of *individual* test cases is higher than their axiomatization coverage. Axiomatization coverage can be as low as 1% for some tests, while code coverage is never less than 25%. That is, there is a certain amount of "core code" exercised by all tests, while there are no "core axioms" used by all tests (this may, of course, be different for other verification systems).

Also, the coverage for other elements of the prelude besides axioms, e.g., type declarations, turned out to be much higher than the axiomatization coverage. It was 81% for the first experiment and 72% for the second experiment. It decreased as the new version of the prelude contains declarations related to information-flow analysis, for which no tests have been added to the test suite (yet).

Errors Found in VCC The main goal of our case study was not to find bugs in VCC but to evaluate the quality of the tests. And it was to be expected that no errors could be detected using VCC's own test suite as those tests, of course, had already been used by the VCC developers for testing.

Using tests from other sources (which had a rather low coverage), we found (only) one completeness failure and no soundness failure. The completeness bug in the axiomatization, which has been fixed in the current version of VCC, related to the ownership model. In some situations it was not possible to prove that some part of the state had not changed after an assignment to a memory location outside that part of the state.

[5] Code coverage was computed using the code coverage feature available in Microsoft Visual Studio 2010 Premium.

6.2 Testing the Calculus Rules of KeY

The KeY System As second target for our case studies we have chosen the KeY[6] tool [6], a verification system for sequential Java Card programs. Similar to VCC, programs can be specified using annotations in the source code. 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 into a sequent in Java Dynamic Logic, a multi-modal predicate logic. Validity of this sequent implies that the program is correct w.r.t. 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 plays a similar role as the prelude in case of VCC, as it captures the semantics of Java, built-in abstract data types like sequences etc. In comparison to the prelude of VCC, however, KeY also contains taclets that deal with first order logic formulas, whereas first-order reasoning in VCC is handled by the SMT component Z3.

The development version of KeY as of August 2012, contains about 1500 taclets. However, not all of them are available at a time when performing a proof, as some of the taclets 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).

Automatic proof search is combined with interactive steps of the user, in case a proof is not found automatically. For our purposes, the interactive part of KeY is irrelevant, 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 also similar to those in VCC: either the generated Java Card DL 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.

Axiomatization Coverage Results Using a modified version of our testing framework, we automatically executed the test cases contained in KeY's test suite, as well as parts of a custom Java compiler test suite and measured the taclet completeness coverage.

The procedure used here is similar to reducing VCC's axiomatization. However, the set of taclets to start from is directly taken from the explicit proof object that KeY maintains and that is used to save and load (partial) proofs.

[6] See `http://www.key-project.org`

This object stores sufficient information about each taclet application in the sequent calculus proof, such that the proof can be reconstructed by KeY without performing proof search. These taclet applications recorded normally contain a lot more taclets than are actually relevant for finding the proof. Thus, we reduce the set of taclets used in the proof construction one by one in a second step in the same manner as for VCC.

Third Experiment. In this experiment we used the development version of the KeY tool[7] as of August 16, 2012. As part of the KeY source distribution, a test suite is provided containing 335 test cases of which 327 are completeness and 8 are soundness tests. 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.

From the 327 completeness tests of the KeY test suite, we computed the taclet completeness coverage of 319 test cases testing verification of functional properties—another eight test cases are concerned with the verification of information-flow properties and were omitted due to resource constraints. The test runs were distributed on multiple computers using Amazon's Elastic Compute Cloud service, taking approx. 135 EC2 Compute Unit[8] hours to complete in total.

The overall taclet coverage we computed for the 319 completeness tests was 38% (with 585 out of 1527 taclets covered). Figure 2 shows a histogram of the number of test cases each taclet is covered by. The overall coverage seems to be comparable to the coverage results gained from the VCC test suite. However, in contrast to VCC, KeY also performs first-order reasoning with the help of taclets instead of using an SMT solver—for better comparison with VCC, all such taclets would have to be excluded from coverage computation.

While some features of Java or JML are not covered at all by the current test suite of KeY, for other taclets, low coverage might result from:

(a) the fact that some taclets have been introduced just recently, for a particular use case (e.g., taclets enabling automatic induction proofs in certain cases) and no test cases have been written yet,
(b) redundant taclets that are used to shorten the proof (e.g., the KeY taclet replacing "$F \rightarrow \text{true}$" by "true" for any FOL formula F is made redundant by the taclet replacing implication by its definition, together with the taclet rewriting the result "$\neg F \vee \text{true}$" to "true". This is one example for a taclet used but *not* covered in a proof.) and
(c) obsolete taclets that are still contained in the rule base.

Measures to handle cases (b) and (c) have already been discussed in Sect. 5.1. For case (a), a review and testing process has to ensure that the axiomatization coverage for newly introduced taclets is increased by writing specific test cases.

[7] Available at `http://i12www.ira.uka.de/~bubel/nightly/`

[8] According to Amazon's EC2 documentation, "One EC2 Compute Unit provides the equivalent CPU capacity of a 1.0-1.2 GHz 2007 Opteron or 2007 Xeon processor."

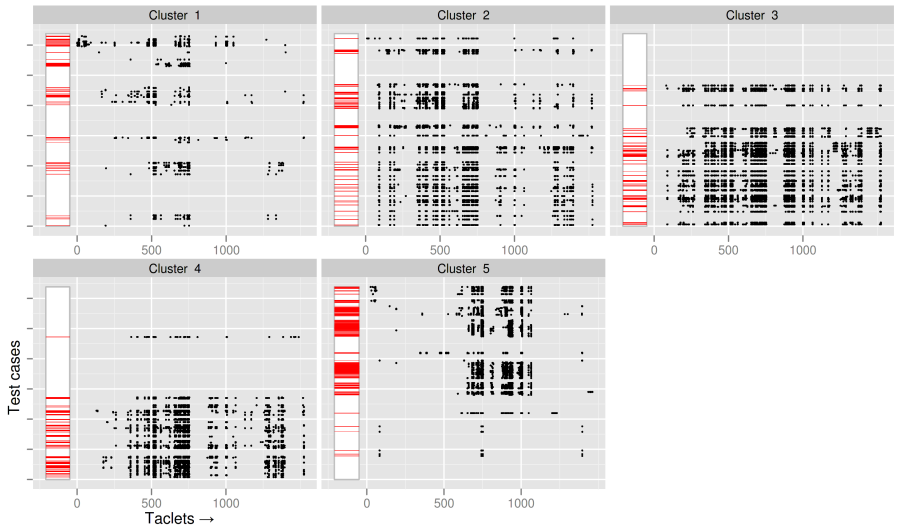

Fig. 1. Groups of similar KeY test cases in terms of taclet coverage. Each point in the diagram indicates a taclet covered by a test case. Clusters are computed by hierarchical clustering using Ward's method and Jaccard distance as measure for dissimilarity between taclet coverage of two test cases. The x-axis shows taclets sorted by group, the y-axis shows test cases sorted by directory. If a test case is contained in a cluster, this is indicated by a red line in the box on the left.

In order to perform a more detailed analysis, coverage results were examined by groups of taclets respectively test cases. For grouping taclets, we used the already existing structure given by the organization of taclets in different files in the KeY distribution—similar taclets (e.g., taclets handling Java language features or taclets for propositional logic) are contained together in one file.

Histograms of the number of test cases a taclet is covered by, split by taclet group, already allowed us to compare the quality of the test suite w.r.t. the different groups: not surprisingly, taclets handling propositional logic or Java heap properties are covered quite often (in both groups over 60% of taclets are covered at least in one test case). On the other end of the spectrum, 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 us to focus the effort of writing new test cases on constructing specific tests for rarely covered taclet groups.

In order to group similar test cases together to identify commonalities, we used the R environment for statistical computing [19] to cluster test cases. The result of this clustering is shown in Fig. 1.

Two of these test case clusters are notably representative for different types of test cases: while the tests in Cluster 5 mostly encompass a small set of related taclets, tests in Cluster 3 span almost the entire taclet base of KeY. Indeed,

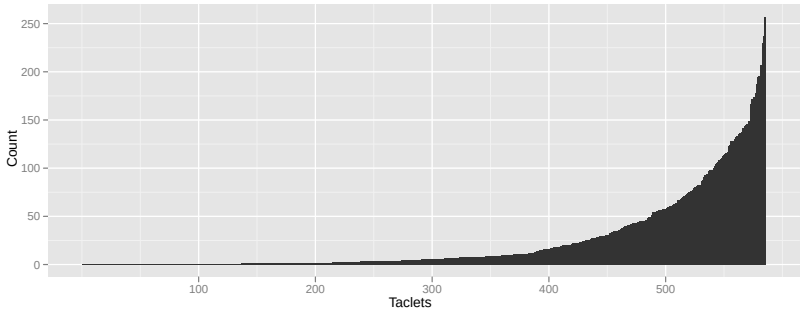

Fig. 2. Taclet coverage counts (y-axis shows number of test cases a taclet is covered by; x-axis shows taclets at least covered by one test case, sorted by y values).

test cases belonging to Cluster 5 have been written as specific tests for single features of KeY—in this case, integer arithmetic and handling strings (which need similar taclets because the functions retrieving characters from a string or getting substrings use integers; the small group of taclets in the upper left corner of Cluster 5 are the taclets handling strings). Test cases corresponding to Cluster 3 are mostly taken from verification competitions dealing with data structures on the heap like linked lists.

We believe that a good test suite for a verification system needs both of these types of test cases *for each taclet*. The need for broader test cases, covering several combinations of taclets, is supported by studies (e.g. [14]) 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 in the proof a taclet will presumably be applied in. As a measure for this, we define the *selectivity* of a test case as the number of taclets covered by the test.

The current state of the KeY test suite w.r.t. this selectivity criterion is shown in Fig. 3. For each taclet, the average selectivity of all test cases covering this taclet is shown, together with the population standard deviation from the average. The leftmost taclets in this diagram are good candidates for which additional test cases might be needed, as they are only covered by specialized test cases. Also taclets with a high selectivity average of the corresponding test cases but low deviation indicate need for improvements, as only broad test cases cover the taclets.

Fourth Experiment. The last experiment we conducted used parts of the test suite of the Java2Jinja[9] compiler [15] in order to increase taclet coverage compared to KeY's own test suite.

The part of Java2Jinja's test suite we considered here is hand-written and consists mostly of small Java programs testing few Java features at a time, often

[9] See `http://pp.info.uni-karlsruhe.de/projects/quis-custodiet/Java2Jinja`

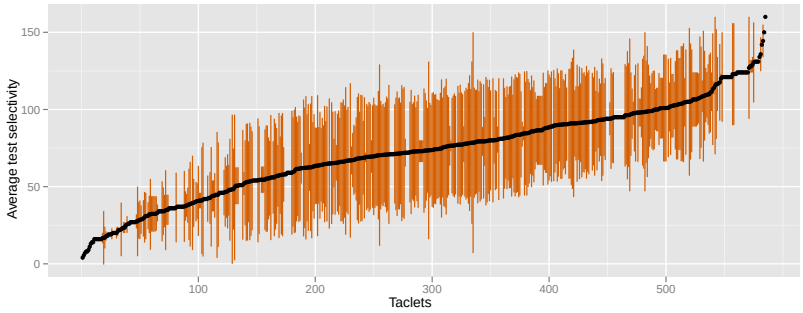

Fig. 3. Average test case selectivity by taclet. Black points: average selectivity (see Sec. 6.2) of all test cases covering a taclet. Population standard deviation of this value from the average is shown as red bars.

dealing with corner cases of the Java semantics. All tests are run by providing concrete, fixed input parameters to the test's main method. The result of the execution of a test is an output to the console—the expected outcome of the test case is given as annotation in the test file. This annotation can easily be converted into a postcondition suitable to be proven with KeY.

From a total of 43 annotated test cases, 21 were applicable to KeY—most of the other tests included features not yet supported by KeY (e.g., Java generics). Of the 21 suitable tests, 12 were directly provable (two thereof using user interaction), the rest of the proofs exceeded allocated time or memory resources.

One result of the test runs was that the Java2Jinja test suite covered 195 of 1527 taclets, corresponding to 13% taclet coverage. Even this small set of additional test cases covered nine taclets that were not covered by KeY's own test suite.

Errors Found in KeY As already argued in Sect. 6.1, we could not expect to find soundness bugs using KeY's own test suite. Therefore, we performed the last experiment using Java2Jinja's test cases—resulting in two bugs found in the rule base of KeY.

The first bug we discovered is related to implicit conversion of integer literals to strings in Java: in case a negative integer literal is converted to its string representation, the minus sign is placed at the last instead of first position in the resulting string. The Java2Jinja test case that revealed this bug has not been written to test exactly this conversion feature but rather the correct handling of precedences in integer arithmetic. To write an additional test case for this exact feature that increases axiomatization coverage is easy, as exactly one taclet is involved in the bug and the condition under which this taclet is applicable is clearly visible to the user.

The second bug found in the taclets of KeY deals with the creation of inner member classes, using a qualified class instance creation expression. How-

ever, the corresponding taclet in KeY that symbolically executes this instance creation expression does not check whether the qualifying expression is null and always creates an object of the inner class without throwing the required `NullPointerException` (see the Java language specification, Sect. 15.9.4).

While both of these bugs in the taclets allow the user to prove properties about certain programs that do not hold, they do not lead to unsoundness in the general case: both bugs are only triggered if the Java program contains the corresponding features (i.e., qualified class instance creation or conversion of negative integer literals to string). In addition, the bugs only influence the correctness of the proof if the property to be verified relies on those features.

In both cases, the KeY test suite did not cover the relevant taclets at all, and the increase in the axiomatization coverage by the two relevant Java2Jinja tests indeed allowed us to reveal faulty taclets. This shows that axiomatization coverage is a useful metric to get a first hint to parts of the axiomatization that may be target for further inspection and validation measures.

7 Related Work

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 [12], and the Mobius project [4], all aimed at the development of fully verified verification systems. Calculus rules of the KeY verification system [6] for Java were verified using the Maude tool [2].

Verifying a verification system is useful, but it cannot fully replace testing; this is further discussed in [7]. The authors claim that verifying a verification tool involves a huge effort that is, to some extent, better directed to improve other qualities of verification systems relevant in practice (e.g., efficiency of the tool). Also some sort of cross-validation between tools is needed, amongst other reasons, as there is no single, authoritative formal language specification for Java. For this, one option mentioned explicitly in [7] is to use "cross-validation with test programs written by different people."

For the theorem provers and SMT solvers that are components of verification systems, there are established problem libraries that can be used as test suites, such as the SMT-LIB library [3]. Alternatively, the results of SMT solvers can be validated using proof checkers. For example, Z3 proofs can be checked using Isabelle [8]. Another example is the Formally Verified Proof Checker that was implemented in ML and formally verified using HOL88 [20].

An interesting application of conformance testing is the official validation test suite for FIPS C (a dialect of C) [13]. To determine how well this test suite covers all features of the C language, a reference implementation of a C compiler was built such that the implementation modules of the compiler could be associated to parts of the C standard. This allowed to relate code coverage of the reference compiler to coverage of the language standard when compiling the programs of the test suite.

In the case studies presented in this paper, the test cases used for coverage analysis were written by hand—coming up with meaningful test cases in this way is a time consuming process and complementing alternatives are worthwhile.

One option to obtain test cases is to use randomly generated programs with known behavior as a basis for comparing the outcome of the tool under test with the expected behavior of the generated program [11]. In order to test parts of Frama-C, the tool Csmith [21] was used to generate random C programs. These C programs output a checksum of their global variables, allowing to compare execution of the program compiled with a reference compiler to analysis results obtained with Frama-C.

Another approach to identify erroneous axioms in an axiomatization is to use model-based testing [1]. In order to test a first-order logic axiom, the user provides an interpretation of the functions and predicates used in the axiom by giving for each a Haskell implementation. The axiom to be tested is then translated into an executable Haskell function (using the user-provided interpretation of functions and predicates). This function is then tested to be true for a set of generated test inputs with the help of a standard Haskell testing framework.

8 Conclusions and Future Work

In this paper, we introduced axiomatization coverage as a new coverage criterion in testing verification systems. We conducted case studies at two verification systems to evaluate the completeness version of the axiomatization coverage of test suites supplied with the corresponding tools—in both cases (not surprisingly) showing a rather low axiomatization coverage.

Already this coarse coverage criterion can be used as a first measure to judge and improve the quality of these existing test suites. Further coverage statistics, like the test case selectivity, may be used additionally to identify axioms that are underrepresented in the test suite. Also clustering test cases similar in their axiomatization coverage may hint at which kind of additional test cases are still missing (e.g., large and complex programs using many program language and specification language features at once; or rather small, specific test cases covering few and similar axioms).

To evaluate the usefulness of our completeness version of axiomatization coverage, in a first step, we tried to increase coverage by additional test cases to uncover erroneous axioms. For this, we used parts of the Java2Jinja test suite to test the KeY verification tool, which revealed two bugs in KeY's rule base.

For the future, we plan to conduct additional case studies to further evaluate our coverage criteria and, in particular, to investigate which kind of tests uncover what kind and what number of errors. Additional test suites not written by tool developers themselves are required though to get useful statistics about bug occurrences.

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 is planned to be conducted where we systematically write specific

test cases aimed to increase the axiomatization coverage for relevant axioms of the existing test suites. If our assumption that axiomatization coverage is a useful measure is right, we should be able to find further bugs with these tests.

Further, we plan to use combinatorial testing, where combinations of language features and axioms are used in test cases, as well as more fine-grained axiomatization coverage criteria, in contrast to the notion of entire axioms as smallest coverage unit presented in this paper.

Acknowledgements. We thank Andreas Lochbihler, Jonas Thedering and Antonio Zea for providing the Java2Jinja test cases.

References

1. Ahn, K.Y., Denney, E.: Testing first-order logic axioms in program verification. In: Fraser, G., Gargantini, A. (eds.) TAP. Lecture Notes in Computer Science, vol. 6143, pp. 22–37. Springer (2010)
2. Ahrendt, W., Roth, A., Sasse, R.: Automatic validation of transformation rules for Java verification against a rewriting semantics. In: LPAR'05. vol. 3835, pp. 412–426. Springer (2005)
3. Barrett, C., Ranise, S., Stump, A., Tinelli, C.: The satisfiability modulo theories library (SMT-LIB). At `http://www.smt-lib.org/`
4. Barthe, G., Beringer, L., Crégut, P., Grégoire, B., Hofmann, M., Müller, P., Poll, E., Puebla, G., Stark, I., Vétillard, E.: MOBIUS: Mobility, Ubiquity, Security, LNCS, vol. 4661. Springer (2006), `http://dx.doi.org/10.1007/978-3-540-75336-0_2`
5. Beckert, B., Bormer, T., Klebanov, V.: Improving the usability of specification languages and methods for annotation-based verification. In: Aichernig, B., de Boer, F.S., Bonsangue, M. (eds.) FMCO 2010. State-of-the-Art Survey. LNCS, vol. 6957. Springer (2011)
6. Beckert, B., Hähnle, R., Schmitt, P.H. (eds.): Verification of Object-Oriented Software: The KeY Approach, LNCS, vol. 4334. Springer-Verlag (2007)
7. Beckert, B., Klebanov, V.: Must program verification systems and calculi be verified? In: 3rd International Verification Workshop (VERIFY), Workshop at Federated Logic Conferences (FLoC). pp. 34–41 (2006)
8. Böhme, S.: Proof reconstruction for Z3 in Isabelle/HOL. In: 7th International Workshop on Satisfiability Modulo Theories (SMT '09) (2009)
9. 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: TPHOLs'09. LNCS, vol. 5674, pp. 23–42. Springer (2009)
10. Cook, S.A.: Soundness and completeness of an axiom system for program verification. SIAM Journal of Computing 7(1), 70–90 (1978)
11. Cuoq, P., Monate, B., Pacalet, A., Prevosto, V., Regehr, J., Yakobowski, B., Yang, X.: Testing static analyzers with randomly generated programs. In: Proceedings of the 4th international conference on NASA Formal Methods. pp. 120–125. NFM'12, Springer-Verlag, Berlin, Heidelberg (2012), `http://dx.doi.org/10.1007/978-3-642-28891-3_12`
12. Jacobs, B., Poll, E.: Java program verification at Nijmegen: Developments and perspective. LNCS 3233, 134–153 (2004)
13. Jones, D.: Who guards the guardians? (Feb 1997), `http://www.knosof.co.uk/whoguard.html`

14. Kuhn, D.R., Wallace, D.R., Gallo, A.M.: Software fault interactions and implications for software testing. IEEE Transactions on Software Engineering 30(6), 418–421 (2004)
15. Lochbihler, A.: A Machine-Checked, Type-Safe Model of Java Concurrency: Language, Virtual Machine, Memory Model, and Verified Compiler. Ph.D. thesis, Karlsruher Institut für Technologie, Fakultät für Informatik (Jul 2012), http://digbib.ubka.uni-karlsruhe.de/volltexte/1000028867
16. Marché, C., Moy, Y.: The Jessie plugin for Deductive Verification in Frama-C—Tutorial and Reference Manual (2013), at http://krakatoa.lri.fr/jessie.pdf
17. de Moura, L., Bjørner, N.: Z3: An efficient SMT solver. In: TACAS'08. pp. 337–340. LNCS 4963, Springer (2008)
18. von Oheimb, D.: Hoare logic for Java in Isabelle/HOL. Concurrency and Computation Practice and Experience 13(13), 1173–1214 (2001)
19. R Core Team: R: A Language and Environment for Statistical Computing. R Foundation for Statistical Computing, Vienna, Austria (2012), http://www.R-project.org
20. von Wright, J.: The formal verification of a proof checker (1994), SRI internal report
21. Yang, X., Chen, Y., Eide, E., Regehr, J.: Finding and understanding bugs in C compilers. In: Hall, M.W., Padua, D.A. (eds.) PLDI. pp. 283–294. ACM (2011)