Towards Testing a Verifying Compiler*

Thorsten Bormer[1] and Markus Wagner[2]

[1] Institute for Theoretical Computer Science,
Karlsruhe Institute of Technology, Germany
bormer@kit.edu
[2] Department 1: Algorithms and Complexity,
Max Planck Institute for Informatics, Germany
mwagner@mpi-inf.mpg.de

Abstract. In this paper, we present our approach on testing a particular verification system that is industrially used to generate mathematical proofs of the correctness of C programs.
Normally, the tools used in such a verification process are seldomly verified nor thoroughly tested, and their correctness is taken for granted. Our approach to obtain assurance in such tools does not rely on the knowledge of their internal details and enables regular users of these tools to write test cases for them. Those tests are then assessed using our domain-specific *axiomatization coverage* that measures the impact of the axiomatization, which is an integral component of the verification process. Furthermore, we explore several sources of test cases, as the risk of constructing buggy test cases is high due to the input domain's complexity.

Keywords: Software validation, black-box testing, large software system

1 Introduction

Employing formal methods in the software development process is a viable, if sometimes deemed as costly, way to enhance the quality of the resulting product. One of the possibilities to use formal methods is in the verification phase of software development, supplementing the testing effort by formal software verification. Through formal verification, one obtains a mathematical proof that the program is correct with respect to its given specification.

The benefit of such a correctness proof is most apparent with safety-critical software. Additionally, in a certification process with high requirements on software quality and associated evidence of former (for example, in CC EAL 7+ or the upcoming DO178-C standard), these correctness proofs would be a valuable resource. To use the correctness proof in some certification process, the tool that was employed to generate the proof has itself to be validated in some cases.

Unfortunately, this is often not the case with existing software verification tools. As a user of these tools, internal details or even the source code of the tools are often not

* 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.

available to be able to verify that the tool is working correctly—even if access to the source code is possible, lack of resources impede the application of formal methods to the tools themselves, due to the complexity of the tools.

In this paper we propose a different approach to obtain assurance in the correctness of the software verification tools, namely by testing. Our method does not rely on the knowledge of internal details of the verification tool and enables regular users of these tools to write test cases for them.

This paper is structured as follows. First, our subject under test, the verification system VCC, is presented in Section 2 with details on the toolchain and verification methodology. Then, in Section 3, the theoretical aspects of testing verifying compilers are investigated. In the subsequent sections, the theoretical results are applied to the subject under test. For this, a suitable technique for testing VCC is chosen in Section 4, where we define the domain-specific *axiomatization coverage* as our test metric, and explore several sources for test cases. Finally, the results of the testing process are presented and assessed in Section 5.

2 A Typical Verifying Compiler

For the rest of this paper, we have chosen the VCC tool [9,10], developed by Microsoft Research, as the verification system to be tested. This tool is developed in the context of the Verisoft XT project where it is successfully used within two subprojects to verify functional properties of system software.

VCC is chosen here as a particular instance of formal software verification tools that follow the "verifying compiler" paradigm. While the tool description in the following is concerned with the details of VCC, the design and architecture of VCC is similar to other tools in this area, for example Caduceus or Krakatoa, so the testing methodology of our paper is not restricted to this particular setup. VCC is being developed as an industrial-oriented verification environment for low-level concurrent system code written in C. It takes a program that is annotated with function contracts, state assertions, and type invariants, and attempts to prove the correctness of these annotations.

In the following we will give a short overview on the verification workflow and give a description of the internal architecture of the VCC tool. The particular elements of the VCC specification language and methodology are not contained in this section, but described together with the examples presented later on, as far as needed. For a thorough introduction into the VCC methodology, see [9].

2.1 The VCC Workflow

To verify whether a program fulfills certain functional properties using VCC, the intended properties are first formulated with the help of the VCC specification language, such as method contracts or invariants on data types. This specification language is similar to those found in ESC/Java2 [11], Spec#, and HAVOC [8]. As in all these systems, the program's specification is stored as inline source code annotations. These annotations are invisible to a normal C compiler (making use of the C preprocessor features) but are analyzed by VCC within the verification process.

Invoking VCC on an annotated C source file has one of the following outcomes: (a) VCC reports that the program fulfills its specification as given by the annotations or (b) VCC could not prove that the program meets the specification. The latter case may have several reasons (for example, not enough system resources for the prover, a bug in the software or specification)—for each of the error sources, there are appropriate tools in the VCC package to inspect and debug these errors.

2.2 Architecture of the VCC-Toolchain

To prove that a program meets its specification, the VCC tool internally makes use of a toolchain of three tools: from the annotated C code, with the help of VCC's compiler, a representation in an intermediate, imperative programming language with embedded specification constructs (called *BoogiePL* [17]) is generated. This BoogiePL representation is then further processed by the Boogie tool into proof obligations. These are then proven or refuted by the Z3 theorem prover—leading to either the statement that the original program meets its specification, or, if the proof obligations are refuted, to a counterexample.

In the following, we will give a short overview on each of these steps and components in the toolchain.

VCC's compiler The VCC compiler is build by using the Common Compiler Infrastructure (CCI)[3]. Annotated C programs are read and turned into CCI's internal representation to perform typical tasks of a regular C compiler, such as name resolution, and type and error checks. Next, the fully resolved input program is subject to several transformations: (1) simplifying the source, (2) adding proof obligations that result from the methodology, and (3) finally generating Boogie code.

Boogie When a C program is analyzed and found to be valid, it is translated into a Boogie program that encodes the input program according to the employed formalization of C. Boogie is an intermediate verification language and a verification system that acts as a layer on which program verifiers for other languages can be built upon. It is used by a number of software verification tools including Spec# and Havoc.

Before the Boogie program is fed to the Boogie program verifier, which translates it into a sequence of verification conditions, the *prelude* is added, which is an axiomatization of the C intrinsic memory model, object ownership, type state and arithmetic operations. Then, the verification conditions are passed to an automated theorem prover to be proven or refuted.

Z3 Z3 [12] is a first-order theorem prover that checks whether a set of formulae is satisfiable in the built-in theories. Those cover, for example, the equality over free function and predicate symbols, real and integer arithmetic, and bit-vectors.

3 Validation of Verification Environments

3.1 Software Validation

To check whether a software system meets its specification and fulfills its intended purpose, a plethora of techniques (for example, deductive verification, static analysis,

[3] Microsoft Research: CCI. 3 May 2010 `http://ccimetadata.codeplex.com/`

and white-/black-box testing) can be applied. In this work, we have chosen to use black-box testing as a cost-effective procedure to establish assurance that our target, VCC, works correctly.

In general, functional conformance testing is classified as a black-box approach when an external tester can only observe the outputs generated by the implementation upon the receipt of inputs, without any information about the internal design of an implementation. Conformance is the relation between a specification and an implementation, and the relation is valid if the implementation does not present behaviors that are not allowed by the specification. If the implementation is given as a black box, only its observable behavior would be able to be tested against the required behaviors by the specification.

Towards black-box testing of verification systems, we considered the following approaches to be applicable. *Error guessing* is an ad-hoc approach mostly based on experience. *Equivalence Partitioning* can be applied when the domain of each input parameter of a function is structured into equivalence classes. *Boundary Value Analysis* assumes that errors tend to occur near extreme values because typical programming errors—for example, wrong termination conditions for loops—are often related to these boundaries. *Model-driven testing* [2] was not considered applicable because of the very costly process of constructing a model for large systems. This is the case for verification systems, where the input and output data is tightly coupled to the behavior specifications of the verification system.

3.2 Validation Techniques for Verification Systems

When it comes to identifying the components of the verification system that are to be validated, we identified two major obstacles. The first one was the *complexity of the toolchain*. Verification systems are usually large software systems: they are composed of complex parsers for the input languages, mechanisms to rewrite the input into proof obligations, and possibly problem solvers and other tools. The second obstacle was the *complexity of the supported languages*. Automatic verification systems usually support a programming language that is annotated with elements from a specification language. This results in the complex interaction of elements from both languages.

The complexity of the toolchain can be countered by testing the components individually, if possible. A structured divide-and-conquer approach towards the interaction of language elements cannot be defined as straightforward. This is due to the rather unstructured input domain of a verification system; each test is not simply a combination of some values for a function to be tested, but an entire C program including annotations. Some structures within the domain can be achieved by defining some orders over the individual language elements, or by aggregating elements, such as “arithmetic operators” and “memory model specific operators”, to domains. Based on these domains, test cases can be created systematically by using the combinatorial testing approach. Once a thorough test is performed, combinatorial testing offers an easy and intuitive evaluation of the testing process itself: based on the structured approach, the coverage on n-wise coverage combinations can be computed, and these numbers can help to build trust in the tool.

Related Work In principle, instead of using our testing approach, parts of the verification tools available could be formally verified by using either the verification tools themselves or others. There have been several efforts to develop completely certified program verifiers, e.g., in the Bali project [21], the LOOP project [15], and in the Mobius project [4]. Several times, tricky verification examples were proposed to test verification tools ([14]), and furthermore, components of Java verification tools were verified ([1]). One example of such a soundness proof conducted is the verification of the rewrite rules of Caveat's[4] integrated theorem prover by using PVS[5]. In addition, though, also all combinations of C's syntactic constructs were tested. Due to our limited resources, a comparable approach could not be realized in our scenario.

In their discussion on whether verification systems and calculi have to be verified in general, Beckert and Klebanov [6] argued that in practice, a more powerful and sufficiently correct system may be used in favor of a less powerful but correct system. Although they considered the verification of the tools or its components as important, they advised the developers of verification systems to test more frequently.

A less labor-intensive method than (cross-)verifying parts of the verification systems would be conducting (cross-)validation of the components by comparative testing. However, the question is whether such a comparable (verification) system exists. For the part of programming language, regular compilers may be used as sources for comparative statements on the parsability of source code, but finding several verification systems with similar features that are able to produce comparable outputs from the same source code is a problem.

Regarding the annotation languages that are commonly used, we observed the relative similarity between languages such as Java Modeling Language [7], the ANSI/ISO C Specification language (ACSL)[6], and Microsoft's variants. With possible convergence of specification languages in the future, we expect the number of comparable verifications systems to increase. Thus the creation of verification-specific test cases will become more desirable because of their increased reusability.

Concentrating on the theorem provers that are used in the last stage of VCC to discharge verification conditions, different approaches are capable of building trust in them. For example, cross-validation can be used, based on established problem libraries such as the well-known TPTP library[7]. Alternatively, the results can be validated by using proof checkers. One example of such a system is the Formally Verified Proof Checker that was implemented in ML and even formally verified by using HOL88 [22].

An interesting application of conformance testing is the official validation test suite for FIPS C (a dialect of C).[8] In order to determine the coverage on the language standard of a test suite, the language standard itself was implemented in a so-called *model*

[4] CEA-LIST: The Caveat Tool. 3 May 2010 `http://www-list.cea.fr/labos/gb/LSL/caveat/index.html`

[5] SRI International: PVS. 3 May 2010 `http://www.csl.sri.com/projects/pvs/`

[6] CEA-LIST/INRIA-Sacley: ACSL. 3 May 2010 `http://frama-c.com/acsl.html`

[7] Geoff Sutcliffe, Christian Suttner: The TPTP Problem Library for Automated Theorem Proving. 3 May 2010 `http://www.cs.miami.edu/~tptp/`

[8] Derek Jones: Who Guards the Guardians? 3 May 2010 `http://www.knosof.co.uk/whoguard.html`

implementation, i.e., an actual compiler based on the language description. Statements of the model implementation were then mapped back to the standard, allowing the authors to show that all of the requirements in the standard were implemented. With this approach, statement coverage w.r.t. this model implementation thus relates to coverage of the language standard elements. In the end, a statement coverage of 84% of the model implementation was achieved by a comprehensive test suite, demonstrating that the test suite checks a substantial portion of the C programming language.

4 Testing of VCC

To effectively apply software testing to VCC, we have to first identify the important quality attributes of the subject under test. These attributes can then be used to derive or select useful metrics in order to assess the quality of testing. Then, we discuss several possible sources for test cases in order to apply the concept of cross-validation to verification environment testing. It has to be noted that our approach is only weakly related to "regular" compiler testing, as our focus is not on the parsing capabilities and automatic error corrections, but on VCC's design goal, i.e., the ability to fully automatically prove a program's correctness.

4.1 Test Objective

In the following we concentrate on the soundness of verification systems. The discussion of Beckert et al. [5] on the completeness of verifying compilers is related to this definition, in which they distinguished between different types of annotations. For example, it can be the case that a program is correct with respect to its requirement specification, but the toolchain is unable to prove it. The reason for this is the missing *auxiliary* annotations that would have guided the theorem prover to the correctness proof. In the following, the term *annotations* is used to cover all the requirement specification annotations and the auxiliary annotations of a program.

From VCC's point of view, there is no way of differentiating a test case that is supposed to succeed from one that is supposed to fail. It is important to remember that we are not in the situation of verifying annotated programs, but of observing VCC's verification attempts on annotated programs. This leaves us with the following classification of test cases: (1) *successful* cases, where the outcome of a verification attempt equals the expected outcome, and (2) *failing* cases, where the outcome of a verification attempt does not equal the expected outcome. Hence, a test case for a verification system consists of an annotated program and some expected output. The verification system itself is not part of the test case; components such as the axiomatization remain unchanged for the testing process.

4.2 Testing with Respect to the Axiomatization

As already mentioned in Section 2.2, VCC's prelude contains an axiomatization of C written in Boogie. And within the multi-stage verification process, it has a significant impact on the outcome. Furthermore, the prelude is accessible to a human analyst, both on the code level and on the level of understanding the effects of the prelude's elements.

Based on the importance and its accessibility, we chose to test VCC with respect to the prelude, that is, to observe the impact of the prelude on the verification process. Alternatives will be discussed in Section 4.3.

The prelude itself is a Boogie program. In general, a Boogie *program* consists of a *theory* that is used to encode the semantics of the source language, and an *imperative part* [3]. A theory is composed of type declarations (*keyword: type*), symbol declarations (*const*, *function*), and axioms (*axiom*). The imperative part of a Boogie program consists of global variable declarations (*var*), procedure headers (*procedures*), and procedure implementations (*implementations*). The size of the prelude in our case is about 2900 lines of code—for easier maintenance and improved legibility the prelude is further structured into sections concerning different language- and specification features. Later on, we will modify the structure of the prelude.

4.3 Coverage Measurement

In the following, we describe our approach chosen to determine the impact of the axiomatization used. The idea is to determine the subset of elements of the original prelude that is used by the test case. It is checked whether or not an element of the prelude is needed by comparing the original output of VCC with the result when the selected element is left away. If the element can be left away, it is discarded for the given test case and the process is iterated until no more elements can be left away. Note that, in general, the minimal set of prelude elements for a given program is not uniquely defined. Depending on the generated proof obligations, different sets of prelude elements may be needed[9] and thus the selection strategy when reducing the prelude matters.

Based on our approach, the straightforward definition of axiomatization coverage follows:

Definition 1. *Given a verification environment v, its specification s , the complete axiomatization consisting of m elements, a test case t, and a corresponding minimized axiomatization a_{vst} with n_{vst} elements. Then the* axiomatization coverage *is defined as $Cov(v, s, t) = n_{vst}/m$. For a set of test cases $T = t_1, \ldots, t_n$ and corresponding minimized axiomatizations $a_1, \ldots, a_n$, the coverage is defined as $Cov(v, s, T) = n_a/m$ where $\bigcup_n a_i$ has n_a distinct elements.*

Tests that need rather many elements of the axiomatization can be regarded as *strong tests*; on the other hand, tests that requires only a rather small number of elements can also be regarded as strong because the verification task itself may be very complex. However, the latter kind of test strength addresses the problem of "the difficulty of verification", rather than "the interaction of the prelude's elements", in which we are actually interested.

Discussion and Alternatives In his work, Littlefair [19] examined the relation between the consideration of quality in software engineering and software metrics. Based

[9] e.g., consider the proof obligation $a \vee b$: either all elements needed to prove a are needed or all relevant elements for b

on his observations and on Weyuker's proposed conditions for useful measures of software complexity [24], the usefulness of our own measurement can be evaluated—exemplarily, we discuss two conditions.[10] One of the conditions requires that "there exist programs P and Q such that $|P| \neq |Q|$". This requirement motivates that for a measure to have any value at all, it has to enable some discrimination between different programs. Axiomatization coverage fulfills this requirement. Another condition requires that "For all programs P and Q, and the program $P;Q$, which is obtained by combining P and Q, $|P| + |Q| \leq |P;Q|$". The justification for this property is the notion that the interaction between parts of a program may introduce complexity, additional to that present in the components. Hence, the amount of added complexity may only be non-negative. Axiomatization coverage does not fulfill this requirement: due to significant overlap in the minimized preludes needed by two programs, $|P| + |Q| \leq |P;Q|$ usually does not hold. Despite not fulfilling several of Weyuker's conditions[11], our definition of axiomatization coverage has the advantage that test cases can easily be compared because the result of the coverage computation is a single number. While allowing for a quick classification of test cases into *comprehensive* test cases and test cases with *limited scope*, it does not take into consideration *which* elements of the prelude are needed.

Alternative 1. As the first alternative, we suggest to count the number of the covered equivalence classes of "language feature", for example, (1) *array indexed with negative value* vs. *array indexed with the value 0*, (2) *unwrapping an object that is not wrapped* vs. *unwrapping a wrapped object*. In spite of its obvious benefits attributed to the strong relationship to the boundary value analysis, we do not expect it to be measured easily if automatically at all, as it is not clear what a language feature is.

Alternative 2. Going away from the axiomatization coverage and back to the idea of covering language features, a metric can be used that counts the language features used by a test case. In fact, this can be refined by counting the features of the programming language and the features of the annotation language separately. Based on combinatorial testing, an extension of this metric would be the use of the number of "t-wise language feature element combinations" that are covered by a test case. After testing, a high value of this extended metric would allow for a high trustworthiness in the subject under test, as data reported in several studies ([23,20,16]) show that software failures in a variety of domains were caused by the combinations of several conditions.

Alternative 3. Similarly to the last alternative, and by adapting the derived metric LOC/COM (lines of code per line of comments, interpretable as *maintainability*), it is possible to use LOA/LOC, where *LOA* is the number of lines of annotations needed. Thus, it is possible to estimate how many annotations are needed to verify a given code block. A high ratio may indicate code that is difficult to verify, while a low ratio may indicate easily verifiable code.

Further alternatives are imaginable, but as the way of defining the metric gets more complicated, it becomes less conclusive how to actually interpret the results. In general,

[10] In Weyuker's notation the letters P, Q, R represented distinct programs, and the result of the adequacy measurement was signified by $|P|$, $|Q|$, $|R|$.

[11] The discussion was omitted because these are all conditions based on the composition of programs, which cannot be fulfilled due to the overlap in necessary prelude elements.

it is very likely that a single metric is never able to be presentable as a single expression for software quality because the objectives targeted by the models of software quality tend to be multi-dimensional and hierarchical.

4.4 Sources of Test Cases

In the following, we address the issue of producing meaningful test cases, as the systematic creation of a large number of meaningful tests is not trivial. To reduce the impact of erroneous test cases on the testing process, we obtain the test cases from three independent sources, as a form of cross-validation.

At first, we explored the possibility of using the official C language standard. In the next step, we analyzed existing C compiler test suites and investigated ways of adapting those tests. Finally, tests from other verification systems were analyzed for their possible adaption.

C Standard The C standard ISO/IEC 9899:201x, often called *C1X*, was selected to be the first source of possible tests. Although it does not include tests, it specifies the form and establishes the interpretation of C programs. The standard uses the Backus-Naur form for the syntax and prose for the semantics and constraints.

Due to the implementation details of the VCC toolchain, two variations of the C language had to be considered that deviate partially in language features supported compared with C1X. For neither a comprehensive lists of the supported C language features are available, leading to only partially usable C test cases.

In the following, we demonstrate how test cases can be constructed, based on the sources of information on the supported C standards, based on the first paragraph of C1X's section 6.3.2.3 on pointers:

> **6.3.2.3 Pointers**
> A pointer to `void` may be converted to or from a pointer to any incomplete or object type. A pointer to any incomplete or object type may be converted to a pointer to `void` and back again; the result shall compare equal to the original pointer.[...]

Based on the information gathered from the standard we created a single test file for this paragraph. Exemplarily, we annotated the test file with as much information as possible; that is, we have listed the motivational source for the tests, the links to supplementary information, and we have quoted the sentences from the standard's paragraph. Thus, we achieve a strong link between the standard and the derived tests. An excerpt of the full version, which was successfully verified by VCC in accordance with the standard, is presented below:

```
1  //Scope: C1X 6.3.2.3 Pointers, p. 61f
2  #include "vcc.h"
3
4  void function6323_1(void) {
5    //object types:
6    char* b;  [...]
7
8    //paragraph 1: 1. A pointer to void may be converted to or from a pointer to
9    //any incomplete or object type.
10   b = (char*)v; v = (void*)b;  [...]
11
12   //paragraph 1: 2. A pointer to any incomplete or object type may be converted
```

```
13   //to a pointer to void and back again; the result shall compare
14   //equal to the original pointer.
15   assert(b == (char*)(void*)b);  [...] }
```

C Compiler Test Suites With VCC being a special kind of compiler, the construction of test cases using compiler test suites is a straightforward approach. Test suites that check conformance to standards are often called validation suites, and those validation suites are very influential. Several commercial validation suites are available on the market (for example the ACE SuperTest or the Perennial ANSI C Validation Suite), although for licensing reasons we chose to use the test suite of the GNU Compiler Collection (GCC) version 4.4.0[12]. The C compiler of GCC supports C90, and parts of C99. The C specific part (about 12,000 files) of the GCC test suite contains generic tests that are supposed to run on any target, and platform specific tests. Information on the purposes of the tests is limited, thus complicating the analysis of the test cases.

In addition to verifying developer-defined functional properties, VCC implicitly checks for undefined behavior, such as, null pointer dereferences, division by zero, over- and underflow. It does so by automatically inserting additional assertions into the verification conditions, which precede the translated operation. Because of these checks, non-annotated source code normally cannot be verified, despite the lack of explicitly stated functional properties. Therefore, minimal annotations have to be included, for example, the definition of `writes` and `reads` clauses.

We used an iterative approach to adapt files from the GCC test suite. First, we checked whether Microsoft's own C compiler can compile the source code without warnings or errors. Then, minimal annotations were added, so that VCC verifies the source code without warnings or errors. Finally, additional specification was added based on comments and a close inspection of the C code, defining pre- and postconditions, as well as invariants to capture the functional properties of the program. This step-wise approach is demonstrated in Figure 1. There, we used the information given in the `main` function to construct the postcondition. In this function, another function `f` is called first, and subsequently, the result of `f` is checked against what seems to be the expected result of `f`, that is `if (b != 9)`. If the expected result is not met, the program stops abnormally, otherwise it stops normally. Both situations return different exit codes, which are then interpreted by the test framework.

Verification Environments The motivation behind this approach is that the time intensive task of creating interesting test cases for verification environments can be saved by adapting existing test cases.

VCC. VCC version 2.1.20731.0 is deployed with a set of 400 test cases, addressing specific domains, such as *arrays*, *claims*, or *ghost code*. Again, information on the tests' purposes is very limited; however, some information on the background can be obtained by an experienced user of VCC by reviewing the source code in combination with the expected result. Out of the 400 test cases, 202 can be regarded as true positives, and 198 as true negatives. This surprisingly balanced ratio indicates that the developers of VCC use a systematic testing approach to test succeeding and failing verification.

[12] GNU Compiler Collection: 4.4. 3 May 2010 `http://gcc.gnu.org/gcc-4.4/`

```
1  #include "vcc2.h"
2  int b;
3
4  void f ()
5  writes(&b)                              //A
6  ensures(old(b)==0 ==> b==9)             //B
7  ensures(old(b)!=0 ==> b==old(b))        //B
8  {
9    int i = 0;
10   if (b == 0)
11     do
12     invariant(0<=i && i<10) {           //B
13       b = i;
14       i++;
15     } while (i < 10) ;
16 }
17 int main ()
18 writes(&b)                              //A
19 ensures(old(b)==0                      //B
20           ==> result == 0)
21 ensures(old(b)!=0 && old(b)!=9         //B
22           ==> result == 1)
23 {  f ();
24   if (b != 9) return 1;
25   return 0;
26 }
```

Fig. 1. Demonstrating the iterative adaption of the GCC test case files. The test case file `990604-1.c` without any annotations is amended with minimal annotations (lines marked with `A`), and with functional specifications (lines marked with `B`).

Spec#. The collection of verified algorithms from Leino and Monahan [18] contains 38 relatively complex real-life algorithms,[13] such as an insertion sorting algorithm and a minimal distance algorithm. The algorithms are written in C# and verified by Spec#. Furthermore, the algorithms are relatively well documented.

Frama-C/Jessie. The Framework for Modular Analysis of C programs (Frama-C)[14] is a set of program analyzers with Jessie as the deductive verification plug-in. Using the Why back-end [13], automatic theorem provers can be used to perform fully automatic verification. The C files are annotated by using ACSL (see Page 3), which is comparable to VCC's annotation language. Similar techniques are used to express, for example, method contracts, invariants of loops and data structures, and ghost code. As of the Frama-C release Beryllium 20090601, the distribution comes a set of 236 test case files for the Jessie plug-in. Compared to the Spec# tests, the translation is slightly more complex because the annotation language shares less common concepts with VCC's than Spec#'s. Still, the annotations can be very helpful, especially when invariants are provided.

Comparison of the Sources During the exploration of the different sources for test cases, we have made several observations. Based on these, the above-mentioned approaches can be compared both qualitatively and quantitatively.

The highest assurance level is given when the programming language standard is used to derive test cases. However, this is the most labor-intensive approach. Furthermore, it may be difficult to determine if a failed test is caused by a misinterpretation of the standard, or by an incorrect implementation inside the verification tool. Nevertheless, this approach may be useful when corner cases are needed.

An almost arbitrary number of test cases can be created by adopting C compiler test suites, which is less labor-intensive than the standard-based one. However, the task

[13] Rosemary Monahan: Verified Textbook Examples. 3 May 2010 `http://www.rosemarymonahan.com/specsharp/`

[14] CEA-LIST/INRIA-Sacley: Frama-C. 3 May 2010 `http://frama-c.cea.fr`

remains very time consuming, and debugging may be difficult because the C compiler and the verification tool may have different implementations of the C features.

The use of other verification tools as sources has the potential to offer substantial support to find the annotations needed for full functional verification. But the number of the transferable tests is relatively small, and it cannot be guaranteed that two different verification tools are capable of verifying the same functional properties of a program.

4.5 Test Framework

We implemented a framework that allows for the automated execution and evaluation of tests. This enables us to find errors in VCC, and to perform the regression testing of VCC and of a code base. The framework has the benefit that it can be reused without any changes at all if VCC supports further programming languages in the future. Furthermore, it can be adapted with little effort to other fully automatic verification environments, if the axiomatization used by these environments is externally modifiable. Support for interactive verification tools is not implemented, although it should be possible to some extend using capture/replay tools.

5 Test Results

In the previous section, we have laid the basis for testing the Verifying C Compiler. The theoretical background was investigated, a suitable test objective defined, and sources for test cases were explored. In this section we present the results of the actual runs of the test framework.

5.1 Prelude Coverage Results

Exemplarily, our test harness computed the axiomatization coverage that is achieved by VCC's own test suite. The used test suite contained 400 test cases, which were automatically extracted from the test suite collection files of VCC 2.1.20731.0. The harness determined the axiomatization coverage that is achieved on the axiomatization of VCC version 2.1.20731.0, and for comparative reasons the coverage on the axiomatization of VCC version 2.1.20908.0 (a version about 6 weeks further into development). The earlier axiomatization contains a total of 858 elements, of which 575 where covered. Out of these 858 elements, 186 of the 378 axioms were covered; the other elements are, for example, type and constant definitions, and helper functions. To give the reader an idea of how this axiomatization is organized: 1) the class of C language features contains 432 elements, containing 212 axioms, of which 84 were covered, and 2) the class of specification language features contains 426 elements, where 102 of the 166 axioms were covered. Regarding the latter axiomatization, it contains 896 elements in total, of which 509 were covered, and only 139 out of 384 axioms were covered.

Before the runs, we had expected that the number of covered elements would stay roughly the same because features that were added to VCC (indicated by the 38 additional elements) could not be tested by the old test cases. However, the number declined significantly from 575 to 509 elements. Investigations reveal that the reason for this is that axioms and procedures were modified, for example, by removing requirements or

by adding predicates. These changes lead to the necessary inclusion of less elements, which is observed in smaller minimized preludes. Based on the detailed information from the test runs, it is possible to establish links between the different minimized preludes required by a test case and the changes made to VCC's source code and prelude. The old tests can still be regarded as relevant to testing VCC, however, they proved to be less adequate to the newer version of VCC. This can be observed in the decline in the overall coverage from 67.0% to 56.8%.

In subsequent tests, we were sometimes able to construct tests so that the use of a specific axiom was triggered. When creating such tests, one has to deal with the complex and not visible interaction between the elements and the dependencies among themselves. Although, this approach could be used to systematically add tests to the test suite, it becomes increasingly difficult to cover previously uncovered elements.

5.2 Issues Encountered

During our investigations, we encountered several issues. For example, we discovered one bug in the axiomatization with regard to the ownership model. Furthermore, we discovered one case of VCC being more lenient than Microsoft's C compiler, as it did not care about a missing semicolon after the declaration of a C structure. Among the minor issues are the following: we discovered problems with the interaction of VCC's command-line parameters, and one problem with VCC's model viewer showing outdated values. Regarding Boogie, we encountered an incompatibility with file names starting with numbers, as they yielded Boogie identifiers with illegal characters, which in return caused Boogie to report errors.

6 Conclusions

The way verification environments are currently tested is not very satisfying. Most of the time, the tools are written by researchers, and as long as those environments are not actually used for the verification of critical systems, there is no real demand for trust in these systems. The Verifying C Compiler (VCC) is one of these tools that are being used for the verification of industrial products. In this paper, we investigated systematic approaches for the validation of verification systems. Once we had specified what it means for a verification system to be correct, we were confronted with the generation and assessment of test cases for VCC.

The input domain of such a system is the result of the manifold combination of elements from the C programming language and from the language of verification-specific annotations. We reduced the risk of constructing incorrect test cases by choosing trustworthy sources, such as, the official C language standard ISO/IEC 9899:201x, the test suite of the GNU Compiler Collection, and the test suites of the verification tools Spec# and Frama-C/Jessie. While the standard offers the highest level of trust, the derivation of test cases is the most labor-intensive one. Tests adapted from other verification tools have the potential to offer substantial support to find the annotations needed for the functional verification. But the number of those tests is relatively small, compared to those available from the compiler test suite. The latter, however, does not contain any annotations, which are not easy to come by.

When we investigated how the individual components of verification systems can be tested and how the test cases can be assessed, we realized that the common approach of measuring the percentage of the system's executed code statements would not take into account the special features of verification systems. Therefore, we defined *axiomatization coverage* as our domain-specific metric for VCC, in order to assess the test cases and to observe the impact that the individual elements of the axiomatization have on the verification process. Concerning the tests we performed, we noticed that VCC's own test suite requires only about 60% of the axiomatization. Based on this fact and on further observations, we draw the following two conclusions: (1) The used part of the axiomatization seems to correctly reflect the developers' assumptions about how the verification methodology is supposed to work, and (2) additional test cases should be written to achieve a higher coverage. If it is not possible to trigger the use of the prelude's element, the importance of this element should be reconsidered.

In the future, we plan to investigate ways of automatically annotating original C compiler test case files with as many annotations as possible. One way would be to statically analyze its abstract syntax tree (AST) and then modify it. An abstract syntax tree is a tree representation of the syntactic structure of the source code, where each node of the tree stands for a construct occurring in the source code. Once the AST is created, it can be modified, and then the tree can be unparsed to obtain the refactored source of the C program. Additionally, we plan to extend the experiments by determining the "regular code coverage" that is achieved by our set of tests. This will enable us to compare our coverage approach with those established in the software testing community.

Acknowledgments We thank the anonymous reviewers for their comprehensive and helpful comments.

References

1. W. Ahrendt, A. Roth, and R. Sasse. Automatic validation of transformation rules for Java verification against a rewriting semantics. In *12th International Conference on Logic for Programming, Artificial Intelligence and Reasoning (LPAR)*, volume 3835, pages 412–426. Springer, 2005.
2. P. Baker, Z. R. Dai, J. Grabowski, I. Schieferdecker, Øystein Haugen, and C. Williams. *Model-Driven Testing: Using the UML Testing Profile*. Springer, Secaucus, NJ, USA, 2007.
3. M. Barnett, B.-Y. E. Chang, R. DeLine, B. Jacobs, and K. R. M. Leino. Boogie: A modular reusable verifier for object-oriented programs. In *3rd International Symposia on Formal Methods for Components and Objects (FMCO)*, volume 4111 of *Lecture Notes in Computer Science*, pages 364–387. Springer, 2005.
4. 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. volume 4661 of *Lecture Notes in Computer Science*, pages 10–29. Springer, 2006.
5. B. Beckert, T. Bormer, and V. Klebanov. On essential program annotations and completeness of verifying compilers, 2009. unpublished.
6. B. Beckert and V. Klebanov. Must program verification systems and calculi be verified? In *3rd International Verification Workshop (VERIFY), Workshop at Federated Logic Conferences (FLoC)*, pages 34–41, 2006.

7. L. Burdy, Y. Cheon, D. Cok, M. Ernst, J. Kiniry, G. T. Leavens, K. R. M. Leino, and E. Poll. An overview of JML tools and applications. *International Journal on Software Tools for Technology Transfer*, 7(3):212–232, June 2005.
8. S. Chatterjee, S. K. Lahiri, S. Qadeer, and Z. Rakamaric. A reachability predicate for analyzing low-level software. In *13th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, volume 4424 of *Lecture Notes in Computer Science*, pages 19–33. Springer, 2007.
9. 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 *Theorem Proving in Higher Order Logics (TPHOLs)*, volume 5674 of *Lecture Notes in Computer Science*, pages 23–42. Springer, 2009.
10. E. Cohen, M. Moskal, W. Schulte, and S. Tobies. Local verification of global invariants in concurrent programs. In *Computer Aided Verification (CAV)*, Lecture Notes in Computer Science. Springer, 2010. To appear.
11. D. R. Cok and J. R. Kiniry. ESC/Java2: Uniting ESC/Java and JML. *Lecture Notes in Computer Science*, 3362:108–128, 2005.
12. L. de Moura and N. Bjørner. Z3: An efficient SMT solver. In *Tools and Algorithms for the Construction and Analysis (TACAS)*, volume 4963 of *Lecture Notes in Computer Science*, pages 337–340. Springer, 2008.
13. J.-C. Filliâtre and C. Marché. The why/krakatoa/caduceus platform for deductive program verification. In *Computer Aided Verification, 19th International Conference (CAV)*, volume 4590 of *Lecture Notes in Computer Science*, pages 173–177. Springer, 2007.
14. B. Jacobs, J. Kiniry, and M. Warnier. Java program verification challenges. *Lecture Notes in Computer Science*, 2852:202–219, 2003.
15. B. Jacobs and E. Poll. Java program verification at Nijmegen: Developments and perspective. *Lecture Notes in Computer Science*, 3233:134–153, 2004.
16. D. R. Kuhn, D. R. Wallace, and A. M. Gallo. Software fault interactions and implications for software testing. *IEEE Transactions on Software Engineering*, 30(6):418–421, 2004.
17. K. R. M. Leino. This is Boogie 2, 2008. Working draft 24 June 2008.
18. K. R. M. Leino and R. Monahan. Automatic verification of textbook programs that use comprehensions. In *Workshop on Formal Techniques for Java-like Programs (FTfJP)*, 2007.
19. T. Littlefair. *An Investigation into the Use of Software Code Metrics in the Industrial Software Development Environment*. PhD thesis, Edith Cowan University Mount Lawley, 2001.
20. V. Nair, D. James, W. Ehrlich, and J. Zevallos. A statistical assessment of some software testing strategies and application of experimental design techniques. *Statistica Sinica*, 8(1):165–184, 1998.
21. D. von Oheimb. Hoare logic for Java in Isabelle/HOL. *Concurrency and Computation Practice and Experience*, 13(13):1173–1214, 2001.
22. J. von Wright. The formal verification of a proof checker. SRI internal report, 1994.
23. D. R. Wallace and D. R. Kuhn. Failure modes in medical device software: An analysis of 15 years of recall data. *International Journal of Reliability, Quality, and Safety Engineering*, 8(4), 2001.
24. E. J. Weyuker. Evaluating software complexity measures. *IEEE Transactions on Software Engineering*, 14(9):1357–1365, 1988.