Abstract Testing: Connecting Source Code Verification with Requirements

Florian Merz, Carsten Sinz
Dept. for Theoretical Computer Science
Karlsruhe Institute of Technology (KIT)
Karlsruhe, Germany
{florian.merz, carsten.sinz}@kit.edu

Hendrik Post, Thomas Gorges, Thomas Kropf
Robert Bosch GmbH
Leonberg / Stuttgart, Germany
{hendrik.post, thomas.gorges, thomas.kropf}@de.bosch.com

***Abstract*—Traditionally, test cases are used to check whether a system conforms to its requirements. However, to achieve good quality and coverage, large amounts of test cases are needed, and thus huge efforts have to be put into test generation and maintenance. We propose a methodology, called *Abstract Testing*, in which test cases are replaced by verification scenarios. Such verification scenarios are more abstract than test cases, thus fewer of them are needed and they are easier to create and maintain. Checking verification scenarios against the source code is done automatically using a software model checker. In this paper we describe the general idea of Abstract Testing, and demonstrate its feasibility by a case study from the automotive systems domain.**

I. Introduction

Producing high quality software is cost-intensive and time-consuming. Whereas traditional software engineering techniques based on testing (on different levels) are widely employed and can be considered standard today, the effort that has to be put into software quality assurance is still immense. It can even dominate the total cost of a software project. But still errors in software remain, causing costs in the billions for the U.S. economy alone [25].

Formal verification techniques promise to raise the level of confidence in software products. The techniques available in formal verification range from precise interactive theorem proving on abstract models of the software [12], [20], over fast abstract interpretation procedures [10], [11], [26] (which often, however, possess the drawback of producing many false alarms), to methods based on source code model checking (like counterexample-guided abstraction refinement [5], [18], [19], [28], or bounded model checking [7]). As of today, these techniques are mainly employed in safety-critical areas like the avionics and automotive industry, where high quality standards are vital (besides being required by legal regulations). With recent progress in fully automatic verification techniques (scalability, ease of use), e.g. Microsoft's Verisoft (XT) project[31], these techniques become available for a more widespread use, though.

In this article we describe a new technique that combines software bounded model checking with requirements analysis. The resulting technique, that we call *Abstract Testing*, can be regarded as a generalization of traditional testing, in which a set of test cases is replaced by a *Verification Scenario*. A verification scenario combines a possibly large set of test cases into one proof obligation for a software model checker. The proof obligation is typically formulated in an assume/guarantee style, and its syntax closely resembles the programming language the tester is already accustomed to. Thus no new formalization language has to be learned by the testing personnel.

The strength of our proposed method—and what also differentiates it from previously published work—lies in the connection with requirements. Often a one-to-one correspondence between abstract test cases (resp. verification scenarios) and requirements can be achieved, which links abstract testing much more closely to the requirements and facilitates construction and maintenance of abstract test cases.

Our abstract test cases cover all possible program executions that are linked to a requirement. By using a bounded model checker, all these program traces can be checked at once, by which we achieve a much higher coverage than with (traditional) test cases (see also Fig. 1 for a comparison).

To evaluate our method, we have conducted a case study together with Robert Bosch GmbH on a software product from the realm of automotive driver assistance systems. We will report on this case study and its results in Section III of this article.

II. Abstract Testing

Before defining the notion of abstract testing, let us briefly review how testing of requirements on the source-code level is typically accomplished today:

A. Traditional Testing

To derive a traditional test case from a system requirement, the following steps have to be taken[1]:

1) Locate the source code that is relevant for the requirement either manually or by using e.g. traceability links. We assume—for simplicity of presentation—that the code under test is either already contained in a single

This work was supported in part by the "Concept for the Future" of Karlsruhe Institute of Technology within the framework of the German Excellence Initiative.

[1]We mainly have black-box unit testing of functional properties in mind when talking about a test case.

function $\texttt{f}(\texttt{x}_1, \ldots, \texttt{x}_\texttt{n})$, it can be inlined into a single function[2], or a simple wrapper function can be written as part of the abstract test case that wraps the code under test into a single function.
2) Set up and initialize (auxiliary) data structures that function `f` requires to operate correctly (i.e. build the environment).
3) Fix input parameters for function `f` to some sensible values as specified in the requirement.
4) Determine what makes a correct result of function `f` for the given input parameters and the given environment, based on the information given in the requirement.

Thus, a typical test case may look as follows (in a C-like programming language):

```
traditional_test() {
  initialize_environment();
  x1 = input value 1;
  ...
  xn = input value n;
  y = f(x1,...,xn);
  if(!correct_result(x1,...,xn,y))
     test_failed();
}
```

Initializing the environment can include tasks such as setting up dynamic data structures (e.g. linked lists), initializing global variables, setting up data base connections, among others.

Typically, not only one tuple of input values $(x_1, \ldots, x_n)$ has to be checked for one requirement, but many of them.

Thus, one requirement translates to a whole set of test cases. Test cases can be selected in such a way that "typical cases" are covered or, more elaborately, to fulfill a coverage criterion like, e.g., MCDC (modified condition decision coverage).

Another established method used for achieving a good ratio between the number of found bugs and the number of test cases needed is equivalence partitioning. Input (or output) variables are split into equivalence classes (e.g. too low, valid, too high for numerical input variables) and for each class a single test case is created to represent that class, assuming that all elements of a class behave similarly. Given more than a single input or output variable it is necessary to create a test case for every possible combination of equivalence classes if good code coverage is to be achieved. But this means the number of test cases grows exponentially.

B. Abstract Testing

Abstract testing, on the contrary, can often get by with only one abstract test case per requirement. These cases are often formulated similarly to equivalence classes, e.g. they describe ranges of correct or incorrect input, but in contrast to equivalence partitioning, with abstract testing the burden of covering all combinations of equivalence classes for all other variables is simply passed on to the software bounded model checker.

In abstract testing, test cases (also called *verification scenarios*) are formulated in an assume/assert style on the source code level. Assume statements encode preconditions that have to hold before the function under test is called, assert statements fix postconditions that have to be valid after the function under test has been executed.

Because these assert and assume statements closely resemble calls to the assert() function most c programmers are accustomed to there is no need to learn a new language or syntax. At the same time the preconditions and postconditions closely resemble the constraints used to express equivalence classes, so a tester who is accustomed to this equivalence partitioning will quickly be able to apply abstract testing.

Using pre- and postconditions instead of fixed values can be applied at two levels:

1) *Environment construction:* Instead of setting up variables in the environment to fixed values, constraints can be used. Thus, e.g., the elements of a linked list may be left completely unconstrained (i.e., all possible values are allowed and considered during verification), if only the list structure is of relevance for the test case. Restrictions on environment variables are also possible, e.g., by assuming that a time variable t always has to lie in the range $0 \leq t \leq t_{\max}$.
2) *Selection of input variables:* Instead of selecting concrete values for input variables, constraints can be applied on them. So it can be specified, for example, that $8 \leq x_1 < 16$.

Setting up constraints for certain variables can also be seen as introducing a form of non-determinism: the value of a variable is non-deterministically set to any value that respects the constraint. Thus, when using a constraint in an abstract test case, this can be regarded as using an "oracle" to guess values that lead to constraint violations. When running a model checker, we can check the postcondition for all possible values that satisfy the precondition constraints. With this in mind, an abstract test case then has the following shape:

```
abstract_test() {
  nondeterministically_initialize_environment();
  assume(precondition(x1));
  ...
  assume(precondition(xn));
  y = f(x1,...,xn);
  assert(postcondition(x1,...,xn,y));
}
```

The code to non-deterministically set up the environment, as well as the pre- and postcondition has to be provided by the test engineer. In environment construction, he may use further assume statements in order to obtain an abstract initialization procedure. The `assume(precondition(x))` statements can be regarded as non-deterministically selecting values for the input variables, thus such a line could also have been written as

```
x1 = nondeterministically choose input value 1.
```

Such an abstract test case—which is, due to its simple, programming-language-like syntactical constructs, also close to the formalism a test engineer is already familiar with—can

[2]Inlining is done by software bounded model checking tools automatically as part of the preprocessing of the code.

be passed directly to a software model checker like CBMC [7] or Java Path Finder [32].

Although an abstract test case looks almost like a traditional test case, keep in mind that it covers a possibly large number of concrete test cases (we will see in the case study below to which extent a reduction in the number of test cases is possible). Thus, by using abstract test cases, test generation and maintenance can be greatly simplified.

C. Bounded Software Model Checking

In our case study we have used the bounded model checker CBMC, so let us briefly review the technique of bounded model checking. Bounded model checking (BMC) is a method that was introduced by Biere *et al.* [3] to check properties of hardware designs, but has later been extended to also allow verification of C programs [7]. In BMC, non-determinism can be introduced by `assume` statements (which put constraints on program variables), by directly expressing that a variable can take any value (`nondet` statements), or by using uninitialized variables (e.g. function parameters). Bounded model checking then generates all possible program execution traces (with bit-precision on the data level). If a correctness condition (formulated by means of an `assert` statement) does not hold, the bounded model checker will report this, and also provide a counterexample trace on which this error occurs. BMC cannot handle unlimited recursion, and restricts loop executions to a fixed bound (by unwinding loops up to this bound). If the bound is high enough to capture the system semantics, BMC is sound and complete. If the bound is too low a warning about the possible unsoundness is provided. In practice, at least in our setting of low-level software, this limitation turned out to be not essential. An implementation of BMC for C programs is CBMC [7].

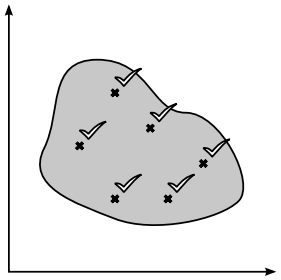
Bounded model checking has already been applied successfully for medium to large scale software projects, e.g. for checking the correctness of Linux kernel modules [27].

Notice that CBMC does not support checking properties formulated in temporal logic (like, e.g., eventually something good happens). For our application this turned out to be not a limiting factor, as all requirements referred to particular program states and could therefore be expressed in a pre-/postcondition form. This closely resembles how traditional unit test cases are formulated.

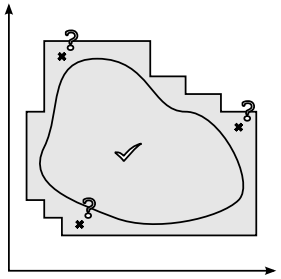
Summarizing, abstract testing is an alternative to traditional software testing, which uses automatic software verification tools (bounded model checkers) to check whole bunches of test cases all at once, such that typically a one-to-one correspondence between requirements and abstract test cases can be achieved.

III. Case Study: Automotive Software

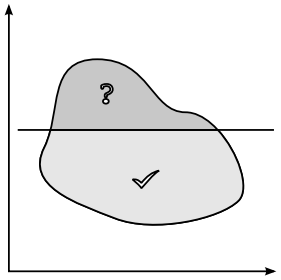
The real benefit of Abstract Testing could only be assessed through practical application. For this reason we carried out a case study to compare Abstract Testing with traditional methods of software testing as applied in an industrial software development project. This was done in the department for



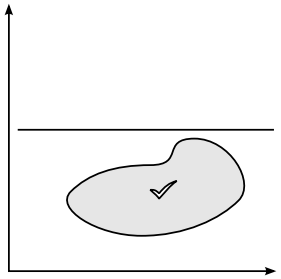
(a) **Software testing** – Few, selected cases are checked for correctness; sound, but not complete.



(b) **Abstraction based verification techniques** – Complete, but may cause false positives and therefore is not sound.



(c) **Bounded Model Checking (BMC)** – Sound, but only execution paths up to a specified length are checked; therefore not complete.



(d) **BMC with large enough bound** – If the bound is chosen high enough, the method is sound and complete.

Fig. 1. Comparing the execution path coverage between different verification methods.

Chassis Systems Control of the Robert Bosch GmbH in Leonberg, Germany.

The case study was realized in the context of a software development project for the Adaptive Cruise Control system (ACC). Over a time frame of six month, the development of the system was accompanied by a student. In parallel to the software testing done by the Bosch engineers, the student attempted abstract testing of the same source code and requirements. Afterwards effectiveness of the methods was compared as well as the complexity of applying each of the competing methods.

A. Background: Driver Assistance Software

Just like a common cruise control system, an adaptive cruise control system tries to maintain the vehicle's speed as previously chosen by the driver. In addition to this, an adaptive cruise control system uses sensors, in our case a long range radar, to monitor the vehicles environment and adapt the driving speed accordingly. For example if there is a truck in front of the vehicle in the same lane, then the vehicle's speed is reduced to match the speed of the truck. As soon as the truck leaves the lane, the ACC system accelerates to the previously chosen speed. This is also illustrated in Fig. 2. This functionality makes ACC itself a driver comfort system, as it is only meant to increase comfort for the driver, not increase safety.

On top of ACC, Robert Bosch GmbH also develops the Predictive Safety System (PSS), which is not a driver comfort system, but a safety system. The PSS system is available in multiple expansion stages, ranging from a passive safety system to a system which is able to issue an emergency braking autonomously if a collision with an obstacle is unavoidable.

Both of these systems can be seen as a good representative of typical embedded code in the automotive industry. The code is low-level and written in C/C++, parts of it are safety-critical, strict processes and coding guidelines are applied during the development of the code and the code is thoroughly checked for correctness. Because of these reasons the code was considered a prime subject for the application of Bounded Model Checking and therefore also for an evaluation of Abstract Testing with a case study.

B. The ACC code base

The software developed for the ACC system consists of several software components which together make up approximately hundred thousand lines of C code. Most of the code is not safety critical, but some of it is very much so. Accordingly, the development process requires extensive testing of these safety critical components. This made those parts of the code especially interesting for an evaluation of the applicability of software verification of safety critical, embedded C source code in general. The safety-critical software components consist of several hundred lines of code, and are well-separated to avoid interference with non-safety critical code. This, together with the fact that the MISRA-C [3] rules ensure bounded runtime, made the source code very well suited for software bounded model checking with the bounded model checker CBMC.

C. Requirements

Before Abstract Testing of the software could be attempted, all functional requirements needed to be identified and extracted from the requirements management tool. For some requirements additional design artifacts were necessary for a later formalization, because the requirements alone did not provide all necessary information. These design artifacts could be extracted from the software documentation system used in the project.

Not all of the requirements were functional in nature. Some referred only to the development process, others to the environment in which the code is executed. Out of a total of 73 requirements, 62 were functional requirements and therefore considered suitable for Abstract Testing.

Initially, not all requirements were present. A considerable number was added only during the case study. It was decided to follow the changes in requirements, instead of staying with the first version of the specification documents, as we were interested in how Abstract Testing could handle dynamically changing specification documents.

Each one of the 62 requirements was further analyzed to determine how these requirements could be formalized into abstract test cases. This showed that all of the chosen requirements were of a simple precondition/postcondition form. They describe a partial program state prior to execution of the code and the expected changes in program state after execution of the code. This also means that no temporal logic was necessary for formalization of the requirements and all requirements could therefore be verified using CBMC.

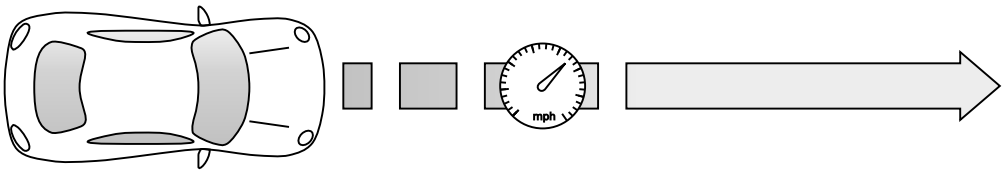
The absense of requirements, which make temporal logic neccessary, is interesting, but can be explained easily. The system, as a real time system, highly depends on precise timing and immediate, correct results. Properties such as "eventually something good happens" is not good enough for such a system. Something good has to happen immediately, before a possibly dangerous situation occurs, not at some unspecified time in the future.

A typical requirement from the specification documents can be seen in Example 3.1.

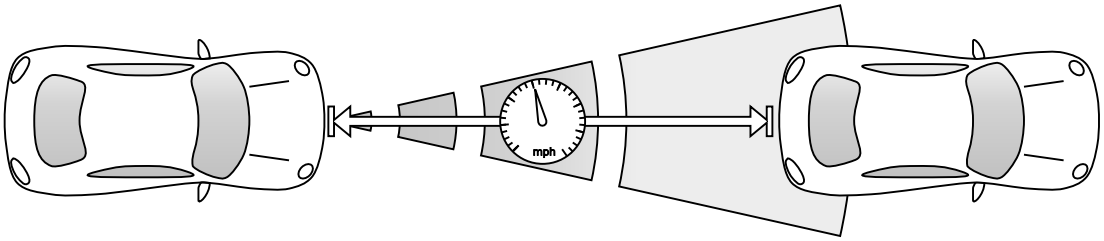
Example 3.1 (Sample Requirement): If the video sensor is not working, the predictive safety system shall not act.

To create an abstract test case from this requirement it is also necessary to include further design artifacts linking conditions in requirements to the variables in C code, which represent these conditions. These can be seen in Examples 3.2 and 3.3, which make reference to two C variables, `VIDEO_SENSOR` and `VETO`.

[3]MISRA-C is a software development standard developed by the The Motor Industry Software Reliability Association (MISRA) containing rules and guidelines for the development of safe, portable and reliably source code for embedded systems in the automotive industry.[24]



(a) The road is free, ACC is accelerating to a driver-set speed.



(b) Another vehicle is blocking the road, ACC is keeping a safe distance.

Fig. 2. The Adaptive Cruise Control System (ACC).

Example 3.2 (First Design Artifact): The variable `VIDEO_SENSOR` is set to one if and only if the video sensor is not working.

Example 3.3 (Second Design Artifact): The driver assistance system acts if and only if `VETO` is set to false.

The information provided by the requirement and the design artifacts is sufficient to create an abstract test case for this particular requirement.

In our case study, abstract, high-level requirements were already broken down into short and concise code-level requirements. These code-level requirements typically consisted of only one or two sentences to describe the desired behavior. The most complex requirement referred to a design document which contained approximately two pages of mathematical formulas needed for calculating a braking distance.

Otherwise, all requirements were kept simple so that the code could be kept simple, too. This was done because of the high safety criticality of this particular code.

An initial assessment of all requirements showed that all of them could easily be formalized for verification.

D. Abstract Test Cases / Verification Scenarios

Because of the concise structure of most requirements, they could be almost directly translated into abstract test cases. For example the requirement shown in Example 3.1 could be translated to an abstract test case as follows:[4]

Example 3.4 (Sample Abstract Test Case):

```
main() {
  havoc();
  assume(VIDEO_SENSOR != 1);
  ...
  component_task();
  ...
  assert(VETO == true);
}
```

[4]The `havoc()` procedure call in the example sets some global variables to undefined (non-deterministic) values.

For the (non-functional) requirements that were removed from the specification document before they could be formalized, no abstract testing was attempted. Equally, those requirements that were only added to the specification documents shortly before the case study was finished were not formalized either. Thus, only a total of 43 abstract test cases were created based on the requirements.

Almost all requirements could be turned into a single, concise abstract test case. For those few, where two or more abstract test cases were needed, it would have been easily possible to adapt the requirements so that a one-to-one relationship could have been established. This clearly shows that there is a much closer link between requirements and abstract test cases than between requirements and traditional test cases, as illustrated in figure 3. It also facilitates the creation of abstract test cases as well as their maintenance, especially when test cases have to be adapted to changes applied to the specification.

For most test cases, the complexity did not exceed the complexity of the sample abstract test case in Example 3.4. Only those requirements that described more complex func-

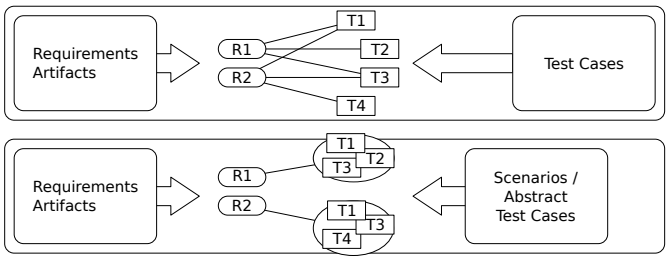


Fig. 3. Linking requirements and Abstract Test Cases.

tional properties, for example the calculation of checksums or braking distances, resulted in more complex abstract test cases, but even those did not exceed 100 lines of code.

E. Results

Traditionally, the worst-case exponential runtime of verification algorithms based on model checking is one of the largest obstacles hindering the industrial application of these tools. In the context of our case study, this turned out not to be a problem at all. Runtime for the bounded model checker CBMC on a single abstract test case was almost always below 60 seconds, and on average even less than 20 seconds. These short runtimes make it possible to use bounded model checking in an agile-like development process, where verification is done early and errors in the code can be detected as soon as the code is passed from the developer to the verification engineer.

A combined approach employing both verification using abstract test cases and traditional software testing uncovered a total of eleven bugs in the code. Out of these, ten bugs were found by verification and nine bugs were found by software testing. The one bug not found by verification was not found due to an error in one of the abstract test cases. Out of the two bugs not found using software testing, one could have been found by the software tests easily, but was not because a necessary test case for this was accidentally omitted. The second bug was not found because it occurs only under very special circumstances. It was located in the error handling code paths of one component at a very specific point and the faulty behavior would only show up if a third condition is also met. The probability that this bug could have been found by software testing were estimated to be very low by the test engineer.

Especially this last error clearly shows that software verification can be more powerful than software testing, and that it can actually be used to detect otherwise missed bugs in the source code.

The fact that both methods missed one bug due to errors in applying the methods shows that the proposed use of software verification is comparable to software testing concerning error-proneness.

The case study also shows that Abstract Testing compared favorably to traditional software testing concerning its effectiveness in finding bugs in the source code. Equally interesting is how both methods compare concerning the cost of creating and maintaining test cases:

This cost can be estimated by the average cost per test case multiplied by the number of test cases. While it took several hundred test cases for software testing[5], an order of magnitude fewer abstract test cases were necessary for complete coverage of all chosen requirements. This became especially obvious when a rather small change in the code caused a large increase in the number of test cases, while only one additional abstract test case was needed.

[5]Due to a non-disclosure agreement we cannot publish the precise number of test cases created during software testing.

We estimate the cost of each individual abstract test case to be of roughly the same order as the cost of a test case in traditional software testing. On the one hand, traditional test cases assign a concrete value to each input variable, such that the program's behavior is easier to understand. On the other hand, a traditional test case can contain a large amount of input variable assignments which is only relevant to ensure complete coverage of the code (for example by coverage metrics such as path coverage, condition coverage etc.). Abstract test cases on the other hand only contain the relevant information for the requirement in question.

While the case study has shows that, at least in our case, Abstract Testing is comparable, if not superior, to traditional software testing in its effectiveness, the study unfortunately does not provide any hard data on the cost of creating and maintaining abstract test cases. Still, our observations suggest that abstract testing are a promising alternative to traditional software test for reducing the cost of the tests.

IV. Related Work

The areas of research related to our proposal are: software verification case studies, formal requirements analysis, work about linking testing and requirements, and model based testing.

It has been demonstrated in many case studies that numerous verification techniques can be applied to real world software. Well known examples include the Microsoft SLAM project [28], which led to an interface specification verification tool that is currently deployed with every driver development kit. Cook et al. present the Terminator [8] tool, which is able to check certain termination properties of windows device drivers. Other examples include abstract interpretation tools, which are successfully applied in avionics industry [9]. CBMC [7] has been successfully applied to numerous complex software systems [21], [27].

Formal requirements analysis deals with formalizations of requirements in an early design phase. Noticeable demonstrations of this technique are given by Dutertre and Stavridou [13] in the area of avionics using non-automatic theorem provers. Crow and Di Vito [12] present a summary of four case studies in space craft industry using non-automatic proof systems. An automata based approach that is more closely related to the model checking technique we presented was proposed by Heitmeyer *et al.* [17]. Miller *et al.* have conducted a case study on using both the model checker NuSMV and PVS for checking requirements of a flight guidance system [23]. In contrast to our work, they did not link the verification to a concrete implementation, however. Also related to our work is that of Staats and Heimdahl [29], where they use CBMC to prove correctness of C code generated by automatic code generators like Simulink.

Chechik and Gannon [6] presented a technique for automatically checking the consistency of requirements and designs (expressed as state machines with event-driven transitions), which resembles our general approach. However, they use light-weight verification techniques which use abstraction on

data flow and are thus less precise than the non-abstracting model checker CBMC that we use.

Uusitalo et al. analyze best practices for linking requirements and testing in industry [30]. Graham argues that testers should be integrated into early development phases [15]: Both sides can profit from a tight linking.

Bringmann and Krämer [4] have conducted a case study in the automotive industry applying model-based testing techniques in this industry. For this, they create an abstract model of the code under test and derive concrete test cases from this model. Fraser and Wotawa [14], too, apply model-based testing, and argue that usefulness of test cases should not be measured in terms of code coverage, but rather in property relevance. Their method uses model checking for generating concrete test cases satisfying this coverage criterion.

Our approach is also related to the *design by contract* approach, as implemented, e.g. in Eiffel. Meyer summarizes verification-oriented aspects of Eiffel in *Eiffel as a framework for verification* [22]:

> Eiffels contracts have so far been applied mostly to dynamic checks, because the benefits are so clear and immediate. With improvements in proof technology—including semantic modeling, theorem provers, abstract interpretation and model checking—it becomes attractive to support proofs, as has already been the plan behind Eiffel.

Our work can be considered as a realization of this idea of generating proofs (using a bounded model checker), although based on C (and CBMC's assume/assert formalism) instead of Eiffel.

Arnold *et al.* ([1], [2]) developed a framework for the specification and execution of testable requirements models. Whereas the general approach is in parts similar to ours, he does not employ verification methods as we do (by employing a bounded model checker). Our abstract test cases cover a possibly very large set of program executions, and a passed abstract test confirms that the checked requirement holds for all corresponding execution traces.

As a representative example of the many articles that are dealing with applications of formal methods in an industrial setting, we want to mention the report of Wassyng and Lawford [33] that evaluates different tools for safety-critical software development in the Canadian nuclear industry. An article by Heitmeyer *et al.* [16] on how to obtain certifiably secure software systems is another typical representative, more related to security properties.

V. Conclusion

In this paper we presented a novel approach to software testing called Abstract Testing. The approach is strongly focused on a close connection between requirements and the corresponding abstract test cases. This decreases the cost of creating test cases and also the maintenance burden for the test engineers considerably, as changes in requirements can be mapped to changes in abstract test cases with little effort.

Abstract Testing has been tried in a case study in a software development project in the automotive industry. The results of the case study show that Abstract Testing can be applied successfully for quality assurance of safety critical embedded software. Abstract Testing turned out to be comparable in effectiveness, if not superior, to traditional software testing. The number of abstract test cases was considerably smaller than in traditional testing, and the complexity of test cases could be reduced to a minimum. All of this made supporting the test cases easier and more efficient than with traditional testing. The method scaled well, even though automatic software verification (model checking) techniques were applied, which traditionally suffer from an exponential blow-up in their state space, and therefore also in their worst-case runtime.

Future work may include extending our formalism to also handle non-functional requirements like run-time behavior, memory consumption, or operation on restricted resources. We believe that, in principle, such properties are also within the reach of Abstract Testing, e.g., by modelling run-time explicitly and using appropriate constraints.

References

[1] D. Arnold, "An open framework for the specification and execution of a testable requirements model," Ph.D. dissertation, Ottawa-Carleton Institute for Computer Science, Carleton University, 2009.

[2] D. Arnold, J.-P. Corriveau, and W. Shi, "Modeling and validating requirements using executable contracts and scenarios," in *Proc. of the 8th Intl. Conf. on Software Engineering Research, Management and Applications (SERA)*, 2010.

[3] A. Biere, A. Cimatti, E. M. Clarke, and Y. Zhu, "Symbolic model checking without BDDs," in *5th Intl. Conf. on Tools and Algorithms for Construction and Analysis of Systems (TACAS), Proc.* London, UK: Springer, 1999, pp. 193–207.

[4] E. Bringmann and A. Krämer, "Model-based testing of automotive systems," in *ICST*, 2008, pp. 485–493.

[5] S. Chaki, E. Clarke, A. Groce, S. Jha, and H. Veith, "Modular verification of software components in C," in *25th Intl. Conf. on Software Engineering (ICSE), Proc.* IEEE Computer Society, 2003, pp. 385–395.

[6] M. Chechik and J. D. Gannon, "Automatic analysis of consistency between requirements and designs," *IEEE Trans. Software Eng.*, vol. 27, no. 7, pp. 651–672, 2001.

[7] E. Clarke, D. Kroening, and F. Lerda, "A tool for checking ANSI-C programs," in *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, ser. LNCS, K. Jensen and A. Podelski, Eds., vol. 2988. Springer, 2004, pp. 168–176.

[8] B. Cook, A. Podelski, and A. Rybalchenko, "Terminator: Beyond safety," in *CAV*, ser. LNCS, T. Ball and R. B. Jones, Eds., vol. 4144. Springer, 2006, pp. 415–418.

[9] P. Cousot, "Proving the absence of run-time errors in safety-critical avionics code," in *Proc. of the 7th ACM & IEEE Intl. Conf. on Embedded software (EMSOFT)*. New York, NY, USA: ACM, 2007, pp. 7–9.

[10] P. Cousot and R. Cousot, "Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints," in *Proc. of the Fourth ACM Symp. on Principles of Programming Languages (POPL), Los Angeles, California, January, 1977*, 1977, pp. 238–252.

[11] P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival, "The ASTREÉ analyzer," in *Programming Languages and Systems, 14th European Symp. on Programming (ESOP),Edinburgh, UK, April 4-8, 2005, Proc.*, 2005, pp. 21–30.

[12] J. Crow and B. Di Vito, "Formalizing space shuttle software requirements: four case studies," *ACM Trans. Softw. Eng. Methodol.*, vol. 7, no. 3, pp. 296–332, 1998.

[13] B. Dutertre and V. Stavridou, "Formal requirements analysis of an avionics control system," *IEEE Transactions on Software Engineering*, vol. 23, pp. 267–278, 1997.

[14] G. Fraser and F. Wotawa, "Property relevant software testing with model-checkers," *SIGSOFT Softw. Eng. Notes*, vol. 31, no. 6, pp. 1–10, 2006.

[15] D. Graham, "Requirements and testing: seven missing-link myths," *Software, IEEE*, vol. 19, no. 5, pp. 15–17, Sep/Oct 2002.

[16] C. L. Heitmeyer, M. Archer, E. I. Leonard, and J. McLean, "Applying formal methods to a certifiably secure software system," *IEEE Trans. Software Eng.*, vol. 34, no. 1, pp. 82–98, 2008.

[17] C. L. Heitmeyer, R. D. Jeffords, and B. G. Labaw, "Automated consistency checking of requirements specifications," *ACM Trans. Softw. Eng. Methodol.*, vol. 5, no. 3, pp. 231–261, 1996.

[18] T. Henzinger, R. Jhala, R. Majumdar, and G. Sutre, "Software verification with BLAST," in *10th Int. SPIN Workshop, Proc.*, ser. LNCS, vol. 2648. Springer, 2003, pp. 235–239.

[19] T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre, "Software verification with blast," in *Proc. 10th Intl. Workshop on Model Checking of Software (SPIN)*, ser. LNCS, no. 2648. Springer-Verlag, 2003, pp. 235–239.

[20] J. J. Hunt, E. Jenn, S. Leriche, P. Schmitt, I. Tonin, and C. Wonnemann, "A case study of specification and verification using JML in an avionics application," in *The 4th Workshop on Java Technologies for Real-time and Embedded Systems (JTRES), Proc.* ACM Press, 2006, pp. 107–116.

[21] M. Kim, Y. Kim, and H. Kim, "Unit testing of flash memory device driver through a SAT-based model checker," in *Int. Conf. on Automated Software Engineering (ASE), Proc.* IEEE Computer Society Press, September 2008, pp. 198—207.

[22] B. Meyer, "Eiffel as a framework for verification," in *First IFIP TC 2/WG 2.3 Conference on Verified Software: Theories, Tools, Experiments (VSSTE'05)*, 2005, pp. 301–307.

[23] S. P. Miller, A. C. Tribble, M. W. Whalen, and M. P. E. Heimdahl, "Proving the shalls: Early validation of requirements through formal methods," *J. Software Tools for Technology Transfer (STTT)*, vol. 8, no. 4-5, pp. 303–319, 2006.

[24] MISRA C Working Group, *MISRA-C:2004 - Guidelines for the use of the C language in critical systems*. The Motor Industry Software Reliability Association, 2004. [Online]. Available: http://www.misra-c2.com/

[25] National Institute of Standards and Technology, "The economic impacts of inadequate infrastructure for software testing," Final report, May 2002.

[26] PolySpace Technologies, "Polyspace Client / Server for C/C++, Version 4.1.1.6 , 2008, http://www.polyspace.com." 2008.

[27] H. Post and W. Küchlin, "Integration of static analysis for linux device driver verification," in *Integrated Formal Methods (IFM), 6th Intl. Conf., Proc.*, ser. LNCS, J. Davies and J. Gibbons, Eds., vol. 4591. Springer-Verlag, 2007, pp. 518–537.

[28] "Microsoft Research. The SLAM Project," http://research.microsoft.com/slam, 2006.

[29] M. Staats and M. P. E. Heimdahl, "Partial translation verification for untrusted code-generators," in *10th Intl. Conf. on Formal Engineering MethodsFormal (ICFEM'08)*, Kitakyushu-City, Japan, Oct. 2008, pp. 226–237.

[30] E. J. Uusitalo, M. Komssi, M. Kauppinen, and A. M. Davis, "Linking requirements and testing in practice," in *RE '08: Proceedings of the 2008 16th IEEE International Requirements Engineering Conference*. Washington, DC, USA: IEEE Computer Society, 2008, pp. 265–270.

[31] "Verisoft - Formal verification of computer systems," http://www.microsoft.com/emic/verisoft.mspx, 2006.

[32] W. Visser, K. Havelund, G. P. Brat, S. Park, and F. Lerda, "Model checking programs," *Autom. Softw. Eng.*, vol. 10, no. 2, pp. 203–232, 2003.

[33] A. Wassyng and M. Lawford, "Software tools for safety-critical software development," *STTT*, vol. 8, no. 4-5, pp. 337–354, 2006.