Cross-Platform Verification Framework for Embedded Systems[⋆]

Ingomar Wenzel, Raimund Kirner, Bernhard Rieder, and Peter Puschner

Institut für Technische Informatik
Technische Universität Wien
Treitlstrasse 3/182-1
1040 Vienna, Austria
{ingo, raimund, bernhard, peter}@vmars.tuwien.ac.at

Abstract. Many innovations in the automotive sector involve complex electronics and embedded software systems. Testing techniques are one of the key methodologies for detecting faults in such embedded systems.
In this paper, a novel cross-platform verification framework including automated test-case generation by model checking is introduced. Comparing the execution behavior of a program instance running on a certain platform to the execution behavior of the same program running on a different platform we denote cross-platform verification. The framework supports various types of coverage criteria. It turned out that end-to-end testing is of high importance due to defects occurring on the actual target platform for the first time.
Additionally, formal verification can be applied for checking requirements resulting from the specification using the same model generation mechanism that is used for test data generation. Due to a novel self-assessment mechanism, the confidence into the formal models is increased significantly.
We provide a case study for the Motorola embedded controller *HCS12* that is heavily used by the automotive industry. We perform structural tests on industrial code patterns using a wide-spread industrial compiler. Using our technique, we found two severe compiler defects that have been corrected in subsequent releases.

1 Introduction

The last years have seen significant advances in electronic control systems. Such systems replace more and more conventional control systems. Driven by the increased flexibility resulting from the use of microprocessors in control systems, increasing functionality is integrated into electronic control units (ECUs) causing higher complexity of the control applications. This causes an increase in the number of safety-critical electronic control systems. Safety-criticality means that a failure of such systems may result in catastrophic consequences [1].

[⋆] This work has been partially supported by the FIT-IT research project “Systematic test case generation for safety-critical distributed embedded real time systems with different SIL levels (TeDES)” and the FWF research project “Compiler-Support for Timing Analysis” (CoSTA).

For instance, the automotive industry is one of the key drivers pushing these developments. Today, new cars contain about 80 ECUs. According to current figures of the Austrian automobile association ÖAMTC, about 27% of all car breakdowns are directly related to defects of car electrics and electronics [2].

Thus, the correct operation of safety-critical systems has to be ensured. This confidence is established by validation and verification [3, 4]. Validation denotes checking the specification, verification refers to whether a system fulfills given requirements. Within verification, testing is a method for getting evidence of the absence of faults.

High-level development tools like *Matlab/Simulink*[1] supporting model-based development help shortening product cycles. Especially code generators boost software development cycles for embedded systems. In such development environments C-code targeting specialized embedded hardware is generated automatically from *Matlab/Simulink* models.

Input to the verification framework is C-source code performing a specified function in the embedded system. This source code is compiled for the target platform and expected to behave according to its specification. The notion of the cross-platform verification framework allows to verify embedded systems software by comparing the execution of the software on the target platform with an alternative execution platform (denoted as host platform). To allow an semantics-equivalent execution of software, it is necessary to transform the program for proper operation on the host platform.

1.1 Contribution

First, the notion of cross-platform verification is introduced. Target-specific C-code is parsed and platform-semantics-equivalent models are generated that are used for test-data generation. These models can also be used for formal verification purposes.

Second, a verification framework has been developed. Instrumented test files are produced and it is checked whether the generated test data fulfill their purpose in the execution scenario.

Third, cross-platform checks using the actual system-under-test allow to determine if the target implementation operates semantically equivalent to the generated test data of the reference implementation. This increases the confidence in the generated models. We found out that this novel self-assessment mechanism helps to identify severe errors within the system development processes. This mechanism allows strengthening the confidence in the overall framework.

Fourth, using this method we found two severe bugs in a widely used commercial cross-platform compiler. For legal reasons, the company name is not published.

1.2 Structure of this paper

The paper is structured as follows: First, basic concepts including verification, testing, model checking and platform behavior are introduced (Section 3). Then, individual patterns used in testing are formulated and composed into the novel concept of cross-platform verification (Section 4). In Section 5, the implementation of the cross-platform

[1] `http://www.mathworks.com`

verification framework is presented. Subsequently, we present experimental results exercising the framework for an embedded system (Section 6). Finally, Section 7 concludes this paper.

2 Related Work

To the best of our knowlegde there is no scientific research that addresses cross-platform verification as a tool to verify a program instance running on a certain platform by comparing its behavior with another program instance running on a different platform.

A research area related to our approach is test case generation using formal methods, which is an active area of research. Chlipala et al. [5] describe how to test reachability using the model checker BLAST. Rayadurgam et al. [6] have worked on generating test of MC/DC coverage using model checkers to comply with the standard DO-178B [4]. The use of model checking to create test cases for mutation testing is described by Ammann et al. [7]. Hamon et al. present results on using the model checker SAL to generate tests for complete state and transition coverage of Stateflow models [8].

3 Basic Concepts

Verification denotes the process of checking whether a system fulfills given properties called verification conditions [1]. Some people consider testing and verification as alternative strategies for increasing the dependability of computer systems. However, in the following, a classification of verification techniques is presented that is based on work of Laprie et al. [1]. The highest level classification criterion is whether the system is exercised or not. If a system is verified without actually executing it, this is called *static verification*. When a system is verified by executing it, this is referred to as *dynamic verification*

If static verification is based on analyzing *the code itself*, we distinguish between *static analysis* (inspections, walk-through, abstract interpretation, etc.) and *theorem proving*. If verification is performed on a *model* of the system behavior (whereby the model usually is a state transition system represented by finite or infinite state automata), this is called *model checking* [9].

When using dynamic verification, the system is exercised by providing inputs to the system. These inputs can be symbolic in case of *symbolic execution* or concrete in case of *testing*.

The cross-platform verification framework is designed for supporting *testing* and *model checking*. Thus, in the following, these two techniques are introduced.

4 Cross-Platform Verification

In this section we introduce the basic verification patterns that are instantiated within our verification framework. Most of these patterns are already used in practice; however, due to the lack of a conceptional description in research literature the respective key concepts are formulated explicitly.

Beside these patterns, functional equivalence between two different platforms is required in order to ensure that the target semantics are equivalent to the host semantics. Especially, this is required for the formal model that is automatically built and used within the framework.

Subsequently, the combination of the concepts of remote testing, cross-platform testing, reference-platform testing, formal verification, and functional equivalence leads to our cross-platform verification framework is outlined in detail.

4.1 Remote Testing

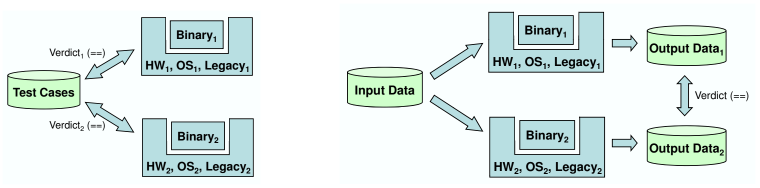
A scenario where the test-control software and the software under test run on different components is called *remote testing*.

When testing embedded systems, the component of the embedded system to be tested is called *target*; the component where the testing control software runs is called *host*. When testing embedded software using automatic test data generation, the host and target are typically different components, i.e., forming a remote testing scenario. Remote testing is used in this case due to resource limitations at the target.

In case of our verification framework we verify the automatically generated test data in a first step on the host where it is generated. In order to verify the correctness of the generated test data, it is verified in a first step on the host where it is generated. Then, in the next step the verified test data are converted and sent to the target platform. This technique increases the confidence of test data significantly. It ensures that the test data fulfills exactly its purpose.

4.2 Cross-Platform Testing

Cross-platform testing aims at performing tests on two or more different platforms with semantically equivalent code and test data.



(a) Cross-platform testing. (b) Reference-platform testing.

Fig. 1. Cross-platform vs. reference-platform testing.

The concept of cross-platform testing is shown in Figure 1(a). The software is tested on platform 1 and 2, using the same test cases. The intended output of the test cases may need to be transformed differently for platform 1 and 2 to compute the test verdict.

Cross-platform testing has to be used to verify safety-critical systems in the case that important parts of the system are computed with diversity on either hardware or software level.

4.3 Reference Platform Testing

Reference platform testing is a technique to achieve automatic test verdicts even if only input test data instead of complete test cases are available. The input test data are processed on two different components, where one of it is typically called *reference platform*.

As shown in Figure 1(b), the test data are applied on both components and the test verdict is obtained by comparing the observed output of both components.

If the two components provide different execution platforms, it is in general necessary to transform the program code of the reference platform to obtain the same intended behavior as of the other platform.

4.4 Functional equivalence between two platforms.

When doing cross-platform verification, one has to keep in mind that the program semantics may include properties not only of the target hardware, but also of a program transformer, e.g., a C compiler. In case of the ANSI C programming language, the program semantics of a sequential C program does not even completely describe the program behavior in the value domain [10]. We call a sequential program's semantics of the value domain *functional semantics*.

Thus, verifying a platform by comparing its behavior in the value domain with a reference platform has the problem that both platforms may behave differently.

Our way to resolve this is to transform the program before executing it on the reference platform (B), such that it exhibits the same functional semantics on the reference platform as on the target platform (A) (functional equivalence between platforms).

4.5 Verification of Embedded Systems

Our verification framework [11] allows to analyze embedded software both, by testing and formal verification. It allows to dynamically test the program and to formally verify the software using the identically generated models. The key idea is that the test data required for testing are generated by model checking. To do this, a formal model of the program is required as input to the model checker. When this model has been built, it can be used for the test data generation using model checking and for formal verification. Thus, in the framework two platforms are supported: the host platform involving the model for the model checker and the target platform (the embedded system). The test data that have been generated using model checking are subsequently used to exercise the program running on the target hardware. During these test runs, it is checked whether the test data produce correct results (both, test data and results have to be transformed to ensure functional equivalence between the platforms). One of the core contributions of this framework is that this way of using the formal model increases the confidence in the model. If any faults are observed within these executions, the reasons have to be investigated: the cause can be located within the framework (i.e., erroneous model transformation) or within the execution platform (i.e., hardware, compiler). Thus, the presented framework forms a kind of intelligent self-assessment mechanism helping to create correct formal models and to ensure the correctness of the applied transformations.

Testing Framework Besides the above mentioned formal model representing the core of the verification framework, reference platform testing is supported. The transformed target program can be executed on the host which yields execution traces. These traces are recorded and compared to executions performed on the target hardware. It is also possible to use test data generated by model checking to exercise the transformed target program on the host in order to obtain a preflight check of the generated data. On the first glance, this may look a little bit strange, however, in practical use it turned out that each of these components is very useful.

Formal Verification As we use a formal model of the software under test, it is possible to use the same model for formal verification. We can extract this formal model directly from the software. However, formal verification is not sufficient to ensure correct behavior of the software. This is because the behavior of a component is defined by the software as well as its execution platform and it is practically infeasible to model and analyze the PSS of the execution platform in full detail.

However, we have to model the PSS at least partially, since as introduced in Section 4.4, the PIS of ANSI C source programs in general is not enough to get the behavior of the software in the value domain.

5 Cross-Platform Verification Framework

The verification framework uses model-based testing to generate test cases. Since we use model checking for our test case generation, it is relatively easy to derive test cases that feature a specific coverage criteria. As with conventional testing, using such a framework does not permit to apply the ideal testing metrics of path coverage. And of course, achieving even more comprehensive coverage like state coverage is far from feasible. In practice, coverage-based testing is reduced to coverage metrics that cover local structures of the control flow graph, e.g., statement coverage or branch coverage.

Our testing framework provides additional flexibility as it allows us to decompose the program into segments of parametric size and achieve path coverage within each of these segments. The verification based on such a segmentation is used to verify the execution platforms of our measurement-based WCET analysis [12].

As a feature of our framework, we utilize the same formal model for both, formal verification and testing. The key drivers for the architecture of our verification framework are:

1. **Automated test data generation:** Test data should be generated completely automatically. We use bounded model checking for this purpose.
2. **Extendability:** The extension to new test cases (formal verification and cross-platform testing) as well as to new target platforms should be easily possible. This flexibility is achieved by using ANSI-C as language for applications subject to test.
3. **Cross-platform testing:** Due to faults in the build environment or hardware configuration, end-to-end testing is necessary even for unit testing.

5.1 Overview

In Figure 2 the architecture of the framework is illustrated. The big gray shaded background arrows show processing that is performed completely automatically by the framework. There are no user interactions required. Following, we present an overview about each step performed by the framework.

The C-code passed to our tool is strongly dependent on the actually used target platform.

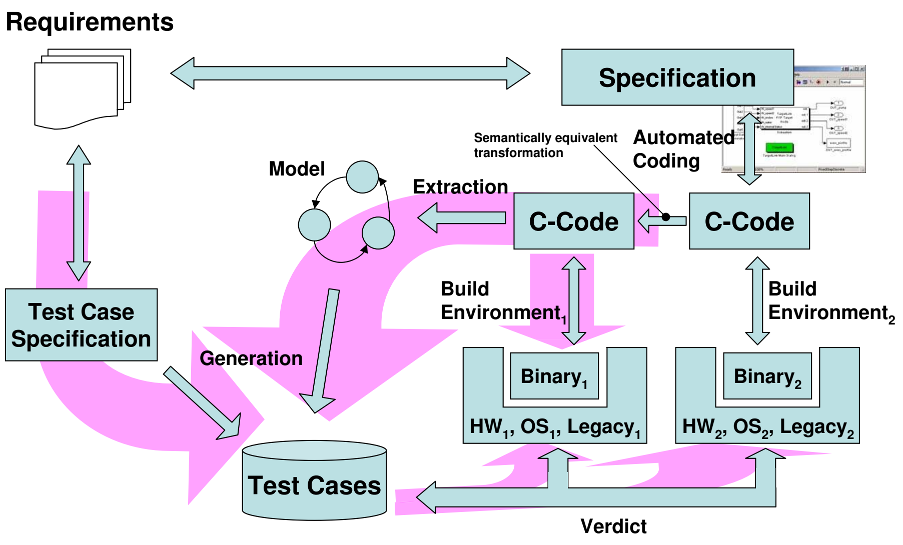


Fig. 2. Verification framework.

Thus, in the first step the code for the target platform is transformed to a semantically equivalent C-code for the host platform. The transformed code is used as a ”reference implementation” for testing hardware and compiler effects.

Further, from this C-code we extract models with respect to the test case specification, i.e., depending on the actual test case specification modifications are automatically applied to the source code (e.g., adding additional constraints from the specification required by the test case).

These modified source codes are passed to the CBMC model checker [13]. The model checker generates the desired test data by deriving a counter-example. Failures occuring in this step indicate possible contradictions in the test case specification. For instance, when generating data for simple C1 coverage (basic block coverage), test data can always be provided as long as the code is reachable. Whenever no test data can be found, the program may contain unreachable code.

Finally, the resulting test data and reference results are delivered to the execution environments. This is a two phase process: first the sample data is issued to the host platform to verify the generated test data. Usually this process finishes successfully.

Then in the second step, the respectively converted test data is provided to the target platform execution environment. Finally, the results calculated on the target platform are compared with the results on the host platform, resulting in a final verdict.

Section 5.2 outlines the process of code transformation in more detail. Section 5.3 explains the test data generation process.

5.2 Code Transformation

A code transformation is performed to generate code for the host platform with exactly the same behavior like the original code on the target platform. For applications written in C the conversion mainly affects integer data types since floating point numbers are defined according to the IEC 60559 single, double, and extended format. The size and the binary layout of the integer types are hardware and software (operating system and compiler) dependent. In addition the C Standard does not define whether a “char” is the same as a “signed char” or an “unsigned char”.

To be able to generate code on the host platform that is equivalent to a given piece of code on the target platform we need the knowledge of the individual value ranges of the integer data types. This knowledge is used to generate a mapping of data types so that for each of the integer types $min(type_{TARGET}) = min(type_{HOST})$ and $max(type_{TARGET}) = max(type_{HOST})$. If there is no equivalent data type on the host platform for a specific data type used on the target platform, no semantically equivalent code transformation can be performed. Type casts can be transformed in the same way.

Signed integer types need special consideration. It has to be ensured that these types are treated in the same way on the target platform and on the host platform. The behavior of some arithmetic and binary operators is not defined in the ISO standard, i.e., it is not specified within the PIS. Therefore $-1 \gg 1$ might be 0, but using a different compiler or hardware platform it might be $0x40$.

The behavior of bit fields is undefined. The ISO standard does not state if an implementation has to restrict variable values to fit the width given in the declaration. If `a` is declared as “`struct { unsigned int a : 3; }`” then `a` should be in the range from zero to seven. The ISO standard does not describe the behavior for the case that a value outside this range is assigned to `a`. The same behavior on the target platform and on the host platform has to be ensured.

There are some compiler or platform specific extensions like `#pragma` directives or `_near` or `_far` modifiers, whose influence on the code semantic has to be examined individually. Modifiers like `_near` or `_far` that influence neither data types nor control flow can be expected to conserve the semantics of the application.

Target-specific hardware has to be available or to be emulated on the host platform if it is used in the examined application. If neither is possible, then no semantically equivalent code transformation can be performed.

When a code transformation is performed the target platform code is parsed into a syntax tree. Based on this tree, the host platform code is generated. However, integer types have to be modified according to the rules given above. If there are any constructs that cannot be converted safely, the transformation has to stop with an error message.

Figure 3 shows the code on the target platform (with a word size of 16 bit) and the semantically equivalent code generated for the i386 platform. In this case the transformation from the target to the host architecture has been accomplished by converting the `int` data type on the target to the `short` data type on the host.

```c
int max (int a, int b)
{
    int max;
    if (a>b) {
        max = a;
    } else {
        max = b;
    }
    return max;
}
```

```c
short max(short a, short b)
{
    short max;
    if (a>b) {
        max = a;
    } else {
        max = b;
    }
    return max;
}
```

Fig. 3. Target code vs. host code.

5.3 Test Data Generation

The target platform code cannot only be transformed into semantics-equivalent code for the host platform. It is used to generate a semantics-equivalent application model that can be used with a model checker. The complexity of this task depends on the model checker syntax and semantics. For model checkers like CBMC [13], which uses models specified in C, the implementation effort is much lower compared to other model checkers like SAL, whose input language strongly differs from C.

In the described testing framework, model checking is performed within an external module which allows to use either CBMC or SAL depending on the size and complexity of the examined application. Since model checkers often use exact math in contrast to C which uses residue classes for arithmetic operations, the behavior of C arithmetic operations has to be simulated on this model checkers.

For this purpose, we implemented two different model checking backends for our framework. One for the symbolic model checker SAL and another one for the ANSI-C model checker CBMC [13]. In case of the SAL backend the C-code is converted to semantics-equivalent code of the model checker input language.

Depending on the selected coverage criteria, different models are generated. Currently, branch coverage (C1) and path coverage (C2) for program segments is supported. The detailed model generation mechanism is described in [11]. The generated test data are stored in a XML repository.

5.4 Communication and Test Data Representation

As already described, differences in the platform-specific semantics have to be considered when transforming the programs between platforms. We decided to choose the host platform as base platform.

Test data are stored platform independent in XML repositories, the respective values are represented as logical integer numbers. When test data is exchanged across platform boundaries, the corresponding transformations are carried out completely automatically by the framework, which has to be aware of the host and the target platform properties. Important properties are the data size and layout which may be either big or little endian.

An important issue is, where the data conversion from XML to platform specific binary values takes place. Since the targets are likely limited in resources the conversions are performed on the host. The host automatically generates stubs for the target that receive the platform specific test data via a RS232 or USB link, writes the data to the variables (local variables are also placed within the stub) and executes the target code. The stub traces the program execution using callback functions and writes the results back to the host where they are checked for correctness.

6 Experiments

In this section we describe the setup of our experiments. We perform cross-platform testing for selected applications.

In order to show the cross-platform testing mechanism, we decided to perform basic block coverage cross-platform testing. In this experiment, the generated and verified test data is used for structural tests on a Motorola HCS12 evaluation board with its respective build environment (using the commercial C compiler).

As benchmark sample we used three C-code files (like before each containing one function that is subject to our test). Function F5 is a simple demo function, function F11 and F12 contain industrial code.

We summarized the results in Figure 4. When applying structural basic block coverage test to the more complex applications we got very surprising results.

	Characteristics					Testing Time				Result		
Function Name	Industrial code	Lines of code	# Basic blocks	# Basic blocks reachable	Complexity (# paths)	Testdata generation (m)	Target verification (m)	Total (m)	Time / basic block (m)	# Basic blocks reached correctly	Correctness %	Error reason
Function F5	no	46	30	30	72	1,2	2,8	4,0	0,13	30	100%	None
Function F11	yes	274	54	54	97	4,5	5,3	9,8	0,18	54	100%	None
Function F12	yes	1150	171	165	1,90E+11	124,7	21,5	146,2	0,85	159	96%	Compiler defect

Fig. 4. Experimental Results

First, there is some difference between the number of basic blocks and the number of reachable basic blocks. With basic blocks that are proved by the model checker being not reachable there may seem to be something wrong. However, the reason for this is that code generators assign fixed values to some parameters. Thus, there is some code that cannot be executed at runtime. Second, when cross-platform testing is applied some basic blocks could not be reached correctly as on the reference platform where test data have been already verified. The reason for this problem turned out to be some compiler defect. We found 3 other industrial applications where similar compiler errors occurred.

Due to the fact that the code of function F12 (respectively the other industrial applications) is proprietary and it is hard to see the defect, we constructed smaller examples D1 and D2 showing these problems.

Defect D1 is illustrated in Figure 1.1. The calculation of the variable Aux_S32 should result in 0x00005560 on both platforms. However, on the HCS12 platform it yields 0xFFFF8000 (using version 4.6a of the compiler). When the statement is rewritten in an alternative form as shown at lines 11 and 12 in Figure 1.1, the computed result is correct. After we reported this bug to the compiler manufacturer, the bug was committed and few weeks later a version was provided having this bug fixed.

```
 1  typedef signed short int T_INT16;
 2  typedef signed long int T_INT32;
 3
 4  T_INT32 in_32;
 5
 6  void test(void) {
 7    T_INT32 Aux_S32;
 8
 9    Aux_S32 = (T_INT16) ((T_INT32)in_32 / (T_INT16)3);
10
11    // Aux_S32 = ((T_INT32)in_32 / (T_INT16)3);
12    // Aux_S32 = (T_INT16) Aux_S32;
13  }
14
15  void my_frame(void) {
16    in_32 = 0x8000001E; // -2147483618
17    test();
18  }
```

Listing 1.1. Code for defect D1.

```
 1  typedef signed short T_INT16;
 2  typedef signed long T_INT32;
 3
 4  volatile T_INT32 t;
 5  volatile T_INT32 t1;
 6  volatile T_INT32 c1 ;
 7
 8  void test2(void) {
 9    t1 = (T_INT32) ((T_INT16) ((T_INT32)c1 /
10                              ((T_INT16) 0x4000)));
11
12    t =  (T_INT32) ((T_INT16) ((T_INT32)0x8001));
13  }
14
15  void ingo_test(void) {
16    c1 = 0x20007FFF;
17    test2();
18  }
```

Listing 1.2. Code for defect D2.

However, after installing the new compiler version and re-running the tests, we found another problem referred to as D2. The simplified example code is depicted in Figure 1.2. In the correct case the calculation of t1 and t yields 0xFFFF8001. However, on the HCS12 platform t1 yields 0x00008001 and t equals 0xFFFF8001. In line 9 the division results in the intermediary result 0x00008001. It seems that in this expression the cast to INT16 is simply omitted by the compiler (although all optimizations are disabled). A few weeks after the second bug has been submitted to the manufacturer, a corrected version has been delivered where no further faults have been detected.

7 Summary and Conclusion

We introduced the notion of cross-platform verification for embedded systems. Based on a target source code, a semantics-equivalent model is generated for a host computer. By model checking, respective test data are generated that are self-checked on the host. As practice has shown, this self-check is very useful to verify whether the test data (and the models behind) are correct (self assessment mechanism). Then, these data are used to exercise the actual execution platform and the results are compared with those on the host computer (reference platform testing). Due to our experience, it is highly important to include the whole execution platform involving the compiler, linker, target loader, and the hardware itself to exhaustively test embedded systems (end-to-end testing).

The model generation mechanism that is used for test data generation can be also used for formal verification. This has the advantage that the confidence into the models is increased as the same mechanisms are used. From our experience, the joint use of formal models and test data obtained from them seems promising.

In the future, we plan to add new platforms and so – by platform diversification – more faults can be detected in execution platforms. Further, it is easily possible to extend the framework to support other coverage criteria.

In our experiments, we have shown that by using this method even compiler faults can be detected.

7.1 Acknowledgments

Our tool uses the CBMC model checker developed by Daniel Kroening and Edmund Clarke at Carnegie Mellon University. Further, we use the SAL model checker developed by Leonardo Demura at SRI Labs.

References

1. Laprie, J.C., Randell, B.: Basic concepts and taxonomy of dependable and secure computing. IEEE Trans. Dependable Secur. Comput. **1**(1) (2004) 11–33 Fellow-Algirdas Avizienis and Senior Member-Carl Landwehr.
2. Lackner, K.: Bordelektronik im Zwielicht Economy Austria - Printausgabe, 25.08.2006.
3. Commission, I.E.: Functional safety of electrical / electronic / programmable electronic safety-related systems. IEC standard 61508 (1998)
4. : Software considerations in airborne systems and equipment certification. RTCA/DO-178B (1992)
5. Beyer, D., Chlipala, A.J., Henzinger, T., Jhala, R., Majumdar, R.: Generating tests from counterexamples. In: Proc. 26th International Conference on Software Engineering (ICSE), Edinburgh, Scotland, UK, IEEE Computer Society (2004) 326–335
6. Rayadurgam, S., Heimdahl, M.P.E.: Coverage based test-case generation using model checkers. In: Proc. 8th IEEE International Conference and Workshop on the Engineering of Computer Based Systems (ECBS ’01), Washington DC, USA (Apr. 2001)
7. Ammann, P., Black, P.E., Majurski, W.: Using model checking to generate tests from specifications. In: Proc. 2nd IEEE International Conference on Formal Engineering Methods, Brisbane, Queensland, Australia, IEEE Computer Society (Dec. 1998) 46–54
8. Hamon, G., deMoura, L., Rushby, J.: Generating efficient test sets with a model checker. In: 2nd International Conference on Software Engineering and Formal Methods, Beijing, China, IEEE Computer Society (Sep. 2004) 261–270
9. Clarke, E., Grumberg, O., Peled, D.: Model Checking. MIT Press (1999)
10. Koenig, A.: C Traps and Pitfalls. Addison-Wesley, Reading, MA (1988) ISBN: 0-201-17928-8.
11. Wenzel, I.: Measurement-Based Timing Analysis of Superscalar Processors. PhD thesis, Technische Universität Wien, Institut für Technische Informatik, Treitlstr. 3/3/182-1, 1040 Vienna, Austria (2006)
12. Wenzel, I., Rieder, B., Kirner, R., Puschner, P.: Automatic timing model generation by CFG partitioning and model checking. In: Design, Automation and Test in Europe, 2005. Proceedings. (2005) 606–611
13. Clarke, E., Kroening, D.: Hardware verification using ANSI-C programs as a reference. In: Proceedings of ASP-DAC 2003, IEEE Computer Society Press (January 2003) 308–311