Development of a Framework for Automated Systematic Testing of Safety-Critical Embedded Systems *

Susanne Kandl[1], Raimund Kirner[1], Peter Puschner[1]

[1]Institut für Technische Informatik
Technische Universität Wien
A-1040 Wien, Austria
{susanne,raimund,peter}@vmars.tuwien.ac.at

Abstract — *In this paper we introduce the development of a framework for testing safety-critical embedded systems based on the concepts of model-based testing. In model-based testing the test cases are derived from a model of the system under test. In our approach the model is an automaton model that is automatically extracted from the C-source code of the system under test. Beside random test data generation the test case generation uses formal methods, in detail model checking techniques. To find appropriate test cases we use the requirements defined in the system specification. To cover further execution paths we developed an additional, to our best knowledge, novel method based on special structural coverage criteria. We present preliminary results on the model extraction using a concrete industrial case study from the automotive domain.*

1 Introduction

Our work was motivated by the increasing occurrence of failures in embedded systems in the automotive domain causing worldwide recalls of cars. Due to the increasing capacity of processors, the functionality and thus the complexity of the embedded system software is growing. Because embedded systems are often used in safety-critical applications, where failures can cause crucial damage to people, creatures or the environment, testing is extremely important to ensure the quality and reliability of the embedded systems. Sophisticated and mature testing techniques using formal methods are of common use in areas of high safety-criticality, for instance transportation, avionics, or aerospace [1]. In other application domains formal methods for testing are only starting to gain ground.

*This work has been supported by the FIT-IT research project "Systematic test case generation for safety-critical distributed embedded real time systems with different safety integrity levels (TeDES)"; the project is carried out in cooperation with TU-Graz, Magna Steyr, and TTTech.

Using formal methods within our testing framework has the big advantage that the complete system behavior can be described in a formal, thus unambiguous way. The limits of the applicability of formal methods are given by the difficulty to use them for software systems of high complexity.

The testing framework we are developing consists of two main parts: First we build the automaton model of the system under test. This is realized by automatically parsing and analyzing the source code. Once we have built the model, we can use it as a basis for our test case generation methods.

Test data is generated in following ways: We generate test data randomly to cover execution traces of the system fast and easily. Test cases for testing a specific system property are found by means of model checking techniques. Within our framework we use model checkers to find concrete execution paths of the system that correspond to requirements defined in the system specification. The input and output values of these executions paths yield concrete test cases.

The article is organized as follows: In Section 2 we characterize our testing method. In Section 3 we introduce the general concepts of model-based testing and the scenario for our testing framework. In Section 4 we describe in detail how we build the automaton model of the system automatically by extracting it from the C-source code. Section 5 deals with the test case generation methods using formal methods and the applicability of model checking techniques for the test case derivation. In Section 6 we discuss the preliminary results of our approach by evaluating it on a concrete case study. Finally we give an overview on related work and conclude with a summary and the plans for future work.

2 Testing Safety-Critical Applications

Testing is a process centered around the goal of finding defects in a system for debugging or acceptance reasons [2]. The ultimate goal is to prove the absence of failures in the system behavior. In general it is not possible to test a system exhaustively, that means to execute all possible system configurations and thus find and fix all defects in a reasonable time. For functional testing it is essential to provide *suitable* test cases to the system under test. The execution of these test cases should prove, whether a system requirement defined in the specification is valid or violated by the implementation.

We characterize our framework as *dynamic, gray box, systematic, and automated testing*:

Dynamic Testing: In dynamic testing the concrete test cases are executed against the code and thus the proper functioning of operations can be tested on different levels: *SIL* (software in the loop: test cases run against the object code), *PIL* (processor in the loop: test cases are applied to the system on the embedded processor) and *HIL* (hardware in the loop: testing is processed in the target environment).

Gray Box Testing: In general it is difficult to distinguish exactly between white box and black box testing. For example, an ambiguity occurs when the module structure of a system is known, but not the code of each module. We want to refer to this kind of testing with the term *gray box testing*.

Systematic Testing: Systematic testing means that the test cases are generated according to specific test purposes. We create test cases based on the formalized properties to be tested. For details please refer to Section 5.

Automated Testing: In automated testing both the generation of the test cases and the evaluation of the test results are done in an automated way.

2.1 Testing Safety-Critical Systems

The requisites for the quality of safety-critical systems are defined in several standards, for instance DO-178B [3] or IEC 61508 [4].

The developed testing framework is oriented on the IEC 61508 standard. IEC 61508 is a standard for functional safety of electrical/electronic/programmable electronic safety-related systems. It provides a list of guidelines for the testing process and multiple testing methods. Some of the relevant issues for our framework are:

- *Structure-based Testing*: Structure-based testing aims to apply tests which exercise certain subsets of the program structure [4]. The standard defines a few coverage metrics, for instance: statement coverage, branch coverage, or MC/DC (modified condition decision coverage).
- *Modeling - Finite State Machines*: As the standard defines, finite state machines, resp. state transition diagrams are means to model, specify or implement the control structure of a system [4].
- *Formal Methods*: Formal methods can be used to express a specification unambiguously and consistently, so that mistakes and omissions can be detected. The resulting description takes a mathematical form and can be subjected to mathematical analysis to detect various classes of inconsistency or incorrectness [4].

The testing framework we are developing aims to meet concrete coverage criteria for the system under test. As described below, we will use finite state machines to model the system behavior and we will use formal methods to prove the integrity of the system and to direct our test case generator to derive test cases.

3 Model-Based Testing

Referring to the *Encyclopedia on Software Engineering, 2001* [5] the term *model-based testing* characterizes an approach that bases common testing tasks, such as test case generation and test result evaluation, on a *model* of the application under test (see Figure 1).

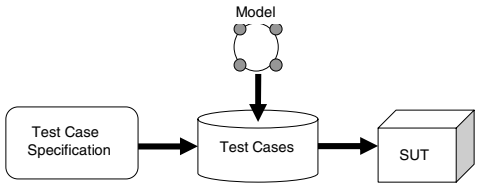

Figure 1: Principle of Model-Based Testing

Several kinds of models can serve as a basis for the generation of test cases. For instance, test cases can be created based on UML-models of the system. Furthermore, lots of other modeling languages for the formal specification exist, for example, synchronous

languages (e.g., Esterel or Lustre), Petri nets, transition systems and automata-based formalism (e.g., finite state machines). For an overview please refer to [6].

The model for the test case generation can be built before the development of the software, as a supplement parallel to the software development process or separated from the software development process. Hence, different scenarios for generating test cases are possible, as described by Pretschner et al. [7]:

1. A common model can be used for both, code generation and test case generation. This scenario is used especially with respect to the model-centric development paradigm, like MDA (model-driven architecture) [8].
2. A second scenario is concerned with extracting models from an existing system. Once the system is built (e.g., hand coded), one creates a model manually or automatically, and this model is then used for test case generation.
3. A further approach consists of manually building the model for test case generation, while the system is again built on top of a different specification.
4. The last noteworthy scenario involves two distinct models, one for test case generation, and one for code generation.

An essential issue in model-based testing is the *automatic verdict* for the test cases. When the same model is used for code generation and test case derivation, the test cases can also be run against the application. But the following problem occurs: The verdict, i.e. the expected output for the test data, is deduced from a model that maps completely to the code. The resulting test cases executed against the system will all pass the test procedure, because possible erroneous behavior from the code is directly reproduced in the model. In this scenario it is not possible to produce test cases that prove if the implementation is valid against the model. Therefore most model-based testing scenarios need an additional manually built model. This model is assumed to be valid against the specification. The test cases derived from this model can be run on the object code, thus errors in the implementation can be detected [9].

3.1 Our Approach to Model-Based Testing

In our framework we derive the test cases from an automaton model that is extracted from the source code. The test cases are generated from the model and the requirements of the specification (see Figure 2). The system behavior is defined in the specification and the requirements. SRS means *safety requirements specification*, which is a subset of the requirements, focusing on the functional safety of the system. Based on the specification, an implementation model is built (in our case study with Matlab Simulink) and the source code for the system is automatically generated with a code generator (TargetLink from dSpace[1]).

For the testing framework the requirements are formalized, i.e. the requirements defined in the system specification are translated into temporal logic formulas. These formulas are the main part of the *test case specification*. The test case specification also contains information about coverage metrics, for instance, aimed coverage criteria, like branch coverage or MC/DC.

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

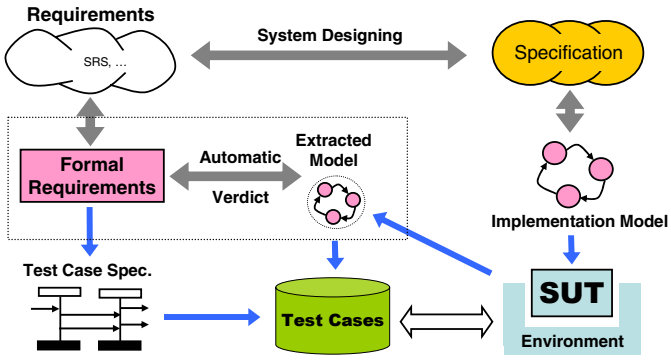

Figure 2: Our Approach of Model-Based Testing

The *extracted model*, which is automatically extracted from the source code, and the test case specification serve as a basis in our framework for the generation of test cases. The model and the formalized requirements are provided to the model checker. The idea is to find traces in the automaton model of the system that represent execution paths corresponding to specific requirements. The input and output values of the resulting traces are used as test cases. These test cases are then applied to the system under test. See also Section 5.

In our approach the *verdict*, i.e. the expected output for the test data, is given by the requirements. The model is assumed to be valid against the specification (see also Section 5). The resulting test cases running on the target platform prove, whether the system behaves consistent to the model and thus consistent to the specification.

In the following, we describe the model extraction process and the formal test case generation methods.

4 Model Extraction

Extracting the model from the source code is extensive but can be easily automated. Our model extraction framework is based on a framework that has been developed for measurement-based worst-case execution time analysis within the project *Model-Based Development of distributed Embedded Control Systems* (MoDECS) [10, 11, 12]. The model extraction is done in the following steps [10]:

1. First the C-code is parsed and the *syntax tree* is generated by static analysis (see Figure 3). Depending on the complexity of the program, optional single blocks (modules of the system) can be identified to ease the analysis.
2. The resulting syntax tree is used to generate the *control flow graph* (see Figure 4) and the *decision tree* (see Figure 5) for further analysis within the final testing process (e.g., path coverage analysis).
3. The syntax tree is used to generate the *automaton model* of the system. This is realized by sequentially processing the syntax tree and interpreting the semantics of the single statements.
4. The description of the automaton is given in an automata language (for instance, the modeling language of the SAL or NuSMV model checker).

We give a simple example to demonstrate our model extraction: We start with a small C-program. In this program the function $test$ takes two arguments x and a and contains two composed if-else conditions. Depending on the current values of x and a, the variable x is assigned with the values 10, 20 or 100. The resulting syntax tree consists of approximately 100 nodes, of which we have to deal with 16 statements for the generation of the automaton model. Figure 3, 4 and 5 show the syntax tree, the control flow graph, and the relating decision tree for the example. The analysis of the syntax tree yields the automatically generated NuSMV model.

Example:

C source code:

```
int test (int x, int a)
{
    if (x == 1) {
      x=10;
    } else if (a == 2) {
      x=20;
    } else {
      x=100;
   }
}
```

NuSMV Model:

```
MODULE main
VAR
    sequence_nr :  0..65535;
    v0_x :  0..255;
    v1_a :  0..255;
ASSIGN
    init(sequence_nr):= 16;
    next(sequence_nr):=
      case
        sequence_nr= 2:  1;
        sequence_nr= 5:  1;
        sequence_nr= 8:  1;
        sequence_nr= 12 & (v1_a=2) :  5;
        sequence_nr= 12 & !(v1_a=2) :  8;
        sequence_nr= 16 & (v0_x=1) :  2;
        sequence_nr= 16 & !(v0_x=1) :  12;
      esac;
    next(v0_x):=
      case
        sequence_nr= 2:  10;
        sequence_nr= 5:  20;
        sequence_nr= 8:  100;
      esac;
```

The additional variable `sequence_nr` in the NuSMV model stems from the static analysis and represents in principle the program counter. The NuSMV model can be directly processed by the NuSMV model checker. In the following section we describe how we generate the test cases from this NuSMV model.

5 Formal Test Case Generation

For the test case generation, amongst other methods, we use model checking techniques. We work with the model checkers SAL[2] and NuSMV[3]. Beside its original purpose for formal verification of systems, model checking has become an applicable tool for test case generation. The main purpose of model checking is to verify a formal property given as a logic formula (in general formalized in some kind of temporal logic, e.g., linear temporal logic *LTL* or computational tree logic *CTL*) on a system model. In the case of that the formal property is invalid on a given model, a model checker typically provides a *counterexample*, which describes a concrete instantiation of variable values and a path

[2] `http://sal.csl.sri.com/`
[3] `http://nusmv.irst.itc.it/`

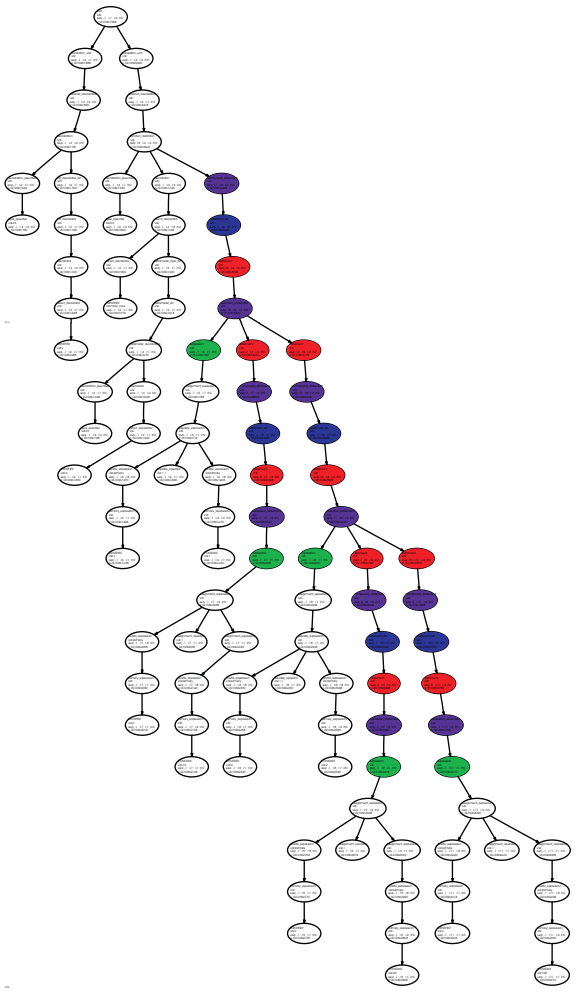

Figure 3: Syntax Tree

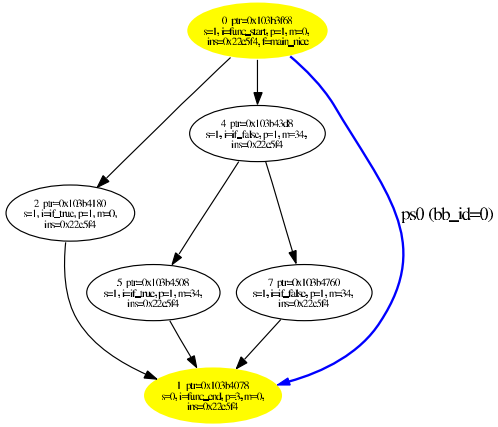

Figure 4: Control Flow Graph

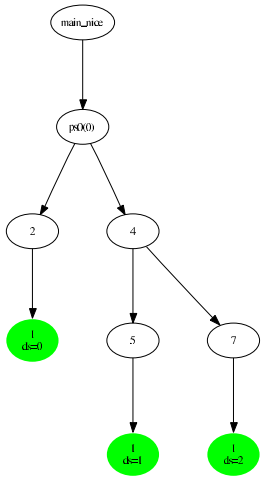

Figure 5: Decision Tree

for which the property is violated. This feature of a model checker can also be used to generate test data in a formal and systematic way [13, 1, 14, 15].

The goal of our testing framework is to test, whether the SUT conforms to the system requirements. Thus, we calculate test cases where the verdict is based on formal assertions derived from the system requirements. So besides building the model we formalize the requirements from the system specification in temporal logic.

Several approaches of requirement-based test case generation using model checkers exist. For example, mutation testing is an approach, where test cases are derived from a system, where either the model or the logic formula to be tested are modified [16]. In the following we sketch the test case generation methods that we will use in our testing framework. The first method “property-based test case generation” is similar to the use of so-called “never-claims” as described by Engels et al. [15]. The second method “relevance-based test case generation” is, to our best knowledge, a novel technique we developed as a complementary test case generation method.

5.1 Property-Based Test Case Generation

The basic idea of this technique is to find test cases for locations in the model where the property f is valid. To find such test cases with the model checker, we state that f is

not feasible within the system model. The negated formula $\neg f$ is provided to the model checker. The model checker searches the state space of the model to prove if this formula is valid or not. In the case of a violation of the assertion $\neg f$ the model checker provides a path on which f is *true* (a counterexample). This path is one instance of an execution trace of the program, where the property f holds and can thus be used as a test case to test the requirement f (see Figure 6). The model checker delivers a counterexample, if the property we want to test is valid.

For example, in the model of the example in Section 4 the assertion (a) is valid. But the assertion (b) is violated, thus, a counterexample is produced by the model checker.

```
PSLSPEC AG !(v1_a=2 & v0_x=100);        (a)
PSLSPEC AG (v0_x=1 & v0_x=10);          (b)
```

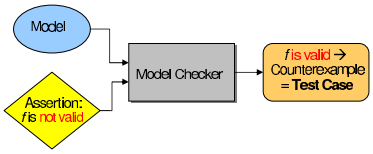

Figure 6: Test Case Generation with Counterexamples

As a by-product we are also performing formal verification. We only get counterexamples that we can use as test cases, if the property is valid in the model. In the case of a violation of the property, we know that the property is not valid in the model. Because the model can be directly mapped to the source code of the system, we know in the case of a violation that a requirement from the specification is violated by the implementation. This points to the fact that the failure in the implementation has to be corrected.

In general, model checkers produce only one counterexample, where a property is violated. When we consider only the requirements from the specification this yields only a few execution paths of the system. One possibility to get more paths is to modify the model checker to produce more than one counterexample. However, this modification could yield a huge number of paths, many of them testing the same property. Thus, instead of using this modification, we developed a complementary test case generation method described in Section 5.2.

5.2 Relevance-Based Test Case Generation

When using the *property-based* test case generation described above, it can happen that a property f is valid in many different locations of the model. In such a case it is desirable to generate test cases targeted to those locations where the property is most relevant, e.g., variables occurring within the formula of a property f influence directly the generation of the output of the model.

In *relevance-based* test case generation, the principle of how the model checker generates test data is similar to the above method. We provide an assertion g to the model checker and the model checker searches for counterexamples that violate g. The only difference between the above method and this approach is the meaning of g. With this approach, g is not simply the negation of f, the property we want to verify by testing. Instead, g formally states that a certain location x of the model cannot be reached. Thus, the counterexample generated by the model checker provides input data to reach the location x.

x is chosen as one of these locations where variables of f can directly influence the generation of output. The interesting instances of location x are calculated by a prior static program analysis phase. For example, if a variable of f is used at a certain location x to decide, whether the output value should be incremented, then x is an interesting code location for a test. Based on g, the model checker can calculate those test data for which x will be reached. This approach provides an effective search for test cases, in the sense that it focuses only on code locations where the output is influenced by variables from the property to be tested.

To illustrate how such an assertion g can be formulated, let us assume that we have found the assignment $x = 20$ in the source code example of Section 4 to be an interesting location to be tested (in reality, this code does not generate an output). Since this location corresponds to the state "`sequence_nr=5`" in the model, we can use the assertion (c) to get a path to this location:

```
PSLSPEC G (sequence_nr !=5);                                    (c)
```

Again, the verdict of the test cases in relevance-based test case generation is derived from the original property f to be tested. For each test execution the values of all variables occurring in f are monitored. The test result of a test case is positive if the property f is fulfilled. Properties that represent invariants of the model can be directly used as test verdicts.

6 Case Study: Preliminary Results and Evaluation

Our case study is a control loop taken from an embedded system application from the automotive domain. The system was built with Matlab Simulink and the code was generated by a code generator (TargetLink from dSpace[4]). The overall system consists of four modules with about 40 variables, 16 of them of boolean data type, the remainder are integer. 10 of the variables are input variables and 10 of them represent output values.

For processing the NuSMV automaton model we have to deal with a few challenging aspects:

- *Arithmetic operations*: In our case study one module of the system is calculating the values for an actuator based on input values from sensors. These values are calculated by a complex arithmetic expression. This causes problems within the model checker, because the model checker has to generate all possible values, the resulting BDD (binary decision tree) is quite big.
- *Complexity*: The set of scalar variables and the operations mentioned above applied to these variables causes a big state space. Abstraction (e.g., data type reduction) can ease the analysis and speed up the test case generation process, but we run the risk of loosing relevant test data. Interesting abstraction mechanisms are: data type abstraction, decomposition of the system and counterexample refinement.
- *C-specific semantics on data ranges*: The model checker builds the complete BDD for all possible data values. Data domain overflow is not treated by the model checker. The semantics of handling data values out of range from C has to be integrated into building the correct NuSMV model.

[4] http://www.dspace.com

- *Local preconditions*: NuSMV determines the validity of an assignment only locally. That means that assertions about the assignment of a variable in the model are not considered at a single transition. This entails that we have to add special preconditions additionally to assertions stated in the C program.
- *Type casting*: Type casting is standard in C programs. In NuSMV variables are declared once before the model is built. Therefore, type casts occurring in the C code cause difficulties within the NuSMV model.

We are able to analyze the source code of our case study and build the control flow graph; it yields nearly 8000 paths. We are on the way to complete the implementation of the model extraction with all features that yields an automaton model representing the overall correct semantics of the C code and can thus be directly processed by the model checker for the test case generation.

Besides the test case generation methods described in Section 5 we use random test data generation to reduce the workload of the model checker.

7 Related Work

In Broy et al. [17] various aspects of model-based testing are described in detail. One chapter of it, written by Pretschner et al. [7], is about *Methodological Issues in Model-Based Testing*, which inspired our research on model-based testing. Several existing testing tools are also based on the concepts of model-based testing, for an overview see [18]. Mirko discusses model-based testing of embedded systems on examples from the automotive domain [9].

A general introduction to model checking can be found in Clarke et al. [19]. The ideas of using model checking techniques also for testing aim back to the mid '90s, for instance [20]. A survey on formal testing techniques can be found in [21]. The test case generation techniques are described in detail in [22]. Since then many works are concerned with using model checking techniques for testing purposes by means of producing counterexamples, for instance Beyer et al. [23]. Amman et al. use model checking to generate tests from specifications [24, 1]. Also Gargantini et al. [25] generate tests from requirement specification. Grabowski discusses issues concerning specification-based testing of especially real-time distributed systems [26]. Engels et al. [15] are using "never-claims" for test case generation.

The way of how the model is build depends on how the test cases are finally generated. We know about model checkers that can process source code directly, for instance BLAST[5], described in Henzinger et al. [27]. CMBC[6] is a bounded model checker for ANSI-C programs, described in [28]. The model checker SPIN[7] provides a tool that translates C-code into the input language of SPIN *PROMELA*, see [29]. Also MUTT[8] developed at Microsoft Research is a tool for automated test generation, but mainly for unit testing, see [30]. Extracting the automaton model from the source code is based on works of Wenzel et al. [10]. Schroder et al. [31] discusses a few modeling aspects for the

[5] `http://embedded.eecs.berkeley.edu/blast/`
[6] `http://www.cs.cmu.edu/~modelcheck/cbmc/`
[7] `http://spinroot.com`
[8] `http://research.microsoft.com/projects/mutt/`

automated test case generation.

Techniques for applying abstraction to models are described in [32, 33, 34, 35]. Clarke et al. are working on "counterexample-guided refinement" [36]. Ball et al. deal with abstraction especially in relation to the falsification of branching-time and linear-time temporal properties [35].

The introduced methods have been applied to a few case studies: Chandra et al. [37] used software model checking for an industrial case study. The inhouse card study is a well exercised case study [38]. An evaluation of model-based testing is given by Pretschner [39] or Paradkar [40], who studied the fault detection effectiveness of model-based testing. Testing purposes especially concerned to the automotive domain are discussed amongst others in Ranville and Black [41] or Mirko [9].

8 Summary and Conclusion

In this paper we introduced the development of a testing framework that is based on model-based testing. The model is automatically extracted from the source code of the system under test. The test cases are derived from this automaton model by producing counterexamples by means of model checking. Besides using the "never-claim" properties directly obtained from the specification (*property-based* test case generation), we presented a novel test case generation method (*relevance-based* test case generation) that focuses on testing those code locations where variables from the property to be tested influence the output generation.

Besides describing the conceptual aspects of our testing framework, we demonstrated the functionality of our implementation for the model extraction on a small sample code. Furthermore, we showed technical details of extracting the automaton model from the C source code of an industrial case study.

In our future work we plan to integrate some abstraction mechanisms to reduce the state space of the automaton model of study to improve the performance of the introduced test case generation methods.

References

[1] Paul Ammann, Paul E. Black, and William Majurski. Using model checking to generate tests from specifications. In *ICFEM*, page 46, 1998.

[2] Bart Broekman and Edwin Notenboom. *Testing Embedded Software*. Addison-Weseley, 2003.

[3] Software considerations in airborne systems and equipment certification. RTCA/DO-178B, 1992.

[4] IEC. IEC 61508 functional safety of electrical/electronic/programmable electronic safety-related systems. Technical report, IEC, 1998.

[5] Ibrahim K. El-Far and James A.Whittaker. Model-based software testing. In *Encyclopedia on Software Engineering (edited by J.J. Marciniak)*. Wiley, 2001.

[6] Aditya Agrawal. *Model Based Software Engineering, Graph Grammars and Graph Transformations*. Area paper, Vanderbilt University, March 2004.

[7] Alexander Pretschner and Jan Phillips. *Model-Based Testing of Reactive Systems, LNCS 3472 (Eds.: M.Broy et al.)*, Chapter 10 Methodological Issues in Model-Based Testing, pages 281–291. Springer-Verlag Berlin Heidelberg, 2005.

[8] Stephen J.Mellor, Kendall Scott, Axel Uhl, and Dirk Weise. *MDA Distilled*. Addison Weseley Professional, 2004.

[9] Mirko Conrad. *Modell-basierter Test eingebetteter Software im Automobil: Auswahl und Beschreibung von Testszenarien*. DUV, 2004.

[10] I. Wenzel, B. Rieder, R. Kirner, and P. Puschner. Automatic timing model generation by CFG partitioning and model checking. In *DATE '05: Proceedings of the conference on Design, Automation and Test in Europe*, pages 606–611. IEEE, 2005.

[11] Raimund Kirner, Peter Puschner, and Ingomar Wenzel. Measurement-based worst-case execution time analysis using automatic test-data generation. In *Proc. 4th International Workshop on Worst-Case Execution Time Analysis*, pages 67–70, Catania, Italy, June 2004.

[12] Ingomar Wenzel, Bernhard Rieder, Raimund Kirner, and Peter Puschner. Measurement-based worst-case execution time analysis. In *Proc. 3rd IEEE Workshop on Software Technologies for Future Embedded and Ubiquitous Systems (SEUS'05)*, pages 7–10, Seattle, Washington, May 2005.

[13] Paul Ammann and Paul E. Black. Model checkers in software testing. Online: http://citeseer.ist.psu.edu/ammann02model.html.

[14] D. Beyer, A.J. Chlipala, T. Henzinger, R. Jhala, and R. Majumdar. Generating tests from counterexamples. In *ICSE '04: Proceedings of the 26th International Conference on Software Engineering*, pages 326–335, 2004.

[15] André Engels, Loe M.G. Feijs, and Sjouke Mauw. Test generation for intelligent networks using model checking. In E. Brinksma, editor, *Proc. 3rd International Workshop on Tools and Algorithms for the Construction and Analysis of Systems*, number 1217 in LNCS, pages 384–398. Springer, 1997.

[16] V. Okun, P. Black, and Y. Yesha. Testing with model checkers: Insuring fault visibility, 2003.

[17] M.Broy, B.Jonsson, J.-P.Katoen, M.Leucker, and A.Pretschner, editors.

[18] Axel Belifante, Lars Frantzen, and Christian Schallhart. *Model-Based Testing of Reactive Systems, LNCS 3472 (Eds.: M.Broy et al.)*, Chapter 14 Tools for Test Case Generation, pages 391–438. Springer-Verlag Berlin Heidelberg, 2005.

[19] E.M. Clarke, O. Grumberg, and D.A. Peled. *Model Checking*. The MIT Press, 2000.

[20] J. Callahan, F. Schneider, and S. Easterbrook. Automated software testing using model-checking. In *Proceedings 1996 SPIN Workshop*, August 1996.

[21] Jan Tretmans. Testing techniques, 2002. Online: http://www.cs.auc.dk/~kgl/TOV04/tretmans-notes.pdf.

[22] Levi Lucio and Marko Samer. *Model-Based Testing of Reactive Systems, LNCS 3472 (Eds.: M.Broy et al.)*, Chapter 12 Technology of Test Case Generation, pages 323–354. Springer-Verlag Berlin Heidelberg, 2005.

[23] D. Beyer, A.J. Chlipala, and R. Majumdar. Generating tests from counterexamples. In *ICSE '04: Proceedings of the 26th International Conference on Software Engineering*, pages 326–335, Washington, DC, USA, 2004. IEEE Computer Society.

[24] Paul Ammann and Paul E. Black. Model checkers in software testing. Online: http://citeseer.ist.psu.edu/ammann02model.html.

[25] Angelo Gargantini and Constance Heitmeyer. Using model checking to generate tests from requirements specifications. In *ESEC/FSE-7: Proceedings of the 7th European software engineering conference*, pages 146–162, London, UK, 1999. Springer-Verlag.

[26] Jens Volker Grabowski. *Specification-based Testing of Real-Time Distributed Systems: Languages, Tools an Applications*. Logos Verlag Berlin, 2003.

[27] T. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Software verification with Blast, 2003.

[28] Edmund Clarke, Daniel Kroening, and Flavio Lerda. A tool for checking ANSI-C programs. In Kurt Jensen and Andreas Podelski, editors, *Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2004)*, volume 2988 of *Lecture Notes in Computer Science*, pages 168–176. Springer, 2004.

[29] Gerard J. Holzmann and Margaret H. Smith. Software model checking: extracting verification models from source code. *Software Testing, Verification & Reliability*, 11(2):65–79, 2001.

[30] Thomas Ball. A theory of predicate-complete test coverage and generation. In *FMCO2004*, 2004.

[31] Patrick J. Schroder, Eok Kim, Jerry Arshem, and Pankaj Bolaki. Combining behavior and data modeling in automated test case generation. In *QSIC'03*. ACM, 2005.

[32] E. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement for symbolic model checking. *J. ACM*, 50(5):752–794, 2003.

[33] John Rushby. Ubiquitous abstraction: A new approach for mechanized formal verification. In *Second International Conference on Formal Engineering Methods (ICFEM '98)*. IEEE, 1998.

[34] T.A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Lazy abstraction. In *Proc.* 29^{th} *Symp. on Principles of Programming Languages*, pages 58–70. ACM Press, 2002.

[35] Thomas Ball, Orna Kupferman, and Greta Yorsh. Abstraction for falsification. In *Proc. 17th International Conference on Computer Aided Verification*, pages 67–81, July 2005.

[36] E. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement. In *Computer-Aided Verification*, volume 1855 of *LNCS*, pages 154–169. Springer, 2000.

[37] Satish Chandra, Patrice Godefroid, and Christopher Palm. Software model checking in practice: an industrial case study. In *ICSE '02*, pages 431–441, New York, NY, USA, 2002. ACM Press.

[38] A.Pretschner, O.Slotosch, H.Lötzbeyer, E.Aiglstorfer, and S.Kriebel. Model based testing for real: The inhouse card case study. In *Proc. 6th Intnl. workshop on Formal Methods for Industrial Critical Systems*, pages 79–94, 7 2001.

[39] Pretschner et al. One evaluation of model-based testing and its automation. In *Proceedings of the 27th international conference on Software engineering*. IEEE, 2003.

[40] Amit Paradkar. Case studies on fault detection effectiveness of model based test generation techniques. In *Advances in Model-Based Software Testing(A-MOST'05)*. ACM, 2005.

[41] Scott Ranville and Paul E. Black. Automated testing requirements - automotive perspective. In *The Second International Workshop on Automated Program Analysis, Testing and Verification*, May 2001.