Towards Fault-based Generation of Test Cases for Dependable Embedded Software

Wolfgang Herzner, Rupert Schlick; AIT Austrian Institute of Technology
Harald Brandl, Technical University Graz
Johannes Wiessalla, Ford Forschungszentrum Aachen

***Abstract**. In the European project MOGENTES[1] methods for model-based generation of efficient test cases are developed. A special focus is laid on test cases, which not only allow for assessing the fulfillment of requirements, but in particular looking for potential faults – or prove their absence. This is achieved by mutation-based testing: an original model is modified to simulate faults, and then test cases are searched which are able to distinguish between original and mutated model. This paper gives an overview of MOGENTES, presents an application example from the automotive domain, and summarizes the results from this example.*

***Keywords.** fulfillment of functional (safety) requirements, model-based test case generation (MBTCG), fault-models, mutation-based testing, MOGENTES*

1 Introduction

Embedded systems not only become more and more an integral part of our life, for instance in cars, trains, airplanes, health care or industrial production, but also their complexity rapidly grows due to continuous integration of existing systems and addition of new functions and responsibilities, fostered by strong increase of computing power. As a consequence, the confirmation of their correct operation and reliability – in particular of their software – becomes an increasing challenge in equal measure. While formal verification of software is successfully applied in special cases, the nature of embedded systems – namely their close interaction with the physical environment as well as the specific constraints and limitations of embedded computing platforms – renders the application of formal verification methods of limited value for embedded software. One reason is that it is not sufficient to prove the correctness of source code itself, but also that of every step/tool along the chain from code development (either manually or automated from models) over compilation and linking down to the execution on the embedded platform [13][12]. Consequently, testing remains the major quality assessment technique in industry.

Testing, of course, needs *test cases*, i.e. at least the input to the system and the expected result. Often, additional actions to bring the software or system under test (SUT) into a state where the test can be carried out as well as a test environment are also needed. Developing and maintaining smart test cases manually is expensive and often requires deep domain knowledge, which contributes significantly to the overall testing costs of a system development. For safety-critical applications, these testing costs can make to up 50% and more of the overall design and development costs [10].

The EU-project MOGENTES aims at improving this situation by developing methods for the automated generation of efficient test cases, manifesting efficiency with respect to both cost reduction and quality improvement. In the next section, the basic principles of MOGENTES are described and the motivation for the taken approaches is given. Section 3 outlines one of the taken approaches, while Section 4 presents an example used to examine the approach, and summarizes some results. Section 5 contains the conclusion and an outlook on future work.

2 MOGENTES Overview

2.1 Motivation

Methods and tools for automated generation of test cases, either from source code or from models, are increasingly available (cf. e.g. http://www.cs.ru.nl/~lf/publications/BFS05.pdf). Usually, coverage metrics such as “all branches” or MC/DC[2] in a source code or “all states and transitions” in a state-machine model are used to evaluate the degree of coverage provided by a set of test cases. However, even if complete coverage is achieved according to these metrics, it is possible that simple faults remain undetected. Consider, for instance, the following code fragment for deciding the type of a triangle with edge lengths a, b, and c (this example is gratefully taken from [2]):

```
if ((a=b) and (b=c))
  then r := “equilateral”
  else
```

[1] „MOdel-based GENeration of Tests for Embedded Systems“, EU Frame Programme 7, contract number 216679; www.mogentes.eu

[2] „Modified Condition / Decision Coverage“: each elementary component of a conditional expression must at least once contribute solely to the value of the complete expression

```
if ((a=b) or (a=c) or (b=c))
  then r := “isosceles”
  else r := “scalene”
```

If it is tested with the three test cases

```
<a: 1, b: 1, c: 1, r: equilateral>
<a: 2, b: 2, c: 1, r: isosceles>
<a: 2, b: 3, c: 4, r: scalene>
```

which together would cover all three possible branches of the code fragment, then a typo such as “`(a=a)`” instead of “`(a=b)`” in the first line would remain undetected. However, if the second test case would be replaced by

```
<a: 1, b: 2, c: 2, r: "isosceles">
```

then r would be set to “equilateral” and the fault is detected. Otherwise, if the test would yield “isosceles” for r, it can be concluded that the specific fault is absent.

This illustrates the principle of fault-based testing, which is widely used in MOGENTES for generating test cases.

2.2 Mutation-Based Testing and Fault Models

A *mutation* is a syntactically correct modification (of some model or code), i.e. the modified artefact remains executable (interpretable). Hence, mutation testing can be applied to source code as well as to models. Actually, originally mutation testing was a way of assessing and improving a test suite by checking if its test cases can detect a number of faults injected into a program. The faults were introduced by syntactical changes of the source code following patterns of typical programming errors [6][7]. In general, small local mutations are used, based on two assumptions:

a) competent programmers do not make big mistakes, but are not immune against typos or small faults,
b) there exists a coupling effect so that complex errors will be found by test cases that can detect simple errors.

Fault models describe the kind of mutation (*mutation operator*), and may contain additional information such as parameters – e.g. the replacing item, possible application locations (within the artefact) and related semantics.

They of course depend on the specification language where they are applied. For C, for instance, typical fault models would be the replacement of a variable name by another variable name of compatible type, or an operator by another one of the same category. In Simulink models, examples would be replacement of operators or redirection of edges. For UML state diagrams, edges (representing state transitions) are associated with trigger conditions and optionally with guards (constraints to be fulfilled for activating a transition). For them, some possible fault models and related semantics are listed in the following table.

Fault model	**Parameter**	**Typical Meaning and result**
Replacing trigger event by another one	Replacing trigger event	Confusion of triggers – transition is activated by another event
Setting transition guard to FALSE	--	Equivalent to forgetting a transition - transition will never be executed
Aiming transition at another state	New target state	Confusion of transition result – trigger event leads to another (system) state

2.3 MOGENTES Approaches

Basically, three tracks addressed in MOGENTES exploit the principle of fault models:

The **UML** track uses UML models (class and state diagrams. OCL[3] is used for transition guards) for representing requirements because of several reasons. Firstly, state charts provide a good means for specifying behavioural aspects at a level of abstraction which permits to concentrate on the requirements without assuming too many implementation aspects. For instance, parallel regions in state diagrams allow separating behaviour aspects, which reduces the complexity and hence increases the readability of models. Secondly, an increasing acceptance of UML in industry can be observed, last but not least due to the availability of inexpensive tools. See Section 3 for more details on the UML track. It should be noted that other modelling techniques, e.g. activity or sequence diagrams [3] could have been used as well.

The test cases can be applied to an independent implementation (i.e. used for *black box testing*) or the implementation might be generated from the model itself. In the latter case, there is a risk that faults in the model remain undetected; hence, in this case more emphasis must be laid on proving the correctness of the model.

The **Simulink** track uses Simulink models for similar reasons: at least in certain sectors (e.g. automotive), it is even more accepted than UML for system specification and modelling. Hence, exploiting existing Simulink models for test case generation saves modelling effort. Of course, if the same models are used for both code and test case generation, again the correctness of the model needs to be verified by other means.

In principle, Simulink models are mutated and then C code is generated from them, which allows for

[3] Object Constraint Language, http://www.omg.org/spec/OCL

exploiting tools like CBMC (bounded model checker for C) for generating test cases. In this approach, research activities also concentrated on specific aspects such as handling of floating point numbers.

In the **Fault Injection** track, focus was laid on developing test cases for examining fault tolerance, fault propagation and fault detection latency of a system. Fault injection (FI) means that at runtime faults are (invasively) inserted into the SUT. Examples for such faults are "bit flip" (inverting the value of a bit, either in memory, a register, or at an input line) or "stuck at 0".

Typically, an "FI campaign" comprises a large number of individual tests (where faults of various kinds are systematically injected at different locations and points in time) and the execution can take days or even more time. Most of the tests may show no effect because the fault is injected at a location which is never used or immediately before it is updated. To improve this situation, a major activity in this track was to migrate FI from the target system to the model. At the model level it can be examined which of the fault injections cause an effect, and only these are relevant for application to the target system.

Besides these approaches, other alternatives for test case generation like UPPAAL[4] or Prover iLock®[5]have been investigated in MOGENTES. However, these tools do not rely on mutation-based testing. Finally, a tool integration framework was developed for easing the realization of the tool chains developed in the individual approaches.

3 Example: the UML Approach

Although it is possible to generate test cases directly from UML models, in this approach we preferred to transform the models to so-called *action systems* (AS), actually first to *object-oriented action systems* (OOAS) [5] and then to plain AS. This gives the UML models a precise semantics. The transformation step to OOAS is also used for generating the *mutants*. As illustrated in Figure 1, both the original and mutated OOAS are then used for test case generation (TCG). This is achieved by searching test cases which force the mutant to behave differently from the original. In this context, a test case is a sequence of inputs (stimuli) and expected outputs (results), possibly enriched by durations and timeouts. Furthermore, test cases can be non-deterministic when at a certain point several responses from the SUT are possible.

To find such test cases, for each mutant its conformance with the original is checked. This is done by exploring the behaviours of the original and the mutated AS. For that purpose, paths (i.e. sequences of actions) are searched starting from a given initial state until non-conformance is encountered. More precisely, we check for input output conformance (ioco) [11]. The result of this conformance check is a so called *product graph*, represented as a *labelled transition system* (LTS) [11]. LTS are directed graphs representing sequences of actions – in our case limited to input and output actions. Figure 2 shows the LTS for the use case described in section 4; edges prefixed with "ctr" denote inputs (*controllable*), and edges prefixed with "obs" denote outputs (*observables*).

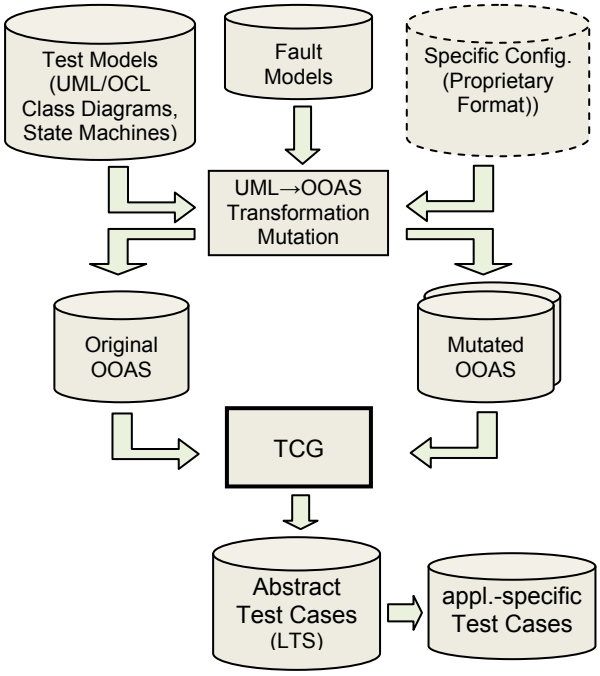

Figure 1. MBTCG tool chain

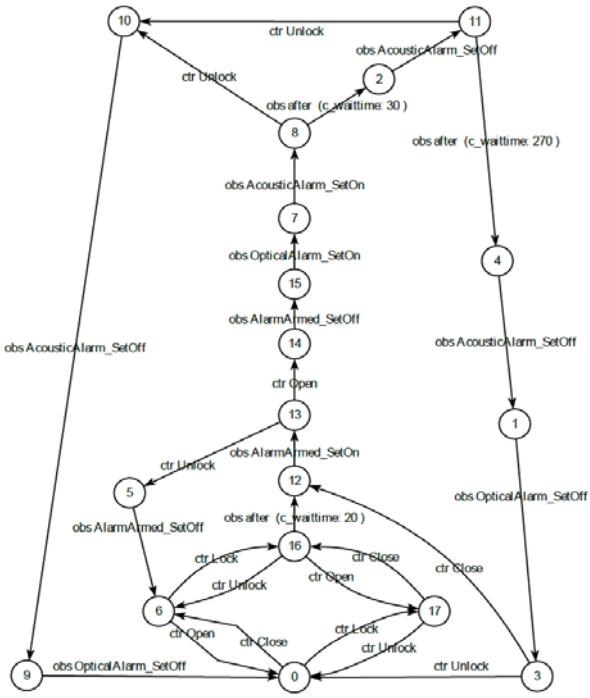

Figure 2. LTS example (car alarm system)

[4] a model checking environment (www.uppaal.com)
[5] from Prover Technologies (www.prover.com)

The product graph may contain transitions to two types of sink states, i.e. “pass” and “fail”. The behaviour of the mutant following a pass state conforms to the original, hence exploration stops in such states. A transition to a fail state denotes non-conformance between mutant and original. For further details see [4].

A test case consists of a selection of input and output transitions in the product graph. It contains at least one pass state. It may also contain transitions to inconclusive states. From such a state a pass state cannot be reached anymore. Fail states are implicit and can be reached from any state after the observation of an unexpected output.

Verdicts in test cases (either pass, fail, or inconclusive) are always given after the observation of actions sent by the SUT. The observation of no action, i.e. so called quiescence, can also be detected as a fault by the ioco relation. For details about the test case generation approach see [1].

Since test cases are derived from models representing requirements, and explicitly include the pass and inconclusive verdicts – or the verdict is fail – they also provide the “test oracle” (i.e. a means to decide whether the execution of a test case by the SUT was successful or not) in accordance with the requirements.

The obtained “abstract” test cases (ATC) are finally converted into a notation appropriate for the target system, which are called “concrete” test cases. Furthermore, some applications need additional configuration data, which is indicated by the dashed symbol at top right of Figure 1.

4 Example: Car Alarm System

4.1 Requirements Modelling

One of the use cases of MOGENTES is a car alarm system (CAS), for which following (somewhat simplified) requirements are defined [9] (another use case is described in [8]):

- The system is armed 20 seconds after the vehicle is locked and the bonnet, luggage compartment and all doors are closed.
- The alarm sounds for 30 seconds if an unauthorised person opens the door, the luggage compartment or the bonnet. The hazard flasher lights will flash for five minutes.
- The anti-theft alarm system can be deactivated at any time – even when the alarm is sounding – by unlocking the vehicle from outside.

Figure 3Figure 3 shows the state diagram representing these requirements.

4.2 Test Case Generation Results

From the fault models developed for UML state diagrams, 15 were applicable to 20 locations in the CAS model, resulting in 111 mutants (not the entire 300 possible mutations result in feasible and syntactically correct mutants). For these, 152 test cases have been generated using a shortest path depth strategy.

Of course, all 152 test cases together find (“kill”) all 111 mutants and thus yield coverage of 100% with respect to the faults represented by these mutants. However, in order to improve efficiency of testing, it is of interest to find a minimal set of test cases which still finds all mutants. In our case, this minimal set contains 8 test cases. An interesting observation is that while the strongest test case covers 75% of the mutants, the weakest covers only one mutant. But this test case is needed, because it finds a mutant which is not found by any other test case! Figure 4 gives a graphic impression of the relation between test cases and mutants; the circle at centre denotes the mutant detected by only one automatically generated test case.

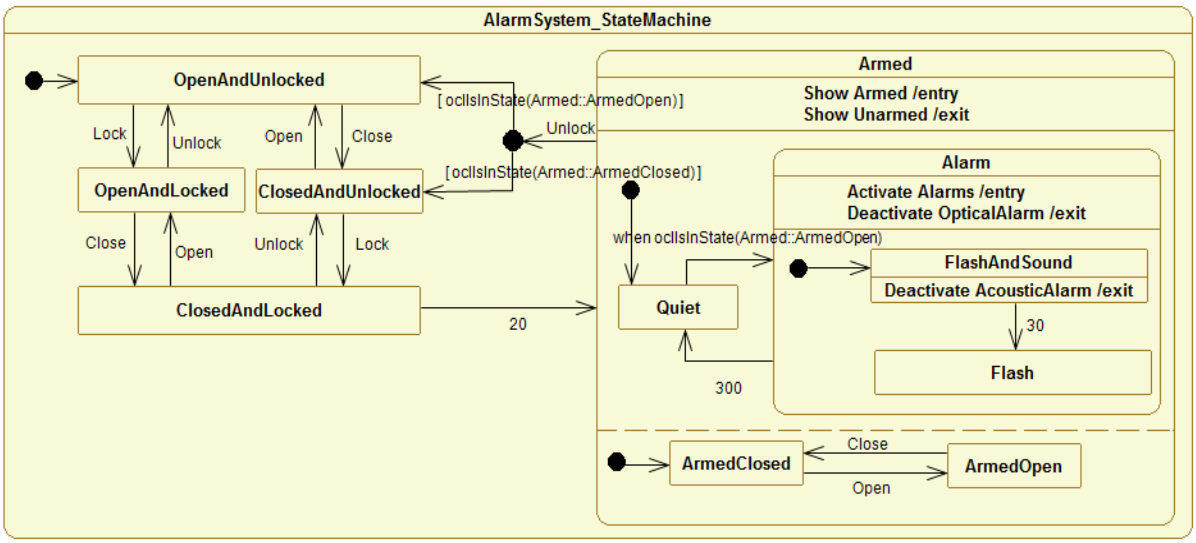

Figure 3. CAS requirements model as UML state diagram

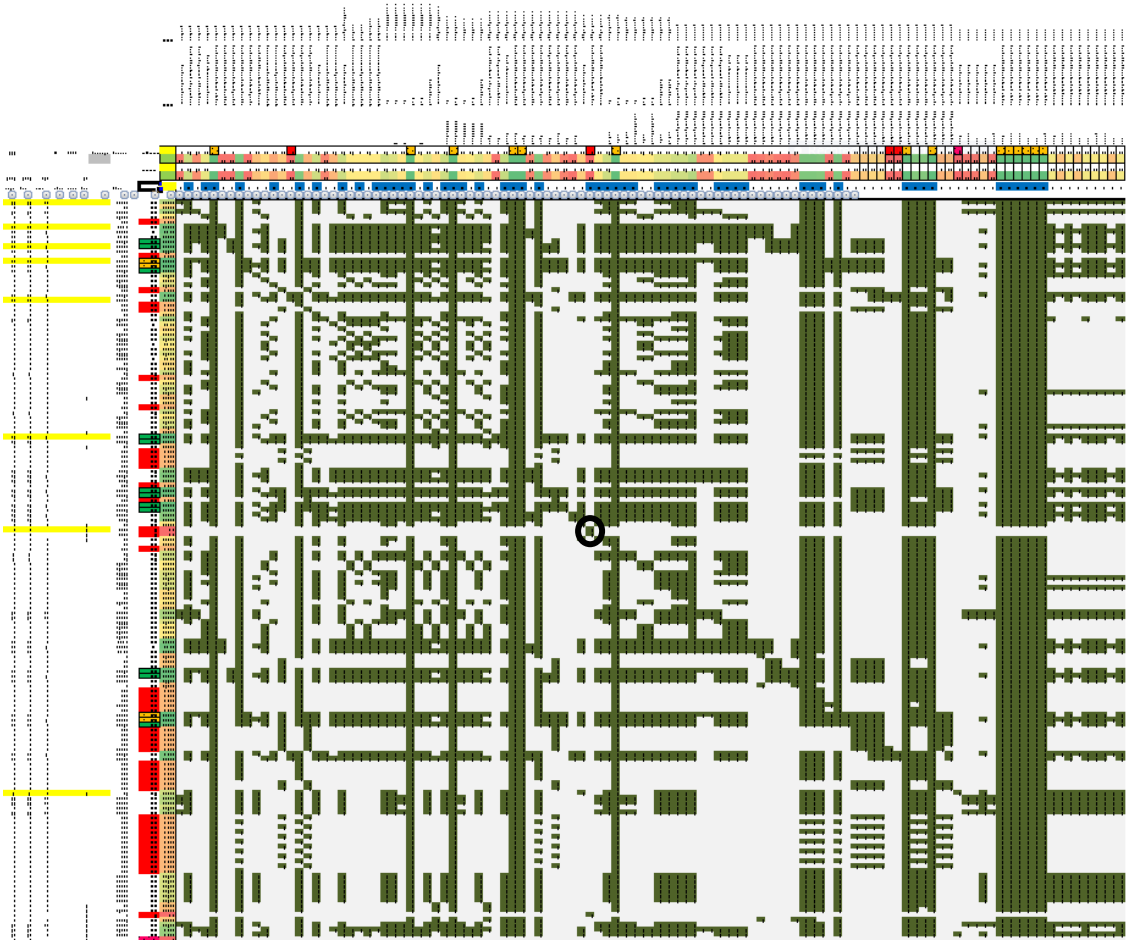

Figure 4. Zoomed out kill matrix, mutants (horizontal) vs. test cases (vertical); dark coloured cells are kills

5 Conclusions and Future Research

In this paper, a new approach for generating test cases based on model-mutation, and its application to an industrial use case was presented. This approach does not only allow to effectively generating test cases which assess conformance of a target application with functional (safety) requirements. It also allows to explicitly searching for faults or – if applied successfully – gives evidence for the absence of the respective faults. As such, test cases found with this method are more effective in detecting potential faults than those found with conventional methods. Together with strategies for identifying minimal sets of test cases that find all mutations, a powerful means for both improving quality and reducing costs of testing (embedded) systems is provided.

Future research activities will address current scalability limitations, the problem of equivalent mutants, reduction of test suites, and domain-specific fault models, which are discussed in the following.

5.1 Domain-specific Fault Models

The fault models currently used are model-language oriented. Although they exhibit some domain-specific semantics (i.e. allow an application-specific interpretation), it shall be investigated whether the effectiveness of the approach can be improved when trying to take application-specific faults into special consideration. This may imply introduction of non-local fault-models.

5.2 Scalability Limitations

The techniques used for finding test cases which force mutants to behave differently from originals are based on formal methods, e.g. model checking. They tend to be affected by inner state space explosion, in particular for state models with parallel regions. Consequently, the described approach does not scale well with the size of requirements models (number of parallel regions, states and transitions).

Several strategies have been identified – and are partially already under investigation – for coping with this issue. For instance, partial order reduction could help in cases where only the observation of a number of outputs is of interest, while the specific interleaving is irrelevant (and an alternative "single observable" is not possible).

Another promising approach is symbolic execution, i.e. tracking symbolic rather than actual values; this

will be helpful in particular for applications with large sets of possible parameter values, e.g. speed measures.

Furthermore, techniques for simplifying models, either through (object-oriented) abstraction, decomposition, extraction of sub-models, or other approaches shall be investigated to allow the use of the described approach to (arbitrarily) large applications. Of course, model decomposition might imply subsequent composition of test cases.

Finally, the technique of online testing could be of interest. There – instead of pre-computing all further potential paths after a non-deterministic point where the SUT can respond in more than one valid ways – only the SUT's actual output has to be followed. The challenge here is that the TCG has to become real-time: it must react to the SUT as the real environment would.

5.3 Equivalent Mutants

Related to the scalability issue is the problem of mutants which show behaviour equivalent to that of the original, because no test case can be found to kill them – but detecting this often takes significant amounts of computing resources.

Of course, it is trivial to avoid mutations which do not change behaviour, such as converting "a > b" into "b < a". But in the context of the respective model, also mutations which usually result in observable mutants may generate equivalent mutants. Nevertheless, a heuristic approach is to avoid mutation operators which tend to produce equivalent mutants.

Other approaches which could be investigated are e.g. k-induction over search depth, or fix-point analysis to derive invariants from the model structure.

5.4 Efficient Test Suites

Evidently, mutation-based test case generation techniques can produce (very) large sets of test cases (with potentially several thousands of test cases).

One way to reduce such a test suite to an efficient minimum is discussed in clause 4.2. However, this approach is only feasibly if the complete set of mutants is fairly small. Currently, the set of already found test cases is applied to each new mutant, and only in the case that no test case kills the new mutant, the quest for an appropriate test case has to be performed. In general, the final set of test cases is much smaller than the number of mutants. But clearly, the sequence of processed mutants influences the final set of test cases. Work is ongoing on further optimization strategies based on mutation operators and location priorities.

References

[1] B. K. Aichernig, H. Brandl, E. Jöbstl, W. Krenn; *Efficient Mutation Killers at Work*; in Proc. of 4th IEEE Int. Conf. on Software Testing, Verification and Validation (ICST); pp. 120–129; 2011

[2] B. K. Aichernig, He Jifeng; *Mutation testing in UTP*; in Journal of Formal Aspects of Computing; Vol.21, No.1-2, February 2009; Springer, London/UK; DOI 10.1007/s00165-008-0083-6

[3] O. Alt; *Generierung von Systemtestfällen für Car Multimedia Systeme aus domänenspezifischen UML Modellen*; INFORMATIK 2006 Informatik für Menschen Band 2, C. Hochberger, R. Liskowsky, Eds., 10 2006; GI-Edition Lecture Notes in Informatics, Gesellschaft für Informatik, Bonn

[4] H. Brandl, M. Weiglhofer, B. K. Aichernig,; *Automated Conformance Verification of Hybrid Systems*; In Proc. of the 2010 10th Int. Conf. on Quality Software (QSIC'10), pp.3-12, IEEE Computer Society; ISBN 978-0-7695-4131-0. DOI: http://dx.doi.org/10.1109/QSIC.2010.53

[5] M. M. Bonsangue, J. N. Kok, K. Sere; An Approach to Object-Orientation in Action Systems; in Mathematics of Program Construction, Springer LNCS 1422, 1998, 68–95

[6] R. DeMillo, R. Lipton, F. Sayward; *Hints on test data selection: Help for the practicing programmer*; IEEE Computer, Vol.11, No.4, April 1978, 34–41

[7] R. G. Hamlet; *Testing programs with the aid of a compiler*; IEEE Trans. on Software Engineering, Vol.3, No.4, July 1977; 279–290

[8] W. Herzner, R. Schlick, H. Brandl, W. Krenn, W. Schütz; *Towards Generation of Efficient Test Cases from UML/OCL Models for Complex Safety-Critical Systems*; In Proc. of Mikroelektronik 2010, 7.-8. April 2010, TU Wien, ÖVE Schriftenreihe Nr.56, 2010

[9] O. Hofmann, J. Wiessalla, E. Pofahl, T. Lenzen, R. Schlick, W. Herzner; *Model Based Generation of Test Cases for Safe and Reliable Vehicle Software in the Framework of MOGENTES*; in Proc. of the 10th Int. Symp. on Advanced Vehicle Control (AVEC); Loughborough, UK, August 23-26, 2010

[10] G. J. Myers C. Sandler; *The Art of Software Testing. John Wiley & Sons*; 2004. ISBN 0471469122

[11] J. Tretmans; *Test generation with inputs, outputs and repetitive quiescence*; Software - Concepts and Tools, 17(3):103–120, 1996

[12] Z. Shao; *Certified Software*; Comm. ACM, Dec. 2010, Vol.53, No.12, 56–66

[13] D. Walter, H. Täubig, C. Lüth; *Experiences in Applying Formal Verification in Robotics*; Proc. of SAFECOMP 2010 (29th Int. Conf. on Computer Safety, Reliability and Security); Springer LNCS 6351, 347–360