Towards an Integrated Methodology for the Development and Testing of Complex Systems

Philipp Helle, Wladimir Schamai

EADS Innovation Works

Hamburg, Germany

Email: {philipp.helle,wladimir.schamai}@eads.net

Abstract—**This paper reports on a framework for the development and testing of complex systems. The framework provides a meta-model for the description of systems at different levels of abstraction which is used as a basis for the combination of model-based testing (MBT) techniques for automated test case generation with executable requirement monitors that continuously observe the status of the System under Test (SuT) during test execution. The overall goal is to reduce the total development and testing effort for complex systems. This is accomplished by enabling a high degree of automation and reuse of engineering artefacts throughout the systems engineering lifecycle.**

Keywords—*Model-based Systems Engineering, Model-based Testing, Monitor-based Testing, SysML.*

I. Introduction

The ever-increasing complexity of products has a strong impact on time to market, cost and quality. Products are becoming increasingly complex due to rapid technological innovations, especially with the increase in electronics and software even inside traditionally mechanical products. This is especially true for complex, high value-added systems in the aerospace and automotive domain - the methodology was developed and is therefore embedded in an aeronautic context but generally is independent of a specific domain - that are characterized by a heterogeneous combination of mechanical and electronic components. System development and integration with sufficient maturity at entry into service is a competitive challenge in the aerospace sector. Major achievements can be realized through efficient system testing methods.

"Testing aims at showing that the intended and actual behaviours of a system differ, or at gaining confidence that they do not. The goal of testing is failure detection: observable differences between the behaviours of implementation and what is expected on the basis of the specification"[1].

The typical testing process is a human-intensive activity and as such it is usually unproductive and often inadequately done. It requires human test engineers to manually write test cases. A test case contains a series of test inputs and expected results. Nowadays, the test execution especially on lower levels of testing is largely automated. Nevertheless, this process is cumbersome and costly. Therefore, testing is one of the weakest points of current development practices. According to the study in [2] 50% of embedded systems development projects are months behind schedule and only 44% of designs meet 20% of functionality and performance expectations. This happens despite the fact that approximately 50% of total development effort is spent on testing [2], [3]. This shows the importance and desirability of reducing test effort by advances in the testing methodologies.

Testing needs to be applied as early as possible in the lifecycle to keep the relative cost of repair for fixing a discovered problem to a minimum. This means that testing should be integrated into the lifecycle model so that each phase in the development contributes to the verification of the product as Figure 1 shows. Laycock claims that "the effort needed to produce test cases during each phase will be less than the effort needed to produce one huge set of test cases of equal effectiveness on a separate lifecycle phase just for testing"[4].
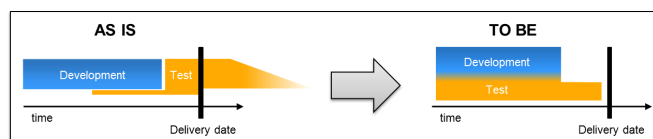

Fig. 1: Envisaged process change

This paper reports on a framework to further automate the system testing process. It is a continuation of the work earlier reported in [5]. The framework provides a meta-model for the description of systems on different layers of abstraction and combines model-based testing (MBT) techniques for automated test case generation based on a whitebox SysML model of the system with executable requirement monitors that continuously observe the status of the System under Test (SuT) during test execution. The overall goal is to achieve a high degree of automation and reuse of engineering artefacts throughout the systems engineering lifecycle.

Paper structure: First we present background information on SysML, MBT and monitor-based testing (Section II) before we will explain the methodology in detail (Section III). Finally, we propose a number of ideas for future research (Section IV) and close with a summary of the current status (Section V).

II. Background

This section provides background information on SysML, Model-based testing, Monitor-based testing and related work.

A. SysML

The Unified Modeling Language (UML) [6] is a standardized general-purpose modelling language in the field of software engineering and the Systems Modeling Language (SysML) [7] is an adaptation of the UML aimed at systems engineering applications. Both are open standards, managed

and created by the Object Management Group (OMG), a consortium focused on modelling and model-based standards.

SysML is not a methodology, i.e., it does not define what steps need to be performed in what order and which diagrams should be used for which step. Estefan [8] provides an overview of existing methodologies used in industry, some of which use UML-based languages. SysML is a graphical modelling language, i.e., diagrams are used to create and view model data. However, the graphical representation is decoupled from the actual model data. The model data and its graphical representation are typically stored in different files in UML/SysML tools.

Neither UML nor SysML define complete model execution semantics in their core specification. This is different from modelling and simulation languages, such as Modelica [9], which specify the syntax (textual notation) as well as the execution semantics. However, work is underway to resolve that [10], [11], [12]. In the mean time, SysML tool suppliers often provide their own execution semantics [13], so it is possible to include action code into models, generate code from the models and then execute them.

B. Model-based testing

The term MBT is widely used today with slightly different meanings. Surveys on different MBT approaches can be found in [1], [14], [15]. The most simple one is that "Model-based testing (MBT) relates to a process of test generation from an SuT model by application of a number of sophisticated methods"[16].

Model-based testing is a variant of testing that relies on explicit behaviour models that encode the intended behaviour of a system and possibly the behaviour of its environment. The use of explicit models is motivated by the observation that traditionally, the process of deriving tests tends to be unstructured, barely motivated in the details, not reproducible, not documented, and bound to the creativity and expertise of single engineers. The idea is that the existence of an artefact that explicitly encodes the intended behaviour can help mitigate the implications of these problems [1].

Intensive research on MBT and analysis has been conducted in recent years, and the feasibility of the approach has been successfully demonstrated, e.g., in [17], [16]. Yet, Boberg [18] shows that most studies apply model-based testing on the component level, or to a limited part of the system while only few studies focus on the application of the technique on the system or even aircraft level. The main difference being that the goal of a system integrator such as Airbus is not to produce code but to provide a high quality specification that can be handed over for implementation to a supplier. Giese [19] explains that this slow adoption is not only due to scalability reasons but he also claims that "to benefit from formal verification and early simulation, a model must be precise and detailed with respect to all aspects that are the subject of verification. This can usually be carried out in the detailed design phase at the earliest"[19].

A major distinction between the different available MBT approaches can be made by looking at the source of the generated test cases [19]. Some approaches rely on separate explicit test models that are disjunct from the system or specification model, as depicted by Figure 2 while other approaches don't make that distinction and generate test cases from the defined system behaviour as shown by Figure 3.

The usage of explicit test models reflects the different objective (validation vs. solution) and point of view (tester vs. implementer) in creating a test model rather than a specification model [20]. A test model is a model representing all possible stimulations of input of the system interacting in various usage contexts and normally also includes verification points stating what is a correct response from the system to an input and what not. It thereby follows a tester's view who also has to think of how to combine the possible input stimuli of a system to achieve a high confidence in its correctness.

The main benefit of this approach is the degree of independence it naturally entails between the generated test cases and the system. The generated test cases can thus be used directly to test any form of the SuT, either a model or the implementation. Additionally, as the test model is not a part of the design it can be optimised for validation and verification purposes thereby increasing the chance to uncover defects that are outside the focus of the design artefacts [19]. A drawback of the approach is that there are two models that have to be kept consistent with the requirements at all time which requires further effort.
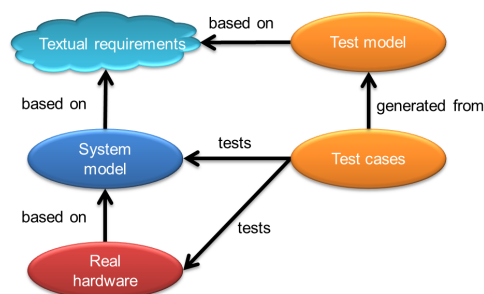

Fig. 2: Model-based testing using explicit test models

One example for an approach that does not rely on explicit models is the work from Lettrari [21] that is the basis for the commercially available IBM Rational Rhapsody Automatic Test Generator (ATG) tool. Test cases are generated from a behaviour model of the SuT using model coverage as test selection criteria. Automated test case generation uses constraint based symbolic execution of the model and search algorithms.

The main benefit is that the approach does not require the creation and maintenance of a separate test model. On the other hand, since the test case generation is not guided by a test engineer it cannot distinguish between "good" and "bad" test cases. The only goal for the generator is to achieve a high degree of model and/or code coverage by generating stimuli that force the executable system model to visit all states and transitions and call all functions of the system's components. Furthermore, there is no independence between the generated test cases and the system model. This means that the test cases cannot be used to test the model they were generated from if the test success criteria is that the observed behaviour and the test case behaviour are the same.
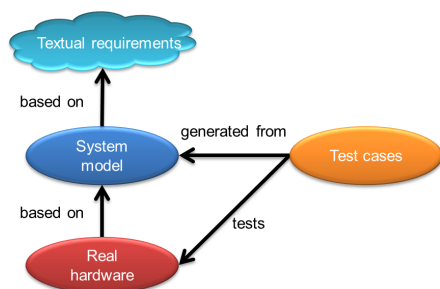
Fig. 3: Model-based testing using design/specification models

C. Monitor-based testing

The idea for formalizing a natural language requirement statement into a requirement monitor is similar to the monitor concept used in runtime verification [22], [23]. Like in runtime verification, the main purpose of the requirement violation monitor is to detect requirement violations without intervening into the analyzed system. A more formal definition states, that "Runtime verification is the discipline of computer science that deals with the study, development, and application of those verification techniques that allow checking whether a run of a system under scrutiny satisfies or violates a given correctness property"[22].

Runtime verification itself deals with the detection of violations of correctness properties. Thus, whenever a violation is observed, it typically does not influence or change the programs execution, say for trying to repair the observed violation. Checking whether an execution meets a correctness property is typically performed using a monitor. In its simplest form, a monitor decides whether the current execution satises a given correctness property by outputting either yes/true or no/false [22].

D. Related Work

In [24], Artho et. al. propose a method for combining test case generation and runtime verification for software systems. In their framework they combine automated test case generation, which is based on a systematic exploration of the input domain of the tested software system using a model checker that is extended with symbolic execution capabilities with runtime verification techniques, that monitor execution traces and verify them against properties expressed in a temporal logic notation. They include further capabilities for the analysis of concurrency errors, such as deadlocks and data races. The paper also provides a description of the application of the method using a NASA rover controller.

While similar on an abstract level, our work differs from the work by Artho et. al. in some major points. Firstly, the test oracles are written as temporal logic formulas whereas we use SysML for both the modelling of the system as well as the requirement monitors. Secondly, the test scenarios are generated based on a definition of all possible inputs using a model checker, whereas we generate the test scenarios from a whitebox model of the system under test.

III. METHODOLOGY FOR DEVELOPMENT AND TESTING OF COMPLEX AIRCRAFT SYSTEMS

This section provides a description of our methodology in terms of the overall concept, the underlying metamodel and the envisaged process.

A. Concept

Our methodology combines monitor- and model-based testing to test the system model and the resulting system. Our aim is to achieve a high degree of reuse of artefacts from early development stages at later development stages and a high degree of automation throughout the process. Since we consider multiple levels of abstraction in our metamodel it is necessary to provide means which can verify a model at any abstraction level or the final product without the need for redeveloping the verification means for each verification stage. To this end, we use executable requirement monitors, which can be built very early on as soon as the first requirements are defined and which can be adapted easily for verifying the models or the product. Also, these monitors can be reused for testing different variants and/or design alternatives.

Figure 4 provides an overview of the main artefacts involved and their relations.

A requirement monitor is an exectuable model representing one requirement that, at any point in time, indicates the requirement violation status. The status should be enumerated with at least the following values:

- Not evaluated (default value), to indicate that the requirement has not been evaluated yet. Typically, this means that a necessary precondition has not been met yet.

- Not violated, to indicate that no violation has happened and implying that the requirement has been evaluated.

- Violated, to indicate a violation of the requirement and implying that the requirement has been evaluated.

This enumeration is referred to as three-valued semantics in [22] with the literals inconclusive, false and true respectively.

The monitor status can be obtained from a monitor at any point in time and can change between not evaluated, not violated and violated in any possible way. Following this approach, the status of the individual reuqirement monitors that are instantiated during one test can be used in aggregation to derive the test verdict. Removing the test verdict from the test cases will enable the reuse of test cases, that we now call test scenarios, for the verification against several requirements.

The task of converting the natural language statement into a formal language will require a correct interpretation of the requirement statement and the ability to translate the meaning into a model that expresses exactly the same. The general systematic way for deriving a monitor from natural language requirement is as follows:

1) Read the requirement statement
2) Identify properties that can be quantified either by explicit numbers or by logical conditions

3) Identify pre-conditions (if any), which must be satis-fied before the requirement can be evaluated
4) Express when the requirement is violated and when not

Neither a particular design of the system nor scenarios are needed for formalizing a requirement. The resulting monitor can be used for the verification of any design alternative of the system using any scenario. Generally, the task of formalizing a requirement into a requirement monitors can be accomplished in many different ways using different formalisms. We decided to use SysML for the task because using the same notation for design and testing artefacts enables integrated development and testing without the need for additional tools or data converters.

We drive the tests using scenarios that we generate from the system models using MBT technology. Since we derive the test verdict from the requirement monitors independently from the system model we can use the scenarios derived from the system model to actually verify the system model as well as the final product.
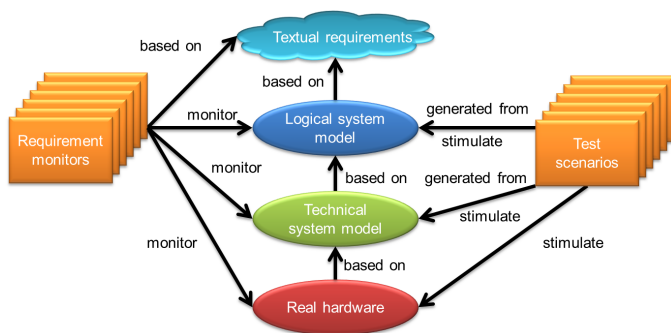

Fig. 4: Model-based testing using monitors

B. Meta-model

For our purpose, we extended the already established meta model for functional and systems architecture modeling [25] to allow a distinction between the functional, logical and the technical architecture of the system as depicted by Figure 5.
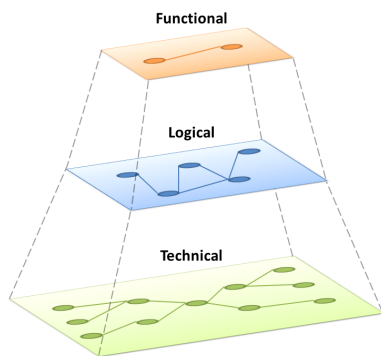

Fig. 5: Levels of abstraction

The main rationale for the distinction between these differ-ent layers is reusability. Between different aircraft programmes the functional architecture of a system is quite stable whereas the implementation can differ drastically. For a given aircraft programme the logical architecture is fixed quite early but different technical implementations might be considered and compared in trade studies. Ideally, we can now reuse the same functional architecture that is mature and proven and derive different logical and even more possible technical implemen-tations that satisfy these functional needs.

The functional architecture, consisting of functions and data exchanges via functional dependencies is mapped to a logical system architecture, consisting of logical components that are instances of logical component classes and logical links between these components. This logical architecture can then in turn be mapped to the technical architecture of the system, which contains technical components, i.e., devices, and technical connectors, i.e., cables that connect the components. As can be seen from Figure 6 the relations between the ele-ments in the different modelling layers allow a full traceability. This is crucial especially for maintaining the consistency of the models after changes.
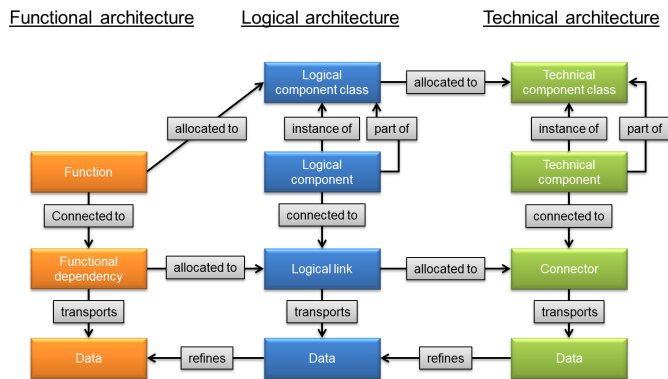

Fig. 6: Meta-model for current approach

While the modelling of the functional architecture in our approach is purely descriptive, the logical and the technical system architecture models are fully executable. Typically, the complexity of the models increases from the functional over the logical to the technical model. This is mainly due to two reasons: Firstly, when following this top down approach for systems modelling the level of abstraction decreases, which in turn increases the level of detail and complexity. Secondly, most aircraft systems require a certain degree of redundancy to abide by the safety constraints. A fact which is normally not considered during the functional analysis, only partly in the logical design but has the most impact on the technical architecture.

C. Process

The overall process underlying our methodology is straight forward and consists of the following steps:

1) Formalize requirements: create a violation monitor for each requirement
2) Build system models
3) Generate test scenarios from system models using MBT
4) Prepare the test environment: instantiate the monitors of the requirements that can be tested using the

available scenarios and connect them to the SuT (models or real hardware) appropriately
5) Execute tests: run all defined scenarios
6) Evaluate tests: aggregate the individual statuses of the requirement monitors that were active during a test to derive a test verdict

IV. FUTURE WORK

This section provides a couple of topics for current or future work for extending the approach described in this paper. Apart from extensions to the framework, we are also working on the application of the methodology for a concrete industrial-based use case to validate the framework.

A. Combination of model-based testing and model-based analysis

Dijkstra's famous aphorism holds that tests can only show the presence of errors not their absence [26]. Analysis techniques, e.g., model checking can be used to proof required characteristics of a system. Model-based analysis (MBA) and testing are complementary quality assurance techniques since static and dynamic analysis provide altogether different types of information: typically, static analysis provides general information about a model of the system while dynamic testing provides specific information about the system under test itself. Substantial quality and cost improvement can be obtained when they are systematically applied in combination.

One example for such a combination of MBT and MBA is the application of MBA in form of a model checker to improve the completeness of a test suite generated from a whitebox model using MBT as Figure 7 shows. The problem that is addressed by this method is that the automatic test scenario generator does not always achieve to generate a test suite with 100% coverage (coverage for this scenario means model/code coverage). At the moment, manual effort is required to complete a test suite to achieve 100% coverage. This manual effort can be replaced by the application of a model checker. If a test case generator manages to cover 95 out of 100 states of a model using test scenarios then we can write properties that check the reachability of the remaining five states. If the model checker manages to reach a state then the proof trace provided by the model checker can be directly added to the test suite as a new test scenario. If the model checker cannot find a solution for reaching a state then the model needs to be adapted.
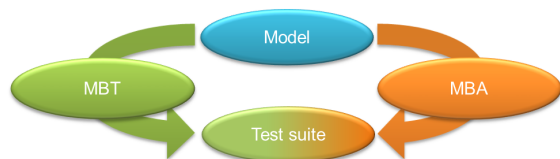

Fig. 7: Combination of model-based testing and model-based analysis

B. Combination of test scenarios obtained from different sources

Evluation of different MBT approaches and tools in the recent past, e.g., [27], [28] showed, that each tool has specific strengths and weaknesses and almost none of them can replace additional manual test scenario creation completely. If we use more than one test scenario generation approach and if we allow test scenario generation at different levels of abstraction as Figure 8 shows, then there is a high probability that the resulting test suite contains a high amount of redundant test scenarios. In order to test efficiently, especially when we are in the phase of hardware testing where a test run is much more expensive than a test run on a model, the redundancy in the test suite must be reduced to find an optimal test suite. Adaptation of previous work, e.g., [29], on that topic to our overall development and testing approach is currently being investigated.
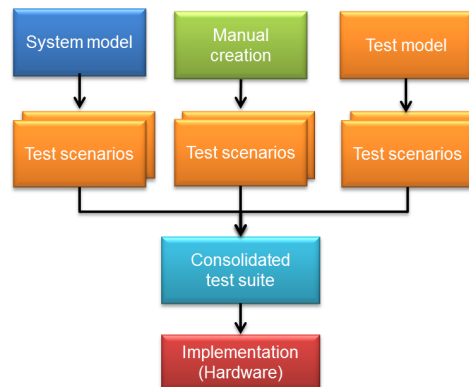

Fig. 8: Optimal test suite from different sources

C. Automated model-composition and results evaluation

The creation of models that integrate requirement monitors, a SuT system model and scenarios, i.e., the finding of suitable combinations of system model, scenarios and requirements, can be automated. Such a combined test model consists of one system model, which can be logical or technical, one scenario that can stimulate the design alternative and a set of requirements, which can be tested using the selected scenario. To automate the process further information is needed to evaluate the suitability of a combination of a test scenario and a design model, a test scenario and a requirement or a requirement and a system model. An approach for encoding this information and thereby enabling the automated composition of such test models is presented in [30]. Combining this approach with the one presented here is ongoing work. Additionally, running the tests, post-processing of the test results, and presenting the verification results appropriately can also be done in an automated fashion.

V. CONCLUSION

We presented a framework for an integrated development and testing approach of complex systems. The main driver behind this development was the need for more efficient testing. This was succesfully achieved by increasing the degree of reusability of engineering artefacts and automation of the testing process in the following way:

- Reusability
  ○ Explicit modelling of different architecture levels enables reuse of architectures.

- ○ Requirement monitors can be reused for testing different architecture levels as well as the real hardware product.
- ○ Removing verdicts from test cases allows using the same test scenario for testing multiple requirements. Additionally, testing a requirement in different test scenarios increases the confidence in the conclusions drawn from the test results.

- Automation
  - ○ Executable requirement monitors allow automated test verdict derivation.
  - ○ Generation of scenarios using MBT.
  - ○ Automated test execution of formal test scenarios.

The approach requires, as most model-based approaches, a frontloading of effort, a personnel shift and a different education of the involved people compared to the current state of practice. While evidence suggests that, through the high degree of reuse and automation, the effort for model-based testing is only slightly higher than the one for traditional testing [31] the adoption of the presented approach in an industrial environment nevertheless requires a rethinking of traditional roles and process steps.

ACKNOWLEDGMENT

REFERENCES

[1] M. Utting, A. Pretschner, and B. Legeard, "A taxonomy of model-based testing approaches," Software Testing, Verification and Reliability, vol. 22, no. 5, pp. 297–312, Aug 2012.

[2] V. Encontre, "Testing embedded systems: Do you have the GuTs for it," IBM: http://www.ibm.com/developerworks/rational/library/459.html, Nov 2003.

[3] A. Helmerich et al., "Study of Worldwide Trends and R&D Programmes in Embedded Systems in View of Maximising the Impact of a Technology Platform in the Area," European Commission: http://www.artemis-austria.net/uploads/media/FAST_final-study-181105_en.pdf, Nov 2005.

[4] G. Laycock, The theory and practice of specification based testing. Department of Computer Science, University of Sheffield, 1993.

[5] P. Helle and W. Schamai, "Specification model-based testing in the avionic domain - Current status and future directions," in Proc. Sixth Workshop on Model-Based Testing (MBT 2010), ser. ENTCS, vol. 264, no. 3, 2010, pp. 85 – 99.

[6] Object Management Group, "OMG Unified Modeling Language (OMG UML), v2.3," 2010.

[7] ——, "OMG Systems Modeling Language (OMG SysML), v1.2," 2008.

[8] J. Estefan, "Survey of Model-Based Systems Engineering (MBSE) methodologies," INCOSE: http://www.incose.org/productspubs/pdf/techdata/mttc/mbse_methodology_survey_2008-0610_revb-jae2.pdf, 2008.

[9] P. Fritzson and V. Engelson, "Modelica - a unified object-oriented language for system modeling and simulation," in Proc. European Conference on Object-Oriented Programming (ECOOP98). Springer, 1998, pp. 67–90.

[10] B. Selic, "The less well known UML," in Formal Methods for Model-Driven Engineering, ser. LNCS, M. Bernardo, V. Cortellessa, and A. Pierantonio, Eds. Springer, 2012, vol. 7320, pp. 1–20.

[11] Object Management Group, "Semantics Of A Foundational Subset For Executable UML Models (FUML, v.1.0)," 2011.

[12] ——, "Concrete Syntax For A UML Action Language: Action Language For Foundational UML (ALF), v1.0.1 beta," 2013.

[13] D. Harel and H. Kugler, "The Rhapsody Semantics of Statecharts (or, on the executable core of the UML)," in Integration of Software Specification Techniques for Applications in Engineering, ser. LNCS, H. Ehrig et al., Eds. Springer, 2004, vol. 3147, pp. 325–354.

[14] M. Utting and B. Legeard, Practical Model-Based Testing: A Tools Approach. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2007.

[15] M. Broy, B. Jonsson, J.-P. Katoen, M. Leucker, and A. Pretschner, Model-Based Testing of Reactive Systems: Advanced Lectures (Lecture Notes in Computer Science). Springer, 2005.

[16] J. Zander-Nowicka, Model-based testing of real-time embedded systems in the automotive domain. Fraunhofer FOKUS, Berlin, 2009.

[17] T. Bauer, H. Eichler, A. Rennoch, and S. Wieczorek, Model-based Testing in Practice - Proc. 2nd Workshop on Model-based Testing in Practice (MoTiP 2009). University of Twente, The Netherlands, 2009.

[18] J. Boberg, "Early fault detection with model-based testing," in Proc. 7th ACM SIGPLAN workshop on ERLANG. New York, NY, USA: ACM, 2008, pp. 9–20.

[19] H. Giese, G. Karsai, E. A. Lee, B. Rumpe, and B. Schatz, Model-Based Engineering of Embedded Real-Time Systems, ser. LNCS. Springer, 2010, vol. 6100.

[20] H. Le Guen, F. Valle, and A. Faucogney, Model-Based Testing - Automatic Generation of Test Cases Using the Markov Chain Model. Wiley, 2012, pp. 29–81.

[21] M. Lettrari, "Using abstractions for heuristic state space exploration of reactive object-oriented systems," in FME 2003: Formal Methods. Springer, 2003, pp. 462–481.

[22] M. Leucker and C. Schallhart, "A brief account of runtime verification," Journal of Logic and Algebraic Programming, vol. 78, no. 5, pp. 293–303, 2009.

[23] J. Levy, H. Saïdi, and T. E. Uribe, "Combining monitors for runtime system verification," ENTCS, vol. 70, no. 4, pp. 112–127, 2002.

[24] C. Artho et al., "Combining test case generation and runtime verification," Theoretical Computer Science, vol. 336, no. 2, pp. 209–234, 2005.

[25] P. Helle, "Automatic SysML-based safety analysis," in Proceedings of the 5th International Workshop on Model Based Architecting and Construction of Embedded Systems, ser. ACES-MB '12. New York, NY, USA: ACM, 2012, pp. 19–24.

[26] E. W. Dijkstra, "The humble programmer," Communications of the ACM, vol. 15, no. 10, pp. 859–866, 1972.

[27] M. Shafique and Y. Labiche, "A systematic review of model based testing tool support, Technical Report SCE-10-04," Carleton University: http://squall.sce.carleton.ca/pubs/tech_report/TR_SCE-10-04.pdf, 2010.

[28] A. C. Dias Neto, R. Subramanyan, M. Vieira, and G. H. Travassos, "A survey on model-based testing approaches: a systematic review," in Proc. 1st Workshop on Empirical Assessment of Software Engineering Languages and Technologies (WEASELTech'07). ACM, 2007, pp. 31–36.

[29] G. Fraser and F. Wotawa, "Redundancy based test-suite reduction," in Fundamental Approaches to Software Engineering, ser. Lecture Notes in Computer Science, M. Dwyer and A. Lopes, Eds. Springer, 2007, vol. 4422, pp. 291–305.

[30] W. Schamai, P. Fritzson, C. J. J. Paredis, and P. Helle, "ModelicaML value bindings for automated model composition," in Proc. 2012 Symposium on Theory of Modeling and Simulation - DEVS Integrative M&S Symposium, ser. TMS/DEVS '12. Society for Computer Simulation International, 2012, pp. 31:1–31:8.

[31] T. Bauer, F. Böhr, and R. Eschbach, "On MiL, HiL, statistical testing, reuse, and efforts," in 1st Workshop on Model-based Testing in Practice (MoTiP 2008). Fraunhofer, 2008, pp. 31–40.