# Model-Based Testing for Model-Driven Development With UML 2.0 – Extended Abstract

Jan Peleska
University of Bremen
jp@tzi.de

M. Oliver Möller
Verified Systems International GmbH
moeller@verified.de

Helge Löding
Verified Systems International GmbH
loeding@verified.de

**Contents of this Presentation.** In this presentation we introduce methods and tool support for automated model-based testing of embedded real-time systems specified in UML 2.0. Our approach focuses on (1) composite structure diagrams for the definition of system components, operational environment, interfaces, and parallelism, (2) class diagrams and method specifications for the description of guard conditions, event handlers and further actions involving data transformations, (3) Statecharts for the specification of reactive system behaviour.

We describe the automated generation of test cases for SW integration and HW/SW integration testing, where test cases are derived from composite structure diagrams and Statecharts, and classes and methods have to be investigated in order to decide how to trigger the state transitions desired, by making true the associated guard conditions and by generating the necessary events.

Given a specification model of the system under test (SUT) using the 3 UML ingredients listed above, test cases can be generated in a fully automated way. The result, however, will be unsatisfactory in frequent situations as long as the automation tool generates test cases from the model in an arbitrary or random manner: Test cases which are "interesting" from the domain experts' point of view will occur very late in the test suite or event not at all, and the automation process will come up with many cases which could be readily identified by these experts as "useless" or of "low priority".

In order to increase the effectiveness of the automated test case generation process we introduce *built-in* and *user-defined test case generation strategies*. The former are test case generation concepts which – according to our experience – are crucial for the most test campaigns:

- A strategy for *transition coverage*. This strategy aims at finding an as small as possible set of test cases which suffice to exercise each state transition of a Statechart (and therefore also each of its states) at least once. The importance of transition coverage is due to the fact that it implies requirements coverage for many UML specification models.

This strategy can also be further refined in an automatic way such that the components of compound guard conditions, say $[c_0$ and $c_1]$ are exercised with all significant combinations, say (1) $c_0 = c_1 = \texttt{true}$, (2) $c_0 = \texttt{true}, c_1 = \texttt{false}$, (3) $c_0 = \texttt{false}, c_1 = \texttt{true}$ (so-called *MC/DC coverage for guard conditions*).

- A strategy for *time-related boundary value tests*: In this strategy the points in time when an additional input is fed into the system under test are selected in such a way that time bounds specified in the Statecharts model are just met or just violated. This strategy helps to verify the correctness of timing behaviour, such as reaction on missed deadlines and timeouts.
- A strategy for *robustness tests*: Following this strategy the test automation system feeds input data to the system under test which should be ignored and discarded according to the system's current state.

For user-defined strategies, an easy-to-use variant of temporal logic is provided, allowing to describe

- which states or transitions of a Statechart should be covered at least / at most how many times in the test case execution,
- in which order certain states or transitions should be visited,
- which input ranges should be used,
- which time bounds should be enforced for time-dependant inputs.

An equally useful but conceptually different type of user-defined strategies is elaborated by means of *behavioural environment specifications*: Testing experts provide specifications of the admissible behaviour of the operational environment $E$. Now test data is no longer generated directly from the SUT specification $S$ but from $E$, and the specification $S$ is only used for checking the SUT outputs observed against the the expected behaviour $S$.

Generating the concrete (potentially timed) sequence of input data needed for the implementation of a test case it is necessary to interpret the guard conditions and input events which determine whether a specific transition is feasible. Moreover, inputs trigger actions in the system under test which may lead to the change of internal object states and further state transitions. To cope with these problems a *constraint generator* collects the conditions which have to be fulfilled in order to drive the system under test along the path through the specification model as proposed by the strategy. A *constraint solver* tries to construct the concrete test data – that is, the sequence of input data associated with the points in time specifying when to pass each input to the system under test – leading to the desired execution path. If the path is infeasible, learning mechanisms help to avoid future investigations of paths containing infeasible fragments.

**Practical Application.** A major portion of the results presented here has been implemented in the RT-Tester tool developed by Verified Systems International GmbH in cooperation with the first author's research group at the University of Bremen. The tool is currently applied in the field of avionic systems and railway control systems of the highest criticality levels. While RT-Tester supports all testing stages – from module testing via software and HW/SW integration testing to system integration testing (see, for example, [Pel02]) – the techniques described here are typically applied on software and HW/SW integration level. It should be emphasised that while this presentation refers to a specific tool, our objective is to point out the general conceptual and algorithmic features which – according to our experience – should be present in every test automation tool suitable for model-based testing of a wide range of embedded applications.

**Related Work.** In [PZ07] the close relationship between module testing and static analysis is elaborated, and we advocate that tools for the automation of module tests should also support automated static analysis and vice versa. In the model-based testing approach described in the present contribution, module testing and static analysis are used for the verification of the methods occurring as guard conditions, for the capture of events and for the implementation of actions: In the practical application of model-driven development with UML 2.0 these methods are typically directly coded using C/C++ or other programming languages, while more abstract method specification styles such as OCL are currently more of an academic interest. We recommend to check these programmed method implementations on module level before performing the model-driven software integration and HW/SW integration tests described here.

In contrast to other approaches to automated static analysis (see, e. g. [Cou05]) and testing we do not restrict the C/C++-code which is allowed to specify methods; in particular, the utilisation of arrays, pointers and pointer arithmetic is supported. In order to make this feasible the gcc compiler has been expanded for the extraction of detailed type information [Löd07]. Moreover, our abstract C/C++ interpreter follows some basic ideas of [VB04] but uses a more detailed memory model whose technical details are currently prepared for publication [PL08].

For the generation of concrete sequences of test input data it is necessary to solve potentially non-linear constraint systems involving transcendent mathematical functions. The constraint solver implemented in the RT-Tester system closely follows the ideas described in [FHT$^+$07] but uses additional data structures and learning techniques in order to optimise test case generation according to the selected strategies. In particular, we use tree structures for the encoding possible paths leading to specific locations or edges in the specification model; the basic concepts underlying these structures have been described in [BFPT06].

## References

[BFPT06] Bahareh Badban, Martin Fränzle, Jan Peleska, and Tino Teige. Test Automation for Hybrid Systems. In *Proceedings of the Third International Workshop on SOFTWARE QUALITY ASSURANCE (SOQUA 2006)*, Portland Oregon, USA, November 2006.

[Cou05] P. Cousot. The Verification Grand Challenge and Abstract Interpretation. In *Verified Software: Theories, Tools, Experiments (VSTTE)*, ETH Zürich, Switzerland, October 10–13 2005.

[FHT$^+$07] Martin Fränzle, Christian Herde, Tino Teige, Stefan Ratschan, and Tobias Schubert. Efficient Solving of Large Non-linear Arithmetic Constraint Systems with Complex Boolean Structure. *Journal on Satisfiability, Bo olean Mo deling and Computation*, 2007.

[Löd07] Helge Löding. Behandlung komplexer Datentypen in der automatischen Testdatengenerierung. Master's thesis, University of Bremen, August 2007.

[Pel02] Jan Peleska. Formal Methods for Test Automation - Hard Real-Time Testing of Controllers for the Airbus Aircraft Family. In *Proc. of the Sixth Biennial World Conference on Integrated Design & Process Technology (IDPT2002), Pasadena, California, June 23-28, 2002*. Society for Design and Process Science, June 2002. ISSN 1090-9389.

[PL08] Jan Peleska and Helge Löding. A C/C++ Memory Model for Abstract Interpretation and Module Testing. In Ralf Huuck and Bastian Schlich, editors, *First International Symposium on Real Software, Real Problems, and Real Solutions (RSPS 2008)*, 2008. In preparation.

[PZ07] Jan Peleska and Cornelia Zahlten. Integrated Automated Test Case Generation and Static Analysis. In *Proceedings of the* $6^{th}$ *International Conference on QA&Testing for Embedded Systems*, Bibao (Spain), 17 – 19 October 2007.

[VB04] Arnaud Venet and Guillaume Brat. Precise and Efficient Static Array Bound Checking for Large Embedded C Programs. In *Proceedings of the PLDI'04, June 9-11, 2004, Washington, DC, USA*, 2004. ACM 1581138075/04/0006.