Multi-Level Test Models for Embedded Systems

Abel Marrero Pérez, Stefan Kaiser
Daimler Center for Automotive IT Innovations
Technische Universität Berlin
Ernst-Reuter-Platz 7, 10587 Berlin
abel.marrero@dcaiti.com, stefan.kaiser@dcaiti.com

Abstract: Test methodologies for large embedded systems fail to reflect the test process as a whole. Instead, the test process is divided into independent test levels featuring differences like the functional abstraction levels, but also similarities such as many functional test cases. Desirable instruments such as test front-loading feature a considerable test effort and test cost reduction potential, but their efficiency suffers nowadays from the strict separation of the test levels and the consequent lack of appropriate mechanisms for reusing tests across test levels. Multi-level test cases have shown to provide the means for a seamless test level integration based on test case reuse across test levels. This paper extends this concept by introducing multi-level test models which are capable of systematically integrating different functional abstraction levels. From these models, we can derive multi-level test cases that are executable at different test levels. With this novel approach, multi-level testing benefits from the principles of model-based testing while the requirements for providing multi-level capabilities to any test models are analyzed and described.

1 Introduction

The complexity of large embedded systems leads to particularly long development and testing cycles. The long time span between the availability of the first software components and the first system prototype (e.g. an automobile) highlights the importance of testing earlier on in the development process. In this context, the introduction of test front-loading is essential, since the sooner a fault is revealed, the cheaper its correction becomes.

Test front-loading addresses the earlier execution of test cases conceived for higher test levels. For instance, a system test case might be executed on a software component. Common lifecycle models such as the V model do not explicitly consider front-loading. Moreover, their strict separation of test levels constitutes an obstacle for systematic and efficient test front-loading.

We thus call for ending this strict separation by integrating test levels. With test level integration we do not aim at merging test levels but at bringing them closer together. The idea is to establish relations between test levels that support a systematic test front-loading. For this purpose, we need a new test methodology that takes the whole V model into account rather than single test levels. An additional objective is to reduce the effort necessary for test specification, design, implementation and maintenance.

Effort reduction for these test activities has become a major issue after the notable advances in test automation of the last decade. Much of the optimization potential of test automation has now been tapped. In contrast, new methods focusing on optimizing the mentioned testing activities promise significant efficiency gains. We focus on these activities and pay particular attention to test level integration. For testing embedded systems, we expect to additionally benefit from new approaches in this field due to the fact that more test levels exist for testing embedded systems than for testing standard software.

We have proposed multi-level test cases as an instrument for supporting functional test case reuse across test levels as well as test level integration (cf. [MPK09a]). Such test cases consist of two parts. Firstly, a test level-independent test case core that is reused across test levels. Secondly, test level-specific test adapters. The structure of multi-level test cases supports the separation of commonality and variability across test levels.

In this paper, we introduce multi-level test models as an extension of this concept. With this approach we take advantage of the numerous synergies within a test suite in order to create abstract test models from which concrete multi-level test cases can be derived. With this novel approach we aim at benefiting from model-based techniques as well as further reducing the test effort, particularly with regard to the aforementioned test activities.

The headlights example presented in the next section will serve as an example throughout this paper. Section 3 details the reasons for applying reuse techniques across test levels. A brief overview of multi-level test cases follows before multi-level test models are introduced in Section 5. We conclude the paper with a related work overview and a summary.

2 Example

Our running example concerns the headlights function of a modern vehicle. As shown in Fig. 1, four different electronic control units are involved: the driver panel (DP), the ignition switch control (ISC), the light sensor control (LSC) and the automated light control (ALC). The ALC is the master component. It decides whether to turn on the headlights depending on the light switch position, the ignition switch position and the outside illumination. This information is received via a CAN bus.

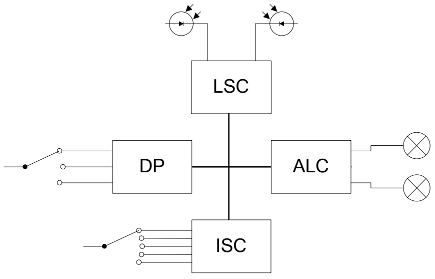

Figure 1: Electronic Control Units Involved in the Headlights Function

3 Test Reuse Across Test Levels

In the introduction we mentioned that the test process for embedded systems features additional test levels. We will consider four different test levels in this paper: software component testing, software integration testing, software/hardware integration testing, and system integration testing (cf. Fig. 2). In terms of functionality, software component testing is the most detailed test level, while system integration testing represents the most abstract one. These test levels reflect a large integration process starting with software components and ending with the complete system consisting of a set of electronic control units. We have described these test levels in-depth in [MPK09a].

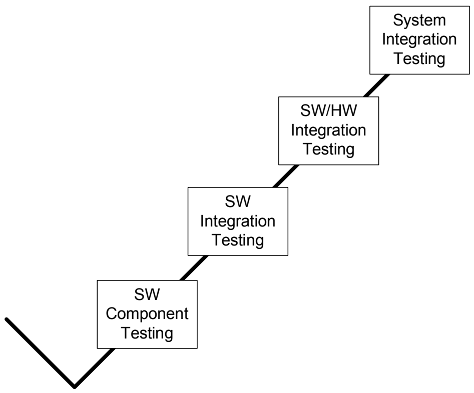

Figure 2: Right Branch of the V Model for Embedded Systems

The V model [SL07] treats each test level independently. The model recommends considering the development documents originated at the corresponding development level on the V model's left branch for creating test cases. This is a good practice since the functional abstraction levels of both development and test levels are the same. The V model does not consider test front-loading and further similarities between the test levels, however. We need test front-loading for an early assurance of the development artifacts' quality.

Our running example provides evidence of the significance of front-loading. A basic test case for the vehicle's headlights might consist of two steps: turning the headlights on and off using the light switch. During development, this test case can be executed once the first vehicle prototype has been built using the light switch and visually observing whether the headlights react. The prototype will be built relatively late in development, so that finding a fault in the prototype that could have been found earlier would be a costly exercise.

The idea of front-loading is to start executing this test case as soon as possible. We could take the electronic control unit driving the actual lamps and check if the headlights can be turned on and off. We might not have a switch available at this test level if the switch is wired to another control unit. Hence, we might have to stimulate a CAN bus at software/hardware integration test level. We can port the test case further down towards the software. We will find a software component including the basic logic that decides whether to turn on the headlights depending on the switch position. In fact, if the software is developed model-based, we can even test these functional models, as well. With this practice, which intrinsically considers test case reuse across test levels, we will be assuring at every test level that the headlights will later work based on the components integrated so far.

Systematically applied front-loading leads to revealing only those faults at every test level that are directly related to the integration step performed by that test level. If the headlights do not turn on at software/hardware integration test level, but did at software integration test level, there might be a problem with the hardware or its interface to the software.

A test case might specify that the switch has to be turned on, but at most test levels no switch will be available. The test interface, i.e. the concrete information channels that are used for both stimulating and observing the test object, will typically be different for each test level, because at each test level there are different test objects. Consequently, only test cases that abstract from the test interface will really be reusable. This is the case, for instance, for informal textual test specifications. In other words, we can reuse the headlights test case presented above as long as we do not concretely specify which signals with which coding are used for test stimulation or observation.

Handling different interfaces is inevitable for test implementations, however. Hence, a dedicated approach for reusing test cases is necessary for practicing front-loading systematically and with low effort. Such an approach has to take the different test objects and test interfaces into consideration. Another fundamental difference between test levels is the functional abstraction level of the test objects. Software components are very concrete because they include many implementation details. On the other hand, system integration testing represents the most abstract test level.

Acquiring the capability of dealing with these reuse obstacles will report several advantages in terms of test effort reduction. Test reuse across test levels is essential for an efficient front-loading, but it also has the potential to reduce the test suites' size at the different test levels. Similar or even identical test cases across test levels could be reused and thus only implemented and maintained once.

In summary, we can state that it is desirable to apply front-loading in order to find faults earlier. This implies reusing test cases across test levels which is directly possible if the test interface is omitted (e.g. for textual test specifications). For test design and implementation there is a set of barriers mainly regarding the test interface that needs to be sorted out before these artifacts can be reused. We propose using multi-level test cases for this purpose.

4 Multi-Level Test Cases

We introduced multi-level test cases in [MPK09a] as an instrument for test level integration. Reusing a test case implementation across up to four different test levels instead of implementing separate test cases for each test level promises great effort reductions.

Front-loading requires additional test effort in order to find faults earlier. It is necessary to reduce this additional effort as much as possible in order to benefit from this practice.

Completely reusing test cases is impossible due to the differences in functional abstraction and test interfaces across test levels. Hence, the goal is to achieve the highest possible reuse percentage. For this purpose, we distinguish two different modules within test cases: a reusable test case core (TCC) containing the actual test behavior and a variable test adapter (TA) specifically designed for a concrete test level or even test object. Fig. 3 shows the structure of multi-level test cases.

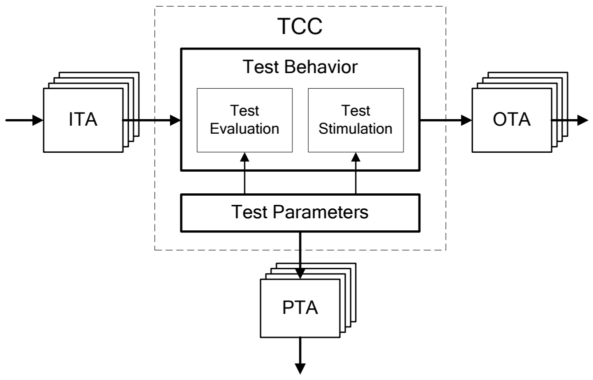

Figure 3: Structure of Multi-Level Test Cases

The test behavior in the TCC is invariant across test levels, i.e. test level-independent. It is abstract and can be reused across test levels. It possesses a stimulation interface and an observation interface for interacting with the system under test. The only form of variability accepted within the TCC are test parameters, which are necessary for bridging differences between test levels such as the level of accuracy (cf. software vs. hardware) or the kind of execution (e.g. real-time vs. simulation). These differences only imply minor variations within the test behavior that do not affect its intention.

In contrast, the TA varies across test levels and is thus test level-specific. It is responsible for bridging the (major) differences between the TCC and the test object at a certain test level. The test adapter consists of three parts: the input test adapter (ITA), the output test adapter (OTA) and the parameter test adapter (PTA).

Both ITA and OTA adapt the TCC interface to the test object interface. They thus map the abstract test behavior to the test object at its abstraction level. In the headlights example, reusing a system integration test case stimulating the light switch at software/hardware integration test level will require mapping the switch stimulation to an equivalent stimulation of the CAN bus. The OTA thus has to emulate the test case-related functionality of any components located between the switch and the CAN bus in the real system.

Both ITA and OTA basically have to model the behavior of components that are not available at a certain test level but whose presence is expected by the TCC. The behavior of such missing components does not necessarily have to be modeled completely – a partial behavior model covering the needed interface signals is sufficient (test case-related behavior). The PTA has to map any test parameters that apply to the test object.

Multi-level test cases can also be applied for reusing test cases from lower test levels at more abstract test levels. In this case, both ITA and OTA will have to model the inverse behavior of all components that are added along the integration process. In this context, more advanced mechanisms will have to be applied to maintain causality in case these additional components introduce delays. Delay balancing and delay compensation constitute a solution for this problem [MPK09b].

We can summarize that the key idea of multi-level test cases is to separate the test behavior into an abstract part within the TCC and an abstraction (ITA) or refinement (OTA) of this abstract behavior to a concrete test level featuring different test objects with different interfaces. This separation makes the reuse of major parts of the test cases possible, which is an essential requirement for efficient front-loading.

5 Multi-Level Test Models

Our model-based approach for designing multi-level test cases aims at creating a single multi-level test model for a concrete system function from which all multi-level test cases regarding that function can be derived. For example, instead of creating several multi-level test cases for testing the headlights, we create a multi-level test model and derive all multi-level test cases from it.

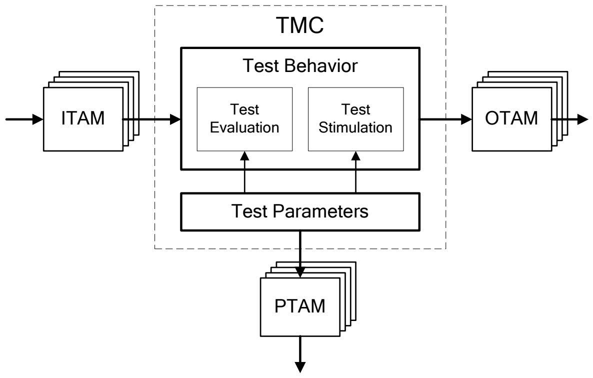

Figure 4: Structure of Multi-Level Test Models

The desired derivation of multi-level test cases from multi-level test models suggests maintaining the structure of the test cases for the test models. We thus propose separating the test model core (TMC) from the test adapter models (TAM) – as shown in Fig. 4. The TCCs will be derived from the TMC and the test adapters from the TAM.

This separation provides the same advantages as for multi-level test cases. For a concrete system function, only one test level-independent TMC has to be created. This TMC describes the test behavior and can be reused across all test levels. On the other hand, test level-specific TAMs have to be developed, i.e. single TAMs for each test level.

5.1 Test Model Core

While the TCC of multi-level test cases describes *partial* test behavior, the TMC will feature a more general behavioral description consisting of at least the composition of the test behavior of all test cases included in the textual functional test specification. Ideally, the TMC should describe the *complete* test behavior for a certain system function.

Note that test cases for a concrete system function will exist at different test levels. These test cases will belong to different functional abstraction levels. For instance, at the software/hardware integration test level there might be a test case stimulating a timeout of the CAN bus message containing information about the switch position. Furthermore, we assume that after $1\,s$ of not receiving this cyclic message, the CAN bus driver reports a timeout, and for safety reasons the headlights have to be turned on as long as the timeout persists. This test case is out of the system integration test level's scope because the handling of missing messages is too concrete for that test level. However, this test case can be reused at lower test levels (front-loading) in order to assure that the software reacts correctly to such a timeout before the first hardware prototype is available.

In consequence, the TMC will have to integrate test behavior at different functional abstraction levels. The alternative to this integration is to create a separate multi-level test model for each abstraction level. We opt for the integration approach in order to take advantage of the synergies between test cases across test levels. These synergies basically regard the test interface.

The TCCs of multi-level test cases created at different test levels often share part of the test interface. Some signals within the test interface are test level-specific. The switch message timeout represents an example of a test level-specific signal. Other signals are shared with higher abstraction levels, however. For example, the timeout test case mentioned above also observes the headlights, analogously to the basic headlights test case discussed in the previous sections consisting in turning the headlights on and off.

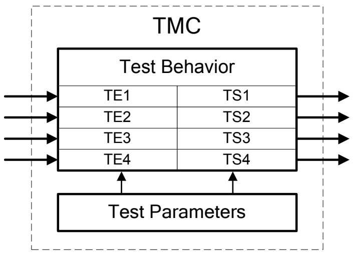

Figure 5: Division of the Test Behavior into Regions

We propose dividing the TMC into different regions – one for each abstraction level (cf. Fig. 5). Each region will be responsible for part of the TMC interface. Signals shared by different test levels will be assigned to the region corresponding to the most abstract test level. With this practice, the test behavior concerning a concrete signal is bundled in a single model region and does not have to be redundantly modeled in different regions.

For the derivation of a TCC at a certain abstraction level from a TMC, the region corresponding to the TCC abstraction level will be the most relevant. Furthermore, any more abstract regions might also be considered for *inheriting* additional signals. In contrast, less abstract regions will definitely be out of the derivation's scope. For instance, the basic on/off headlights test case introduced in Section 3 will be derived from the first model region only, since it belongs to the top test level. The timeout test case, however, will be derived from the first and second model regions. While the most relevant region will be the second, the headlights evaluation will occur in the first region, for example.

In some cases signals from different regions might need to be synchronized within test models. Appropriate mechanisms for performing this synchronization across regions have to be provided. By *appropriate* we mean effortless to a large extent, since this synchronization could otherwise constitute a disadvantage that counters the advantages of integrating the different abstraction levels of test behavior within a single test model.

5.2 Test Adapter Models

A test adapter model is a generalization of every possible test adapter for a concrete test object at a certain test level. Test adapter models are thus test behavior-independent. They only rely on the signals of the TMC interface and the TMC test parameters.

Test adapter models consist of three parts (cf. Fig. 4): an input test adapter model (ITAM), an output test adapter model (OTAM), and a parameter test adapter model (PTAM).

Our aim is to automatically generate the appropriate test adapter that maps a given TCC interface to a given test object interface from the corresponding test adapter model. This generation will be based on the interface of the system under test and the test object parameters. The generated test adapter will provide both the test object and the TMC with the necessary input signals and the test object with the appropriate parameter values.

The simplest ITAM will consist of a set of functions describing the input signals of the TMC as a function of the test object outputs. Analogously, a simple OTAM model will describe how to stimulate the test object inputs using the TMC outputs. These functions model any components lying between the interfaces being mapped.

An OTAM for the headlights example for the software/hardware integration test level will include a function mapping the switch position to a CAN bus message. This function partially models the behavior of the control unit attached to the switch. Additionally, it might contain other mappings such as that of the key position to the corresponding CAN bus message. For the timeout test case described above, we might need to map the switch position (for the beginning of the test case, before the timeout) but we might not need the key position, which might be needed for other test cases derived from the TMC. This will be taken into consideration for the test adapter generation. The generated test adapter will thus include the switch position mapping but exclude the key position mapping.

Fig. 6 shows an implementation example of an OTAM that can map any TCC at system integration test level to the interface of the ALC electronic control unit (which is at software/hardware integration test level). It is a MATLAB/Simulink [MLS] model consisting of two lookup tables that map the switch signals to the CAN bus coding and a moving average filter that models the behavior of the LSC with respect to the outside illumination. The LSC provides an illumination signal without short peaks by filtering the original sensor data. The OTAM thus has to model this filtering in order to provide the CAN bus signal. Note that single test adapters do not necessarily have to include a filter. If a concrete TCC stimulates the light sensor with a constant value, the multi-level test case does not require the OTA to apply a filter since the filter output will equal its input. In analogy, the complete lookup tables will typically be unnecessary for a single test case. These examples demonstrate that test adapter models are more robust against changes than test adapters.

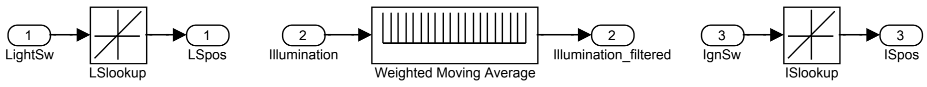

Figure 6: Output Test Adapter Model for SW/HW Integration Testing

Different test adapters can be generated from this OTAM. They will only differ in the signals used. For each test object input, the same test adapter will be used for any TCC because the OTAM is test behavior-independent, as we mentioned above. TAMs are hence similar to a mapping library containing different mappings ready for use.

Any test adapter generated from a test adapter model will correspond to a concrete test object at a certain test level. Mappings across multiple test levels can be achieved using more complex test adapter models but also concatenating multiple test adapter models mapping the interfaces of test objects at consecutive test levels. For example, once the switch position is mapped to the corresponding CAN bus message, we can reuse this mapping at software integration test level by mapping only the CAN bus message to the software variable representing the switch position. Test adapter model concatenation is particularly efficient for non-trivial mappings that require notable implementation effort as well as for frequently changing requirements and mappings (evolution). In those cases the concatenation provides a significant effort reduction (cf. test adapter concatenation in [MPK09a]).

5.3 Discussion

In this section, we have defined multi-level test models in a generic manner analogous to the multi-level test case definition. While multi-level test cases describe partial test behavior, multi-level test models cover the complete test behavior. The use of test level-specific TAMs leads to automatically adapting the TMC to the different test objects and abstraction levels along the test process. With this practice, the TMC can be reused across test levels without alteration – only test parameters might be subject to change.

Almost any test model can represent the TMC of a multi-level test model. We have found only four limitations to be mandatory in order to acquire the multi-level capability.

Firstly, multi-level test cases have to be derivable from multi-level test models. Secondly, updates within multi-level test models have to be automatically applied to the derived multi-level test cases. Thirdly, the TMC possesses a test interface consisting of a set of observation signals for test evaluation as well as a set of stimulation signals. Fourthly, the TMC has to cover different functional abstraction levels. The test interface is divided into different signal groups which are assigned to different regions within the test model. Each region corresponds to a certain test level.

This approach is deliberately kept generic, so that almost any test modeling approach can thus be employed for designing multi-level test cases. Dedicated tool support is only needed for the automatic generation of concrete test adapters from TAMs. A further criterion for the selection of a testing technology might be the support of test case portability across different test platforms at different test levels.

The presented approach takes advantage of all the benefits of model-based approaches such as better readability, reduced maintainability effort and a potentially increased functional coverage. Readability is crucial for front-loading and test reuse, since the resulting multi-level test cases will be shared by different parties at multiple test levels. Minimal maintainability effort is also of particular significance for large embedded systems due to the long development cycles which imply numerous changes. The functional coverage of multi-level test cases can increase by systematically modeling the overall test behavior.

On the one hand, a higher initial effort is typically required for creating multi-level test models in comparison to multi-level test cases – even in case there are large portions of commonality and synergies across test levels. On the other hand, once the models are created we can automatically generate concrete test adapters for any TCCs. Depending on the technology used, TCCs will also be automatically generated from the TMC.

The effort inversion in multi-level test models best pays off for large development cycles in which test cases and test objects are subject to continuous changes. The integration of different abstraction levels within the TMC is especially beneficial in terms of maintainability. For example, if the behavior of the headlights TMC input changes, it is sufficient to change one region of a single model in order to keep all multi-level test cases derived from that multi-level test model up-to-date.

6 Related Work

Front-loading has been proposed by Wiese et al. for early testing of system behavior in a model-based development context in the automotive domain [WHR08]. They suggest taking advantage of the early available functional models for testing system functionality. They also presented an approach for specifying test platform-independent test cases based on signal abstraction and a mapping layer. The use of abstraction for test case reuse is also proposed by Burmester and Lamberg [BL08]. Both approaches only consider reuse within a concrete abstraction level. However, they contribute notable ideas to our approach.

Beyond the front-loading context, Mäki-Asiala has denominated *vertical reuse* as the reuse of test cases across test levels [MA05]. He takes a pure reuse approach using TTCN-3

[TTC], i.e. he does not consider abstraction or refinement. He states that the similarities between test levels are fundamental for vertical reuse. We find that it is more important to assure that the differences are not substantial than completely focusing on the similarities.

In analogy to Schätz and Pfaller, Mäki-Asiala takes a bottom-up reuse approach and thus describes *interface visibility* as a central problem. Schätz and Pfaller solve this problem by automatically extending component test cases in order to gain system test cases that test the component at system test level [SP]. This test case extension is comparable to our test adapters. Schätz and Pfaller demonstrate that such test adapters can be automatically generated from formal component specifications. We cannot count on such accurate component specifications, especially for hardware components. Our approach therefore consists in approximating their behavior within test adapter models that focus on the complete test behavior instead of modeling the complete component behavior.

Our notion of test adapters is mainly based on the *adapter* concept introduced by Yellin and Strom in the context of component-based design [YS97]. The *platform adapter* and *system adapter* concepts within TTCN-3 are similar concepts in the testing domain. Lehmann (Bringmann) denominates *test engines* similar constructs within Time Partition Testing [LB03]. The main difference in comparison to our approach is that our test adapters are conceived for bridging abstraction differences across test levels.

The model-based approach for multi-level testing represents the major novelty within this paper. Model-based testing (MBT) nowadays constitutes a well-established research field and is gradually entering industry [Sch08], as well. Zander-Nowicka (Zander) provides an overview of MBT approaches paying particular attention to the automotive domain [ZNZ08]. Our approach is similar to the test models described in [PP05] or in [LB03]. In both approaches, test cases constitute partial behavior and are derived from abstract test models in the form of traces or variant combinations.

While reuse qualities are often attributed to model-based approaches [EFW02], to the best of the authors' knowledge no approaches have described a methodology for applying model-based testing for testing across test levels. The closest approach to ours in this context is TPT [LB03]. TPT features portability across test platforms [BK08].

7 Conclusion

This paper presents a novel model-based approach for systematically and efficiently testing large embedded systems at different test levels. Multi-level test models represent the model-based counterpart of multi-level test cases. We summarized the advantages of multi-level test models in Section 5.3. Our aim in this paper was to define a set of requirements that test models have to meet in order to become multi-level test models – independent of the modeling approach used.

We are currently applying multi-level test models to selected projects at Daimler AG using TPT. We plan on continuing this evaluation and reporting the results. In our future research we will focus on the automatic generation of test adapter models. We will also study efficient front-loading strategies that consider partitioning.

References

[BK08] Eckard Bringmann and Andreas Krämer. Model-Based Testing of Automotive Systems. In *1st International Conference on Software Testing, Verification, and Validation (ICST 2008)*, Lillehammer, Norway, 2008.

[BL08] Sven Burmester and Klaus Lamberg. Aktuelle Trends beim automatisierten Steuergerätetest. In *Simulation und Test in der Funktions- und Softwareentwicklung für die Automobilelektronik II*, pages 102–111, Berlin, Germany, 2008. expert.

[EFW02] Ibrahim K. El-Far and James A. Whittaker. Model-Based Software Testing. In John J. Marciniak, editor, *Encyclopedia of software engineering*. Wiley, New York, NY, USA, 2002.

[LB03] Eckard Lehmann (Bringmann). *Time Partition Testing*. PhD thesis, Technische Universität Berlin, 2003.

[MA05] Pekka Mäki-Asiala. *Reuse of TTCN-3 Code*, volume 557 of *VTT Publications*. VTT, Espoo, Finland, 2005.

[MLS] The MathWorks, Inc. - MATLAB/Simulink/Stateflow. http://www.mathworks.com/.

[MPK09a] Abel Marrero Pérez and Stefan Kaiser. Integrating Test Levels for Embedded Systems. In *Testing: Academic & Industrial Conference - Practice and Research Techniques (TAIC PART 2009)*, pages 184–193, Windsor, UK, 2009.

[MPK09b] Abel Marrero Pérez and Stefan Kaiser. Reusing Component Test Cases for Integration Testing of Retarding Embedded System Components. In *First International Conference on Advances in System Testing and Validation Lifecycle (VALID 2009)*, pages 1–6, Porto, Portugal, 2009.

[PP05] Alexander Pretschner and Jan Philipps. Methodological Issues in Model-Based Testing. In Manfred Broy, Bengt Jonsson, Joost-Pieter Katoen, Martin Leucker, and Alexander Pretschner, editors, *Model-Based Testing of Reactive Systems*, volume 3472 of *LNCS*, pages 281–291. Springer, Berlin, Heidelberg, Germany, 2005.

[Sch08] Hermann Schmid. Hardware-in-the-Loop Technologie: Quo Vadis? In *Simulation und Test in der Funktions- und Softwareentwicklung für die Automobilelektronik II*, pages 195–202, Berlin, Germany, 2008. expert.

[SL07] Andreas Spillner and Tilo Linz. *Basiswissen Softwaretest*. dpunkt, Heidelberg, Germany, 3rd edition, 2007.

[SP] Bernhard Schätz and Christian Pfaller. Integrating Component Tests to System Tests. *Electronic Notes in Theoretical Computer Science (to appear)*.

[TTC] ETSI European Standard (ES) 201 873-1 V3.2.1 (2007-02): The Testing and Test Control Notation version 3; Part 1: TTCN-3 Core Language. European Telecommunications Standards Institute, Sophia-Antipolis, 2007.

[WHR08] Matthias Wiese, Günter Hetzel, and Hans-Christian Reuss. Optimierung von E/E-Funktionstests durch Homogenisierung und Frontloading. In *AutoTest 2008*, Stuttgart, Germany, 2008.

[YS97] Daniel M. Yellin and Robert E. Strom. Protocol Specifications and Component Adaptors. *ACM Trans. on Programming Languages and Systems*, 19(2):292–333, 1997.

[ZNZ08] Justyna Zander-Nowicka (Zander). *Model-based Testing of Real-Time Embedded Systems in the Automotive Domain*. PhD thesis, Technische Universität Berlin, 2008.