Systematic Test and Validation of Automotive Systems

Mugur Tatar, Jakob Mauss, Andreas Junghanns
QTronic GmbH, Germany

Abstract—**We focus on issues related to the test and validation of complex embedded systems. These are systems that include, besides software controllers, also the controlled physical subsystem, often a mixture of hydraulic, mechanical and other physical components. For validation purposes also the interaction with the driver and the environment has to be taken into account. However, validating system-level behavior for such systems is not supported well by most of the existing test and validation methods - consider, for instance: (a) model-checking techniques do not scale to analyze complete systems, while (b) manual test automation does not scale to cover the huge number of situations relevant for a system test. Our method, implemented by TestWeaver, combines simulation and intelligent search to investigate system behavior in a large space of situations automatically. Starting from a compact specification of the relevant search space and goals, TestWeaver is able to generate autonomously thousands of differing scenarios. These are executed using simulation and the results are fed-back to reactively guide the further test with the goal to: (a) maximize the coverage of the relevant system state space, and (b) identify hidden bugs and weaknesses in the design or implementation. We discuss several applications of the method and results from the power-train and driver assistance domain.**

I. Introduction

MORE and more vehicle functions are controlled or affected by software. The behavior at system level emerges from the tight interaction among control software, mechanical, hydraulic, electronic and other physical subsystems, as well as driver and environment. This resulting behavior is extremely difficult to foresee and leads to a new kind of complexity which is difficult to master by the development engineers. Traditional methods, we argue, do not support well the test and validation at system level, new methods and tools supporting the system validation are needed. We present a method, based on virtual system integration, simulation and intelligent test generation, that addresses specifically challenges raised when testing and validating system level behavior.

The paper is organized as follows: Section II discusses challenges of validating system-level behavior. Section III reviews existing test and validation methods and their limitations. Section IV introduces the key ideas of our approach. Virtual integration of vehicle systems, a prerequisite for simulating large systems, is discussed in Section V. Section VI presents TestWeaver - a tool for intelligent test generation and test automation for simulated systems. Several application results from the automotive domain are presented in Section VII.

II. Challenges of System Test and Validation

The number of electronic control units (ECUs) in vehicles is continuously increasing, the complexity of the software running on them is increasing as well. In a typical ECU the software modules are contributed by several teams and organizations, precise details about their function or source code most often do not cross the borders of their supplying organization. Furthermore, complex functions may be distributed over several ECUs, for instance: electronic stability control (ESC, ESP) and active body control (ABC), or battery management, transmission and engine control in a hybrid power-train. Problems caused by bugs in the implementation or by incomplete or incorrect specifications may lead to unintended behavior that is difficult to spot and secure against. Even in the improbable case when all relevant software specifications and source code is accessible to an engineer, the system behavior that results in the interaction of a controller with the physical world is not easy to assess. While software in isolation can, theoretically, be modeled and analyzed using discrete models, the controlled physical system and the environment cannot be in general approximated using discrete models. For certain tasks and limited scopes this might be possible with considerable effort, but the results cannot be reused or generalized for other tasks - the method has a limited reliability and does not scale.

Due to above considerations most often the task of system test and validation can be approached only late in the product development cycle. Physical prototypes, either complete vehicles, or subsystems connected to test-rigs, as well as Hardware-in-the-Loop (HiL) simulation platforms that test ECU prototypes are used to investigate system-level behavior. Whether on the road, or with such a test-bench or HiL simulation, the challenge remains to test the vehicle behavior in the huge number of situations that are relevant:

1) *operating states*: the controller and the controlled system may have many distinct operating states and transitions that have to be covered
2) *vehicle configurations*: certain subsystems are embedded in many differing vehicle configurations. Not only the logical specifications need to be proofed, also the concrete configuration parameters have to be checked, for each configuration
3) *component faults, parameter tolerances*: on-board monitoring, diagnosis and recovery have to work properly. System safety has to be ensured, as well as

graceful performance degradation due to aging, etc.

4) *driver controls*: the driver may interact in many different ways with the vehicle, some cases may be rare or even obscure - but the system should stay safe
5) *environment*: intelligent controllers react to street, wind and other changes in the environment. System tests need to ensure that the reactions occur when they are intended to occur, i.e. are neither missing, not spuriously generated without a reason.

The above dimensions span a huge combinatorial space of situations that must be considered when validating systems. Without applicable formal methods, in practice, the goal of covering the entire space is approximated by considering a (small) finite number of test cases.

III. Test and Validation Methods in Use

We shortly review here some of the methods used in the automotive industry for test and validation of systems, sub-systems and modules, as well as some of their strength, costs and limitations. A detailed or comprehensive analysis of all used methods is out of the scope of this paper.

A. *Tests using physical prototypes vs. simulation*

Physical prototypes include test-rigs or complete vehicle prototypes. They play currently in the industry a crucial role for parameter identification, integration and system tests, tuning and ultimate system validation. The test using physical prototypes has the drawback that it is very expensive, it can be performed only in late development stages and, therefore, only a limited number of development-test-change cycles can be performed. The trend is to "front-load" as much as possible from the activities that run using physical prototypes in earlier development and integration stages, for instance using virtual / simulated environments - since this allows faster and cheaper development and test cycles. Virtual environments replace parts of the physical subsystems / components with simulated ones and can range from (a) purely simulated environments, using Model-In-the-Loop (MiL) and Software-In-the-Loop (SiL), to (b) environments that mix simulated subsystems and real components, such as Hardware-In-the-Loop (HiL) simulation, where a real ECU is connected to a real-time simulation of the surrounding physical system. Since virtual environments enable earlier testing and easier automation, thus more comprehensive testing, the rest of the paper deals with testing methods that apply to such environments, i.e. MiL, SiL or HiL.

B. *Module Tests vs. System Tests*

Not surprisingly, many development and test methods for software controllers are derived from "classical" software development and test methods. As such, they assume that the development of a software controller is guided by requirements for software modules that are specified prior to the implementation and is followed by thorough testing of those software requirements. In practice, for many embedded systems, it is seldom the case that one is able to develop correct and complete module requirements prior to experimentation at system level. The requirements at module level need to be validated - and this is only possible if the *system* requirements and the interaction of the controller with the controlled system are thoroughly tested. Only afterwards, the module requirements can be assumed correct. Test methods that apply only to software alone, no matter how powerful and complete at that level, are not enough for ensuring that the system "is correct" and is working "as intended". Effects such as non-converging controllers, system safety, controllability, fault reactions, etc. cannot be assessed at module level. In effect, if not covered in simulation, the system level problems will be discovered too late, at best during integration and test with physical prototypes, or even later, by customers on the street. Conclusion here: development processes that claim to ensure system correctness and quality need to offer a high automation and coverage of the tests at the system level, the module level alone is insufficient.

C. *Formal and Semi-Formal Methods*

We consider here several methods that can guarantee high degrees of coverage of the test results or even completeness. of certain analysis tasks, for instance: static code analysis, model-checking and model-based test generation. All methods from this category, known by the authors to be used in an industrial context, are restricted to the analysis of discrete systems, e.g. some kind of state automata. As such they are appropriate to describe and analyze properties of software modules, but are less appropriate to analyze system-level properties. For instance, mechanical and hydraulic sub-systems, street, wind and other components that need to be included in the system-level analysis cannot be described in general with discrete models. Discrete approximations of continuous behavior, when used in certain contexts, are difficult to generate reliably and have limited or no reusability at all. Therefore, we conclude that these formal methods are only applicable in industrial context to the analysis of software modules and do not cover the system-level requirements.

Static Code Analysis: This applies to the analysis of source code, for instance C-code and can pinpoint several implementation problems: divisions by zero, integer overflows, access violations, and others. Very useful at the code level. Limitations: the method also finds potential problems, i.e. cases that cannot be automatically classified as relevant problems. Many potential problems can be (and will be) ignored after revision by an engineer - but a failure in this revision leaves hidden bugs unsolved.

Model-Checking: Formal analysis of discrete systems. Can prove presence/absence of certain behavior (state reachability), can find counter-examples that can drive the model into a state that should have been unreachable

according to the specification. Of limited use at system level.

Model-Based Test Generation: Based usually on analysis of discrete models, for instance state machines or models derived from source code analysis. Can generate test sequences that guarantee a high coverage, for instance of the model states, of the transitions, and so far. Of limited use at system level.

D. Back to Back Tests - Model vs. ECU

Compare simulation traces from MiL or SiL to traces that use the target ECU processor and pinpoint differences that require manual revision. They are used as a "proof" that model properties can be transported to properties of the target ECU component. This method is often used as an add-on to model-based test generation methods. Limitation: if the model-based test generation neglected system level requirements, so will the back-to-back ECU test do.

E. Human-Specified Tests + Test Automation

In short, this is usually known as "Test Automation": test scenarios are specified in a scripting language (e.g. Python) or a graphical notation. A test automation tool translates the specifications in code that stimulates and checks pass/no-pass conditions on MiL, SiL or HiL simulation platforms. The method applies for the module-level, as well as, for the system-level testing and is the most used test and validation method nowadays. At system level, as argued before, there are extremely many situations that require testing - the manual test production and maintenance is the bottleneck and does not scale with increasing system complexity.

Very often manual test production is used in conjunction with *requirements coverage* as a method to assess how many tests to develop. System requirements must be documented, for instance using statements such as "WHEN <precondition> THEN <post-condition>", at least one scenario should test each requirement. Common practice is to develop the test scenarios such that: a test scenario generates a stimulus that drives the system in a state where the precondition of a certain requirement is satisfied. In that state (and only in that state) the requirement post-condition is evaluated to true or false as test success/failure condition. A particular weakness of this approach is that it ignores the general nature of the requirements: the requirements cannot be tested in arbitrary points of the state space, although they are valid in the whole state space (when taking the preconditions into account). A better strategy implements requirement checking as system invariant observers - this allows a continuous tracking of all requirements at all times during a simulation.

F. Test Coverage

This paragraph does not address a particular test method, but rather the methods used to measure what has been tested, what not. Coverage measurements are also used in practice to decide when "enough tests" have been developed. In particular, that last question is very difficult to answer in the system context: The number of system states is infinite, complete formal methods are not applicable, testing scenarios is incomplete. Any decision to stop testing is arbitrary, since system correctness cannot be guaranteed (except for particularly simple systems, which we do not discuss).

Coverage methods often used in the industrial practice are: (a) source-code coverage - e.g. depicts which code branches have been executed, which not, and (b) requirements coverage - which system requirements have been "touched" by some test, which not. While both coverage measurements deliver useful information, we believe that they do not measure sufficiently well the requirements of a "good" *system test*. A good coverage measurement at system level should assess the coverage of the states of the *complete system*: this includes the software controller, the physical controlled subsystem, the driver controls and any other inputs or states of the environment model, if possible also transitions between important operational states. Moreover, the "measurement" should be meaningful and easy to understand for an engineer. While such measurements seem to be difficult to define in general, we made a good experience with customizable system state coverage measurements (explained in more detail later in the paper): engineers can select system state variables, the coverage measurement shows which states (and transitions) have been reached during a test, which not.

IV. Key Ideas of our Approach

Motivated by the above discussion, we list here the key ideas of our approach for system testing:

- virtual system integration as a platform for system test - this assumes a closed loop simulation of the controller(s) with the relevant surrounding system model, the simulation platform can be based on MiL, SiL or HiL
- modeling of the system (and module) requirements as observers of system invariants - this allows to detect problems in any simulation scenario. In a similar way additional observers of quality criteria can be defined
- automated and intelligent generation of simulation scenarios, following the goals to: (a) increase the coverage of the system state space, and (b) detect violations of the correct behavior and worst-cases for quality criteria
- neat integration of automatic test generation with human-specified test scenarios, as well as with scenarios defined via measurements of test drives
- customizable, easy to understand, visualization of state coverage measurements - as add-on to the commonly used requirement coverage and source-code coverage measurements
- reproducible results: each state reached during some test, whether correct or incorrect, can be reproduced in simulation ("replay" function) as often as necessary and

analyzed in detail until the cause is found

- black-box test: access to the source code of the controllers and plant models is not required - this accounts for the fact that systems integrate modules from different suppliers and source code is not shared across organizations
- ensure that the simulated system uses exactly the same ECU calibration parameters as the tested vehicle configuration. The configurable parameter values must be tested as well, since a single bad parameter value can have in certain situations catastrophic effects.

V. PLATFORMS FOR VIRTUAL SYSTEM INTEGRATION

In principle, any closed-loop system simulation can be used with our testing method, whether MiL, SiL or HiL. However, there are trade-offs to consider when planning which kind of simulation platform to use for a certain kind of test. While MiL and HiL have been used maybe more often in industry until now, we believe that in future the SiL platforms will play a more prominent role for system test and validation. A short look at the platform trade-offs relevant in this context:

Accuracy of the controller simulation. Of course, here the HiL platform gets the best possible score. The virtual (simulated) ECU used in SiL the next best, while the MiL simulation gets the worst score. Controller code simulated in MiL most often uses floating point arithmetic, while most automotive ECUs use fixed-point arithmetic - a source of many potential and real differences. For the high level controller functions the same generated source code for the real (HiL) and for the virtual (SiL) ECU can and should be used. Moreover, the real ECU calibration parameters can be "flashed" in the virtual ECU.

Accuracy of the physical system simulation (plant model). Here the HiL platform gets the worst score, MiL and SiL platforms get a much better score. On HiL platforms the plant model must run in real-time - this constraint leads to extreme simplifications of the physical effects that can be simulated.

Speed of simulation. This affects the test throughput discussed separately below. MiL platforms are often based on model interpretation. SiL and HiL platforms use compiled code and are, in effect, faster. While HiL platforms never run faster than real time, SiL on conventional PCs is often significantly faster than real-time. We have seen realistic models running 20-30 times slower than real-time in MiL, running, of course, in real time on HiL, and running 10-30 times faster than real time in SiL.

Costs and platform availability. HiL platforms are expensive to set-up and to maintain. Pure virtual MiL and SiL setups running on conventional PCs can be copied and instantiated any number of times at no additional costs. A single PC with a multi-core processor can run several simulations in parallel, without performance degradation.

Debugging and analysis comfort. MiL and SiL simulations can be stopped at any time, stepped, connected to source code debuggers. The simulation is 100% deterministic. None of above apply to HiL simulations that include real hardware in the loop.

Test throughput. This is actually directly connected with the test coverage that can be achieved for some system in a given time. By far, due to simulation speed, platform costs and availability, the best score for system tests goes to the SiL platform.

While certain tests will have to run also on HiL platforms, we believe that the large amount of tests required for a good system test will be using pure virtual test platforms, namely SiL on conventional PCs.

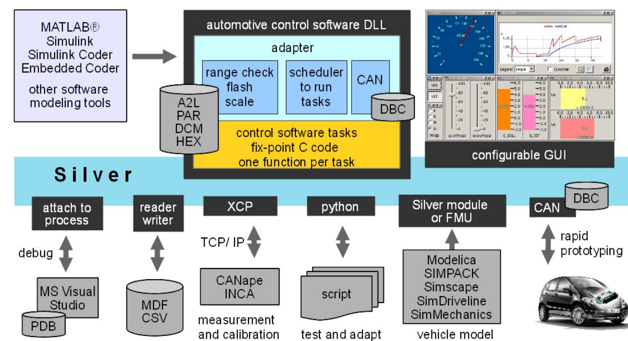


Fig. 1. Silver: virtual vehicle integration, simulation and rapid control prototyping on Windows PC. Silver co-simulates several compiled modules, either virtual ECUs, plant model components, or other utility modules.

Figure 1 depicts Silver, our preferred virtual integration platform based on SiL on Windows PC, for details see [2]. Silver integrates several compiled modules representing virtual ECUs and plant model component. Controller code can be imported from Simulink, Embedded Coder, TargetLink, Ascet or hand-coded C. Virtual ECU integration is supported by the Silver Basic Services: these emulate RTOS and schedules time-triggered or event-triggered controller tasks, emulate DBC-configured CAN communication, support ASAP2/A2L, PAR, HEX, DCM configured calibration via XCP. Measurement and calibration tools such as CANape and INCA can connect to the virtual vehicle as if it were a real vehicle. Plant models can be imported form several modeling tools: Simulink, Dymola, AMESim, SimulationX, Simpack and others. Moreover, the new FMU standard for model exchange and co-simulation [3] is supported. Other features contribute to providing a powerful and comfortable environment for experimentation, test and debugging: configurable GUI, recording and replaying measurements, manipulating signals for fault simulation, scripting with Python, connection to source-code debuggers and code-coverage measurement tools. Silver is in use for virtual system integration and test of control software at AMG, Mercedes-Benz, BMW, IAV, Continental and others.

VI. Automatic Exploration of System States

We present here our method for an automatic system test and validation. The method, implemented by TestWeaver, combines simulation and intelligent search to investigate system behavior in a large space of situations automatically. Module and system requirements, as well as other quality criteria, must be implemented as observers that enhance the system simulation and report alarms when violations are detected. This allows a correctness and quality assessment in any state reached during the simulation. Human defined test scenarios, or scenarios defined by measurements during real drives can be integrated, but are optional. They are complemented by the large number of test scenarios automatically generated by TestWeaver.

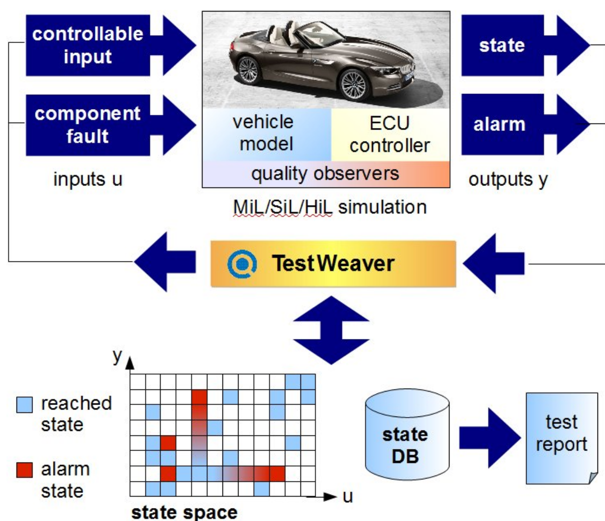
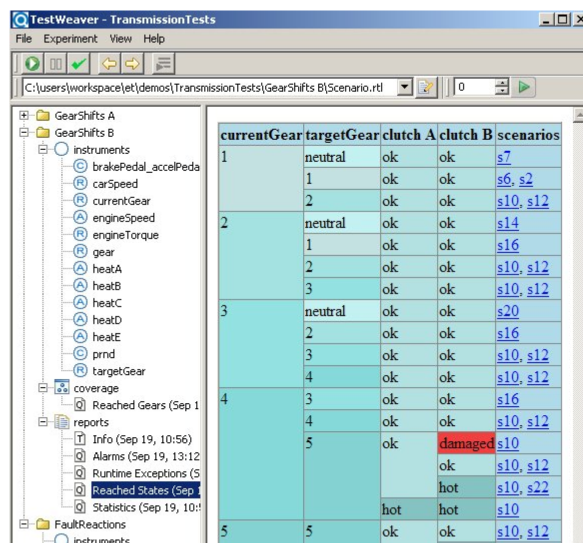


Fig. 2. TestWeaver: autonomous generation and evaluation of test scenarios with the goals to: (a) drive the system into states that have not reached before, and (b) find violations of correctness and worst-cases of quality indicators

Figure 2 depicts TestWeaver. TestWeaver connects to a SiL, MiL or HiL system simulation, controls inputs and observes correctness / quality indicators, as well as other selected state variables stemming from the controller or from the plant model. The communication between TestWeaver and the simulation is done at discrete time points, either time-triggered, or event-triggered. At these time points snapshots of the inputs and of the observables are communicated to TestWeaver. This information is classified and stored in a large state database to document the state coverage and to guide the further scenario generation. For this purpose continuous variable domains are partitioned in a finite number of intervals - for the state coverage assessment these intervals are regarded as "equivalence classes". Note, the behavior of the system under test (SUT) continues to take place in the mixed discrete-continuous state space of the simulation, the classification of the continuous states is done only in order to allow an overview of the state coverage and to build an abstract state model that guides the scenario generation. Each simulated scenario, corresponding to a trajectory in the discrete-continuous state space, is mapped by this classification to a trajectory in the abstract state space maintained by TestWeaver. All reached states can be classified in good or bad according to the correctness/quality observers. Furthermore, all reached states can be later reproduced exactly in simulation any number of times ("replay" function) - each state is associated with at least one scenario that reached that state. This allows a detailed analysis and debugging of the discovered problems.

The state database can be queried at any time during an experiment with a language similar to SQL, the resulting tables are converted to linked HTML reports. This allows engineers to easily produce projections of the complete state space on different sub-spaces for a better overview - see figure 3 for an example. The coverage reports that the engineer builds for his information, are used by TestWeaver during the scenario generation to focus the search on those state regions that the user is interested in. Besides the state coverage information built this way, TestWeaver uses also a mixture of other heuristics, such as: systematic input domain coverage, gradient strategies, or random search at some times. User specified tests can be used as seeds for the scenario generation as well.



currentGear	targetGear	clutch A	clutch B	scenarios
1	neutral	ok	ok	s7
	1	ok	ok	s6, s2
	2	ok	ok	s10, s12
2	neutral	ok	ok	s14
	1	ok	ok	s16
	2	ok	ok	s10, s12
	3	ok	ok	s10, s12
3	neutral	ok	ok	s20
	2	ok	ok	s16
	3	ok	ok	s10, s12
	4	ok	ok	s10, s12
4	3	ok	ok	s16
	4	ok	ok	s10, s12
	5	ok	damaged	s10
			ok	s10, s12
			hot	s10, s22
		hot	hot	s10
5	5	ok	ok	s10, s12

Fig. 3. Fragment from a custom state coverage report. It shows the gear shifts reached during the test of a transmission system, associated clutch temperature classes (ok/hot/damaged) and links to scenarios that can be used to drive the system those states.

The state database stores, besides the observed states and transitions, also the duration of each observed state transition. By post-mortem processing of the state database further quality and statistical information can be extracted: such as longest and average duration of gear shifts, toggling discrete control signals, missing or too late reactions.

TestWeaver can be connected to various simulation platforms via libraries that instrument controllable and observable signals: for Simulink a connection block-set is supplied, for Dymola a Modlica library is provided, for Silver and dSpace HIL systems a Python library is provided, for other environments a C library is provided.

VII. APPLICATIONS

TestWeaver is used for repeated system test and validation during development. We review here application results for several complex power-train and driver assistance systems.

In many applications TestWeaver is used in connection with Silver as virtual integration platform. Besides providing a comfortable PC-based simulation and debugging environment, with a realistic ECU simulation and parameterization, Silver provides built-in support for monitoring the min-max ranges of all ECU signals. The standard ASAP2/A2L ECU configuration information, supported by Silver, provide min-max information for all ECU signals. A typical ECU may contain over 10000 signals - these are monitored by Silver in each ECU execution cycle, violations are automatically reported to TestWeaver.

In [1], [9] and [10] application results for testing automatic transmission systems for classical ATs, as well as various DCTs at Mercedes-Benz and AMG, are reported. Besides classical software bugs (division-by-zero, run-time exceptions) and A2L-range violations, several other system specific correctness and quality criteria have been monitored: diagnostic trouble codes set by the ECU on-board diagnosis, toggling control signals, too long shifts, overspeed of engine or transmission internal components, too high clutch temperatures, mismatches between the plant model state and the state estimated by the controllers, transmission in unintended open or blocking states. Most of these problems are only visible at system level and cannot be detected by module tests alone.

In [8] results of testing a DCT system developed by GIF are reported. Here TestWeaver has been used in connection with a MiL vehicle simulation under Simulink.

In [4] TestWeaver and Silver have been used for automated test and validation of a safety-critical chassis-control system, namely a brake assistant system for heavy trucks. Here the system safety and robustness with respect to faults in the actuators, sensors, network communication, as well as various drive conditions and parameter tolerances had to be assured.

In [6] TestWeaver has been applied in a HiL context to the test of a heavy-machinery ECU. Here the results of a case study, comparing the results of 250 hand coded software tests with the tests automatically generated by TestWeaver are reported. The hand coded tests have been developed with the goal to identify some difficult to reproduce errors reported from the field. For developing the hand coded tests an effort of more than three person-month has been spent - however, without success: the field reported errors could not be reproduced. On the other side, with only a fraction of that effort, TestWeaver has been able to detect the errors that have not been revealed by the manual tests.

In [5] TestWeaver and Silver have been used to validate the cross-wind stabilization function for Mercedes-Benz S-Class cars. The function detects sudden crosswind and compensates it through the actuators of the Active Body Control (ABC). The validation of the stabilization function, as reported in , has been conducted by a single engineer (a novice TestWeaver user at that time) within about three weeks. In that time, about 100.000 different driving scenarios with different wind and road conditions, each 45 sec. long, have been generated, executed by simulation and validated using TestWeaver. The setup has been changed and extended during the investigation to explore also the effect of sensor faults. The test coverage achieved this way would have been hard, if not impossible, to achieve with comparable effort using a less automated approach, e. g. based on hand-written test scripts, driving a real car on the road, or using the Daimler crosswind test facility. As Daimler engineers finally said, the TestWeaver approach seems extremely well suited for the validation of complex automotive controllers during all stages of development.

In [7] application results for testing control software for hybrid drive trains at ZF are reported. The hybrid system tested there has a significant complexity and involves control functions distributed across several ECUs. As platform for virtual vehicle integration and simulation the ZF proprietary tool SoftCar was used. The connection between SoftCar and TestWeaver was done via the C-interface provided by TestWeaver. SoftCar supports co-simulation several virtual ECUs and plant model components using SiL, in a similar way as Silver does. A remarkable aspect of this project was the systematic connection of the system requirements, documented in DOORS in this project, with the over 100 correctness observers implemented for the test with TestWeavers.

VIII. CONCLUSION

We have discussed the challenges raised by the system test and validation of automotive systems, a task that requires to consider the interactions among very complex controllers, vehicle, driver and environment. We have suggested an approach to this task that is based on a combination of closed-loop system simulation and autonomous, intelligent exploration of the reachable system states. The method is implemented by TestWeaver and is in use since several years by several major car makers and suppliers.

The main benefit of the presented method, as assessed by several application reports that have been published until now, is the high test coverage that can be achieved with a relatively low work effort required for the test specification. With the help of TestWeaver not only implementation bugs in software modules can be found - also inconsistencies or "wholes" in the specification of the module/system requirements can be (and have been) found.

REFERENCES

[1] Brückmann, Strenkert, Keller, Wiesner, Junghanns, “Model-based Development of a Dual-Clutch Transmission using Rapid Prototyping and SiL", in *Proc. Intern. VDI Congress Transmissions in Vehicles*, Freidrichshafen, Germany, 2009.

[2] A. Junghanns, "Virtual Integration of Automotive Hard- and Software with Silver", unpublished. Available: http://www.qtronic.de/doc/SilverIntro.pdf
[3] *Functional Mock-up Interface for Model Exchange and Tool-Coupling,* specification. Available: http://www.functional-mockup-interface.org/
[4] M. Gäfvert, J. Hulten, J. Andreasson, A. Junghanns, J. Mauss, M. Tatar, "Simulation-Based Automated Verification of Safety-Critical Chassis-Control Systems", in *Proc. 9th Intern. Symposium on Advanced Vehicle Control,* Kobe, Japan, 2008. Available: http://www.qtronic.de/doc/TestWeaver_AVEC08.pdf
[5] K.D. Hilf, I. Mattheis, J. Mauss, J. Rauh, "Automated Simulation of Scenarios to Guide Development of a Crosswind Stabilization Function", in *Proc. 6th IFAC Symposium Advances in Automotive Control,* Munich, Germany, 2010. Available: http://www.qtronic.de/doc/TestWeaver_AAC2010_paper.pdf
[6] Th. Neubert, M. Tatar, "Extensive Test of Heavy-Machinery ECU on a NI VeriStand HiL using TestWeaver". Presentation at 14th ITI Symposium, Dresden Germany, December 2011. Available: http://www.qtronic.de/doc/TestWeaver_NIVeristand_ITI2011.pdf
[7] M. Neumann, M. Nass, C. Paulus, M. Tatar, "Absicherung von Steuerungssoftware für Hybridsysteme", *5th VDI AUTOREG Fachtagung Steuerung und Regelung von Fahrzeugen und Motoren,* Baden-Baden, Germany, November 2011. Available: http://www.qtronic.de/doc/TestWeaver_AutoReg2011.pdf
[8] N. Papakonstantinou, S. Klinger, M. Tatar, “Test-driven Development of DCT Control Software”, in *Proc. 8th Intern. CTI Symposium Innovative Automotive Transmissions,* Berlin, Germany, 2009.
[9] A. Rink, E. Chrisofakis, M. Tatar, "Automating Test of Control Software - Method for Automatic Test Generation", *ATZelektronik* 06/2009*,* vol. 4, pp. 24-27, 2009. Available: http://www.qtronic.de/doc/ATZe_2009_en.pdf
[10] M. Tatar, R. Schaich, T. Breitinger, "Automated Test of the AMG Speedshift DCT Control Software", in *Proc. 9th Intern. CTI Symposium Innovative Automotive Transmissions,* Berlin, Germany, 2010. Available: http://www.qtronic.de/doc/TestWeaver_CTI_2010_paper.pdf