# TestWeaver
# A Tool for Simulation-based Test of Mechatronic Designs

Andreas Junghanns, Jakob Mauss, Mugur Tatar
QTronic GmbH, Alt-Moabit 91d, D-10559 Berlin
{andreas.junghanns, jakob.mauss, mugur.tatar}@qtronic.de

## Abstract

The tight interaction among an ever increasing amount of software functions and hardware subsystems (mechanics, hydraulics, electronics, etc.) leads to a new kind of complexity that is difficult to manage during mechatronic design. System tests have to consider huge amounts of relevant test cases. Validation with limited resources (time and costs) is a challenge for the development teams. We present a new instrument that should help engineers in dealing with the complexity of test and validation. TestWeaver is based on a novel approach that aims at maximizing test coverage with minimal work load for the test engineer for specifying test cases. The method integrates simulation (MiL/SiL) with automatic test generation and evaluation, and has found successful applications in the automotive industry. We illustrate the approach using a 6-speed automatic transmission for passenger cars. We present also the way TestWeaver and Modelica simulators can work together.

*Keywords: test automation, mechatronic systems*

## 1 Introduction

When developing complex mechatronic systems, like a hybrid drive train or an automatic transmission for a vehicle, contributions from different engineering disciplines, design teams, departments, and organizations have to be integrated, resulting in a complex design process. Consequently, during development, design flaws and coding errors are unavoidable. For an OEM, it is then crucial that all those bugs and weak points are found and eliminated in time, i.e. before the system is produced and delivered to customers. Failing to do so may result in expensive recalls, high warranty costs, and customer dissatisfaction. OEMs have long realized this and spend up to 40% of their development budgets for test related activities. Software offers great flexibility to implement new functions, but also many hidden opportunities to introduce bugs that are hard to discover. Moreover, the complex behaviour that results from the interaction of software and physical systems cannot be formally and completely analysed and validated. Most often, it can only be evaluated in a limited amount of points with physical or virtual experiments. The development teams are often faced with a dilemma: on the one side, the system test should cover a huge space of relevant test cases, on the other, there is only a very limited amount of available resources (time and costs) for this purpose.

This paper presents a novel test method that has the potential to dramatically increase the coverage of testing without increasing the work load for test engineers. We achieve this by generating and executing thousands of tests automatically, including an initial, automated assessment of test results. The test generation can be focused on certain state spaces using constraints and coverage goals.

This paper is structured as follows: In section 2, we take a bird's-eye view on testing mechatronic systems. In section 3, we survey the main test methods used in the automotive industry today. Section 4 presents the proposed test method. Section 5 illustrates the proposed method using a 6-speed automatic transmission for passenger cars. Section 6 discusses our approach to automated test evaluation. We conclude the paper with a summary of the benefits of our test method, and discuss its applicability to other engineering domains.

## 2 The challenge of testing

When testing a mechatronic system, it is usually not sufficient to test the system under laboratory conditions for a couple of idealized use cases. Instead, to increase the chance to discover all hidden bugs and design flaws, the system should be tested in as many different relevant conditions as possible. Consider as an example an assembly such as an automatic transmission used in a passenger car.

In this case, the space of working conditions extends at least along the following dimensions:

- *weather:* for example, temperatures range from - 40°C to 40°C, with significant impact on oil properties of hydraulic subsystems
- *street:* different road profiles, uphill, down hill, curves, different friction laws for road-wheel contact
- *driver:* variations of attitude and behavior of the human driver, including unforeseen (strange) ways of driving the car
- *spontaneous component faults:* during operation, components of the assembly may spontaneously fail at any time; the control software of the assembly must detect and react appropriately to these situations, in order to guarantee passenger safety and to avoid more serious damage
- *production tolerances:* mechanical, electrical and other physical properties of the involved components vary within certain ranges depending on the manufacturing process
- *aging*: parameter values drift for certain components during the life time of the assembly
- *interaction with other assemblies:* a transmission communicates with other assemblies (engine, brake system) through a network that implements distributed functions; for example, during gear shifts, the transmission might ask the engine to reduce the torque in order to protect the switching components.

These dimensions span a huge space of possible operational conditions for an assembly. The possibilities along each dimension multiply to form a huge cross-product. The ultimate goal of testing is to verify that the system performs adequately at every single point of that space. It would be great to have techniques to mathematically prove certain properties of the system (such as the absence of unwanted behavior), which would enable a test engineer to cover infinitely many cases within a single work step. However, such proof techniques (e.g. model checking, cf. [1]) are by far too limited to deal with the complexity of the system level test considered here.

In practice, the goal of covering the entire state space is approximated by considering a finite number of test cases of that space.

# 3 A critical view on some test methods in use

Testing at different functional integration levels (e.g. component, module, system, vehicle) and in different setups (e.g. MiL, SiL, HiL, physical prototypes) is nowadays an important, integral part of the development process. The earlier problems are discovered and eliminated, the better. Very often, however:

(a) relevant tests can only be formulated, or have to be repeated, at higher levels of functional integration (e.g. at system level) - consider, for instance, the system reaction in case of component faults

(b) system-level tests are only performed in a HiL or physical prototype setup[1].

Let us briefly review some of the limitations of the HiL- / physical prototype based testing:

- *time, costs, safety:* physical prototypes and HiL setups are quite expensive and busy resources; testing takes place late in the development cycle; not too many tests can be conducted; the reaction to certain component faults cannot be tested with physical prototypes due to safety hazards
- *lack of agility:* it usually takes a long time between the change of a software function and the test of its effects
- *limited precision or visibility:* due to real-time requirements the physical system models used in HiL setups are often extremely simplified and therefore extremely imprecise; debugging and inspection of hidden system properties is difficult if not impossible for these setups.

Note, the above limitations are not present in MiL / SiL setups. While the importance of the HiL tests and of the tests based on physical prototypes should not be underestimated, our argument here is that they must be complemented by a more significant role of MiL and SiL tests at system level. See also [4], [5].

Irrespective of the setup used, the main limitation of common system-level test practices is the limited test coverage that can be achieved with reasonable effort. For example, test automation in a MiL / HiL setup is typically based on hand-coded test scripts for stimu-

1 There are several reasons why this is very often the case. An important one stems from the complexity of the mechatronic development processes: several disciplines, several teams, several tools, several suppliers – together with lacking standards and practices for exchanging and integrating executable functional models.

lating the partially simulated assembly with a sequence of test inputs, including code for validating the measured response. Coding and debugging such test scripts is a labor intensive task. Given typical time frames and man power available for testing, only few (say a few dozen) cases from the huge space of possible use cases can be effectively addressed by such a script-based approach. For testing using a test rig or by driving a car on the road, this figure is even worse. For example, it is practically impossible to systematically explore the assembly's response in the case of single component faults in a setup that involves dozens of physical (not simulated) components.

When testing for the presence (or absence) of a certain system property, script-based tests verify such a condition only during a few, specifically designed scenarios and not throughout all tests.

In practice this means that many scenarios are never explored during system test and that for those scenarios explored, usually only a few of the relevant system properties are tested. Consequently, bugs and design flaws may survive all tests. These are risks the method presented here can help to reduce, adding additional robustness to the design process.

# 4 Exploring system behavior with TestWeaver

TestWeaver is a tool supporting the systematic test of complex systems in an autonomous, exploratory manner. Although the method could, in principle, be applied to HiL setups as well, it is primarily geared towards supporting the MiL and SiL setups. The overall design objectives of TestWeaver were to:

(a) dramatically increase the test coverage, with respect to system behavior, while

(b) keeping the workload for the test engineer low.

To achieve this, we wanted to remove the necessity of exclusively relying on hand-coded test scripts, since we had identified script production as the main hindrance on the way towards broad test coverage.

## 4.1 The “chess” principle

The key idea was: Testing a system under test (SUT) is like playing chess against the SUT and trying to drive it into a state where it violates its specification. If the tester has found a sequence of moves that drives the SUT in such an unwanted state, he has won a game, and the sequence of moves represents a failed test.

There are more analogies: To decide for a next best move, chess computers just explore recursively all legal moves possible in the current state and test whether these lead to a goal state. This search process generates a huge tree of alternative (branching) games. In TestWeaver, the automated search for bugs and design flaws is organized quite similarly. Our method assumes that the SUT is available as an executable simulation (MiL) or as a co-simulation of several modules (SiL). As usually done, the SUT is augmented with a few components that communicate with the test driver. These communication components, called *instruments*, implicitly carry the “rules of the game” that TestWeaver is “playing” with the instrumented SUT. Namely, they carry information about: the control actions that are legal in a certain situation, the interesting qualitative states reached by the SUT, and, eventually, the violation of certain system requirements. Each instrument specifies a (relevant) dimension of the SUT state space. The value domain along each dimension has to be split into a finite set of partitions. Each SUT, or SUT-module, has to be configured individually by placing and parameterizing the instruments inside the SUT. The “game” is played in this multi-dimensional partitioned system space.

*Figure 1: The chess principle*

## 4.2 Instruments

An instrument is basically a small piece of code added to the un-instrumented version of the SUT using the native language of the executable, e.g. Modelica, Matlab/Simulink, or C. The instruments communicate with TestWeaver during test execution, which enables TestWeaver to drive the test, to keep track of reached states, and to decide during test execution whether an undesired state (failure) has been reached. TestWeaver supports basically two kinds of instruments, *action choosers* and *state reporters,* that can come in several flavors:

**1. state reporter**: this instrument monitors a discrete or a continuous variable (e.g. a double) of the SUT, and maps its value onto a small set of partitions or discrete values (e.g. low, medium, high). During test, this instrument reports each partition change of the monitored variable to TestWeaver. This is used by TestWeaver to keep track of reached states and to maximize the coverage of the partitioned / discrete state space.

**2. alarm reporter**: this is actually a state change reporter. In addition the partitions are associated with severity levels, such as `nominal`, `warning`, `alarm`, and `error`. The reachability of a "bad" state corresponds to a failure of the test currently executed. Note: these failure conditions are verified throughout all the tests run by TestWeaver.

**3. action chooser**: this instrument is associated with an input variable of the SUT. In an automotive application, an input variable may represent the acceleration pedal or the brake pedal of a car. Depending on the details of the instrumentation, this instrument asks TestWeaver either periodically or when a trigger condition becomes true to choose a discrete input value for its input variable from the partitioned value domain of the variable.

**4. fault chooser**: this is a special case of action chooser. The value domain is partitioned into nominal and fault partitions and can can be used to represent alternative fault modes of a component of the SUT. For example, a shift valve model may have behavior modes such as: `ok`, `stuckClosed`, and `stuckOpen`. Instruments like these are used by TestWeaver to inject (activate) a component fault occurring spontaneously during test execution.

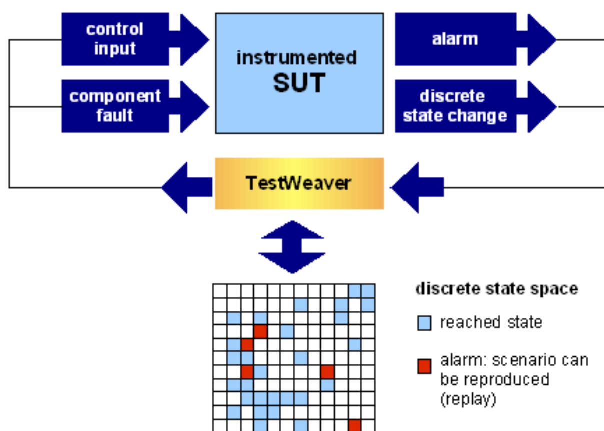


*Figure 2: Instruments connect SUT to TestWeaver*

Engineers like to work with their favorite modeling environment. Therefore we have implemented versions of the above instruments for alternative modeling environments, including Matlab/Simulink, Modelica, and C. The idea is to allow the test engineers to instrument a SUT in their favorite modeling language, i.e. using the native implementation language of the SUT, or of the SUT-module that they are working on.

In addition to the explicit instruments, TestWeaver monitors the process of executing the SUT and records problems, such as divisions by zero, memory access violations, or timeouts in the communication.

### 4.3 Experiments, scenarios and reports

In TestWeaver, an *experiment* is the process of exploring and documenting the states reached by the SUT during a certain period of time, possibly taking into consideration additional search constraints and coverage goals. An experiment usually runs completely autonomously for a long time, typically several hours, and without requiring any user interaction.

When running an experiment, TestWeaver generates many differing scenarios, by generating differing sequences of answers for the action choosers. A *scenario* is the trace (or protocol) of a simulation run of the given SUT in the partitioned state space. TestWeaver combines several strategies in order to maximize the coverage of the reached system states and to increase the probability of finding failures. The results are stored in a *scenario data base* of the experiment, i.e. a tree of scenarios (actually a directed graph), as shown in Fig. 3.

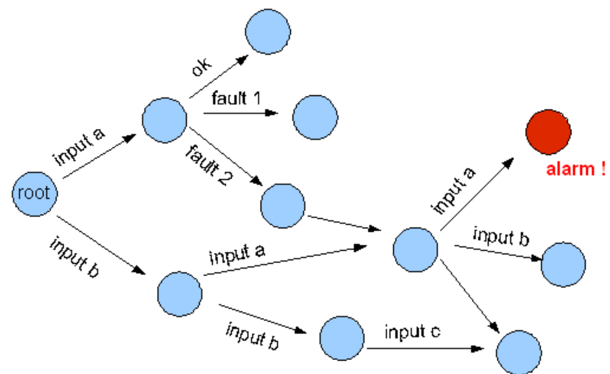


*Figure 3: Scenarios generated by an experiment*

The user can investigate the states reached in an experiment using a high level query language similar to SQL. Results are displayed in reports. A *report* is basically a table that displays selected properties of the scenarios stored in the scenario data base. The user specifies the structure and layout of a table by templates, while the content of a table depends on the content of the scenario data base – see Fig. 9. There are two kinds of reports: *overview reports*, that

document state reachability, and *scenario reports*, that document details of individual scenarios.

A user may specify, start, and stop an experiment, reset the experiment's data base and investigate the reports generated by the experiment, the last even while the experiment is running. Individual scenarios can be *replayed*: i.e. the SUT is restarted and is fed with the same sequence of inputs as the one recorded in order to allow detailed debugging of a problem, e.g. by plotting signals and other means.

### 4.4 The experiment focus

The dimensions and the partitions of the state space are configured by the instruments of the SUT. Apart of these there are also other means that can constrain the exploration, either as part of the instrumented SUT, or as explicitely defined in the specification of the *experiment focus* in TestWeaver. The focus of an experiment specifies which region of the state space should be investigated when running the experiment. During an experiment, TestWeaver tries to drive the SUT into those states that are in the experiment's focus. The experiment focus is currently specified using two means:

- *constraints:* the constraints limit the size of the considered state space of an experiment. They can limit, for instance, the duration of a scenario, or the allowed combinations of inputs and states. A high level constraint language is provided for this purpose. In an automotive application, a user could, for example, exlude all scenarios where brake pedal and acceleration pedal are engaged simultaneously. For a fault analysis, a constraint could be used to exclude certain fault modes from investigation, or to limit the number of faults inserted in a scenario: typical values are 0, 1 and 2. Higher numbers are reasonable when investigating fault-tolerant systems, e.g. systems with complex fault detection and reconfiguration mechanisms
- *coverage:* the user can tell TestWeaver to use some of the reports of the experiment as defining the coverage goals of the experiment. A report used in this way is called coverage report.

Experiments with different SUT versions and with different focus specifications can be created, run and compared with each other.

### 4.5 Analyzing and debugging problems

The alarm and error states of the SUT are reported in the overview reports. For each problem one or more scenarios that reach that state can be recalled from the scenario database. The scenarios can be once again *replayed* and additional investigation means can be connected. Depending on the SUT simulation environment these can be, for instance: plotting additional signals, connecting additional visualization means such as animation, setting breakpoints, and even connecting to step-by-step evaluation with source code debuggers – for instance for SUT modules developed in C.

### 4.6 The Modelica instrumentation library

In this section we briefly present the Modelica instrumentation library of TestWeaver, cf. Fig. 4.

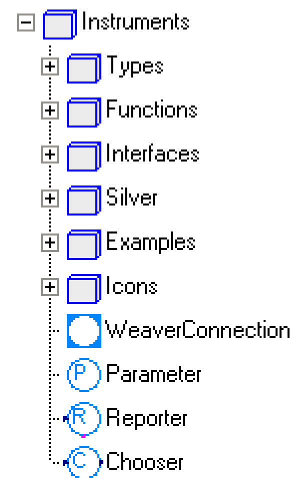


*Figure 4: The Modelica Instruments library*

The package *Instruments* contains ready to use components for instrumenting Modelica models. Each instrument is a Modelica component that communicates with TestWeaver through an outer *WeaverConnection.* Instruments extend the partial base class *Instruments.Interfaces.Instrument,* which is a model with the following parameters:

- *variable:* the name of the SUT variable defined by the instrument
- *comment:* a description of the instrument
- *unit:* of the classified real value, e.g. `"km/h"`, `"kg"`, or `"%"`
- *intervals:* an array of number pairs defining the numeric partitions of the real value, e.g. `[0,100; 100,150; 150,1e10]`
- *labels:* an array with the partition names, e.g. `{"ok", "hot", "damaged"}`
- *weaverConnection:* outer reference.

Furthermore, the *Instruments* package provides the following instruments as Modelica components:

- *Reporter:* an instrument that adds a severity rating to each interval and reports the value of a SUT variable when triggered; whether the reported value represents an alarm or a nominal state depends on the associated severity
- *Chooser:* an instrument that adds an occurence rating to each interval and allows TestWeaver to control a variable of the SUT, when triggered; whether the control value represents a fault to be injected or a nominal value depends on the corresponding occurence rating
- *Parameter:* a *Chooser* whose control variable is a model parameter to be initialized by TestWeaver at simulation start; this is used to model parameter deviations, for instance, due to aging, or due to production tolerances.

Fig. 5 shows how to use a *Reporter* in a model.

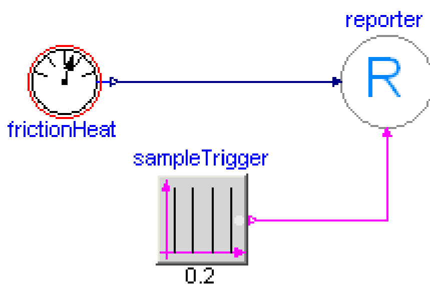


*Figure 5: Reporting a temperature*

The reporter is connected to the output of a heat model, and to a boolean trigger signal. Whenever the trigger signal becomes true, the reporter classifies the temperature w.r.t the partition margins and reports the result through the *WeaverConnection* to TestWeaver.

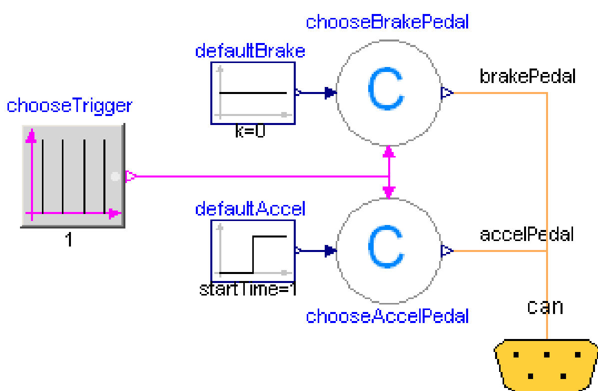


*Figure 6: Controlling two pedals in a car*

Likewise, Fig. 6 shows two *Choosers* called `chooseBrakePedal`, `chooseAccelPedal`. Both accept a default value at the left side, output the signal chosen by TestWeaver and accept a boolean trigger signal as input, which defines the time points when new values can be changed by TestWeaver.

## 5 Example: automatic transmission

As an application example for TestWeaver, consider the development of the control software for an automatic transmission. An instrumented Modelica model of an entire car, including the transmission and a *WeaverConnection* is shown in Fig. 7.

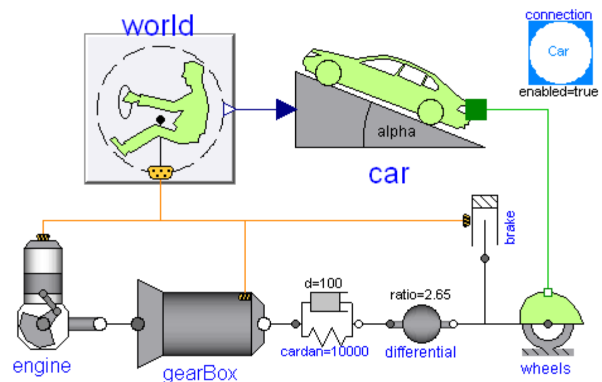


*Figure 7: An instrumented car model*

The control software is developed using a SiL development environment. The executable SUT is hence co-simulating two modules: the compiled control software, and the compiled Modelica model of the car. Since the Modelica model has been instrumented, the SUT also contains functions to communicate with TestWeaver. When TestWeaver starts the SUT to perform a system test, all contained instruments register themselves at TestWeaver with all their declared static properties (intervals, labels, severities etc.). TestWeaver then displays a list of these instruments, see tree in Fig. 9. Selecting an instrument in the tree displays all its properties. Fig. 8. shows how TestWeaver displays the heat reporter of Fig. 5.

**Instrument heatA**

Instrument role: ALARM
Instrument type: REAL
Instrument unit: J
Declared by: Car

| partitions | occurence | severity | color | values |
|---|---|---|---|---|
| damaged | 10 | 10 | #F03939 | 80000.0 : 1.0E10 |
| ok | 10 | 0 | #AFDFDF | -0.1 : 40000.0 |
| hot | 10 | 0 | #7FBFBF | 40000.0 : 80000.0 |

*Figure 8: Reporter (Fig. 5) displayed by TestWeaver*

The tree shown in Fig. 9 also contains an item for each report of the experiment. Selecting such a report displays the report as table. Fig. 9. shows a report that shows which gear shifts have already been reached during the experiment, and whether critical temperatures at the clutches A and B have been reported. For each state, up to two scenarios are referenced in the right most column. Clicking on such a reference displays that scenario as a sequence of discrete states. It is possible to reproduce the entire scenario with an identical simulation such that the test engineer can access all its details. For example, runtime exceptions of the control software (such as division by zero) can be reproduced this way and inspected using the usual software debugging tools.

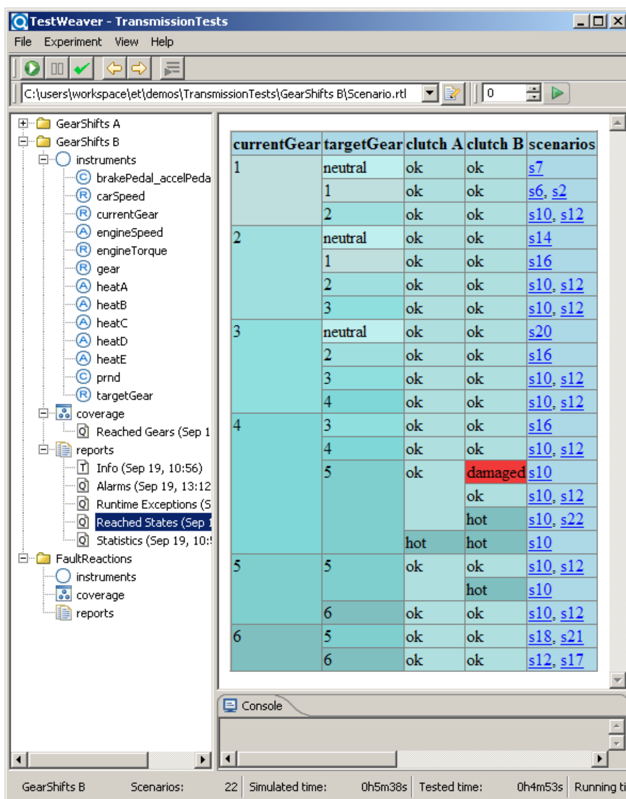

Figure 9: TestWeaver displaying a coverage report

## 6 The instrumentation process and quality watchers

If an executable SUT exists it is relatively easy to add the instrumentation for the action choosers and the pure state reporters – those reporters that do not attempt to classify the states as “good” or “bad”. A good understanding of the system functionality is required, in order to select the relevant system features and the relevant qualitative partitions. This activity in not completely new for the test engineers: the commonly used “Classification Tree Method” for specifying tests relies exactly on this kind of system analysis [2].

For the automatic test evaluation TestWeaver uses alarm reporters. The alarm reporters monitor correctness or quality conditions of the SUT. Some of these conditions are easy to define, others might require quite sophisticated watchers. Let us begin with the easier cases:

(a) often physical components have well known functional or safety ranges of operation that should not be exceeded; examples: maximal power dissipated by a component, maximal temperature, maximal pressure in a container, maximal allowed rotational speed of an engine or gear, etc.

(b) similar “local” correctness conditions exist for software functions; in the software industry such *assertions* are widely used today for detecting, diagnosing and classifying programming errors during test; examples here: maximum number of allowed objects in a buffer, access indices within array bounds, assumed domains for input parameters, assertions about function pre-conditions, post-conditions and other invariants that can be easily declared and monitored – see also [3].

The above quality conditions can be locally implemented, e.g. in a component type library. In such a case, each component instantiation will also instantiate the check of the quality condition and the instrumentation required.

In addition, the run time exceptions of the SUT process (e.g. divisions by zero, memory access violations, etc.) and the communication timeouts (possibly produced by infinite loops, non-converging numerical solvers, etc.) are anyway monitored and reported by TestWeaver.

An important class of quality conditions can be relatively easily defined when the SUT includes a controller and a model of the controlled system. Often controllers have in some form (at least a simplified) inverse model of the controlled system. This makes it easier to formulate system invariants or quality conditions. For instance: when not shifting, the controller gear should match the gear from the transmission model; if no fault is set in the hardware model, the on-board diagnosis should not detect any fault; if a fault code is generated, then the fault should coincide with a fault set in the hardware model. In general, the assumptions made by the control system about the state of the controlled system should match (within certain acceptable delays and tolerances) the state of the model.

The more we migrate from checking correctness to checking quality, the more complex and subtle the watchers can become. For TestWeaver arbitrarily complex quality watchers can be implemented with Modelica, Matlab/Simulink or C. In principle, any conventional test case can be turned into a quality watcher, although it might be sometimes difficult to generalize the specific conditions checked in a test case. The effort will be rewarded because:

(a) the quality condition will be checked not only for one input sequence, but for many differing scenarios, and

(b) a more general formulated condition is likely to survive unchanged, or with only small changes, when new SUT versions are produced later on, during development.

The tuning of some complex quality watchers can be quite laborious. In practice, there will always be cases when false alarms are generated. Therefore, after each experiment, also a detailed manual analysis and diagnosis of the problems found is prescribed by our test method.

# 7 Summary and conclusions

The increasing pressure to shorten and cheapen development for more and more complex products requires new test strategies. Today we see early module tests and late system-level tests, like HiL and test-rigs, as state of the art. The importance of early system-level testing increases with the increasing complexity of module interaction because bugs on system-level are more likely, more costly to fix and harder to find. Testing before physical prototypes exist, for both controllers and hardware, is one necessary step towards early system-level testing.

As long as the behavior of a system can be described easily using stimuli-response sets, script-based testing is a feasible strategy. With increasing system complexity, this method fails to provide the necessary coverage at reasonable cost. On the other side, our test method allows to:

(a) systematically investigate large state spaces with low specification costs: only the rules of the “game” have to be specified, not the individual scenarios

(b) discover new problems that do not show up when using only the predefined test scenarios prescribed by traditional test methods; TestWeaver can generate thousands of new, qualitatively differing tests, depending on the time allocated to an experiment

(c) increase the confidence that no hidden design flaws exist.

In chapter 5, we have sketched the application of TestWeaver to a SiL-based system test of an automatic transmission. We have several years of experience with this kind of applications. However, the application of TestWeaver to other domains seems promising as well, especially for those cases where a complex interaction between the software and the physical world exists. For instance:

- *driver assistance systems:* in car systems such as ABS, ESP, etc. we meet a complex interaction among the control software, the vehicle dynamics and the human driver; this leads to myriads of relevant scenarios that should be investigated during design
- *plant control systems:* in plants for chemical processes, power plants etc. we meet the interaction of the control software, plant physics and the actions of the operators; again, the same kind of complexity that calls for a systematic investigation during design.

TestWeaver runs on Windows platforms. It is a powerful, yet easy to use tool: users can use their native specification or modeling environment and don't have to learn yet another test-specification language.

## Acknowledgments

Special thanks to Volker May who initiated the research that finally led to TestWeaver.

## References

[1] Berard et. al.: Systems and Software Verification: Model-Checking Techniques and Tools, Springer Verlag, 2001.

[2] Grochtmann, M., Grimm, K.: Classification Trees for Partition Testing. Software Testing, Verification & Reliability, Volume 3, No 2, pp. 63-82, 1993.

[3] Meyer, Bertrand: Applying "Design by Contract", in Computer (IEEE), 25, 10, October, pages 40-51, 1992.

[4] Rebeschieß, S., Liebezeit, Th., Bazarsuren, U., Gühmann, C.: Automatisierter Closed-Loop-Testprozess für Steuergerätefunktionen. ATZ elektronik, 1/2007 (in German).

[5] Thomke, Stefan: Experimentation Matters: Unlocking the Potential of New Technologies, Harvard Business School Press, 2003.