Systematic Test and Validation of Complex Embedded Systems

Mugur Tatar, Jakob Mauss
QTronic GmbH, Alt-Moabit 92, D-10559 Berlin, 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 human driver / operators and the environment has to be taken into account. This resulting behaviour is extremely difficult to foresee and leads to a new kind of complexity which is difficult to master by the development engineers. Moreover, validating system-level behaviour for such systems is not supported well by most of the existing test and validation methods - consider, for instance: (a) formal proof methods, such as model-checking, 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. We present here a method, based on virtual system integration, simulation and intelligent test generation, that addresses specifically challenges raised when testing and validating system level behaviour. The method, implemented by TestWeaver, combines simulation and intelligent search to investigate system behaviour 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 applications of the method and results from the automotive power-train and driver assistance systems.**

I. Challenges of System Test and Validation

The number of software-controlled functions in automotive and aerospace systems is continuously increasing; the complexity of the software implementing them is increasing as well. In a typical automotive 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 behaviour 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 behaviour that results in the interaction of a controller with the physical world is not easy to assess. While software in isolation can, theoretically, be modelled and analyzed using discrete models, the controlled physical system and the environment cannot be in general approximated using discrete models.

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 systems, or subsystems connected to test-rigs, as well as Hardware-in-the-Loop (HiL) simulation platforms, are used to investigate system-level behaviour. Whether under real conditions, or using test-bench or HiL simulations, the challenge remains to validate the system-level behaviour 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 / aircraft configurations*: certain subsystems are embedded in many differing system configurations. Not only the logical specifications need to be verified, also the concrete configuration parameters have to be safeguarded, 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) *operator / driver controls*: human operators may interact in many different ways with the system, some cases may be rare or even obscure - but the system should operate in a safe way
5) *environment*: intelligent controllers react to certain environment changes: wind, temperature, street and others. 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 rather limited number of test cases.

II. Test and Validation Methods in Use

We shortly review here some of the methods used in the industry for the 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 system 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 real embedded controllers are 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*

In short, the definition of module and system as used here is: modules are functional building blocks of systems. An embedded system integrates, beside software modules, also a physical sub-system that is controlled by the software. Modules are usually well understood by their software developers, while systems are more difficult to understand: they integrate the work of many developer teams from several disciplines. The system simulation integrates, beside all software controller modules, also a simulation of the controlled physical sub-system - the "plant model". Because there is a very complex interaction between the software and the physical world very many system requirements can be tested only at system level in general.

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 their 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 to be 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 the module level alone. 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, during operation. Conclusion here: development processes that claim to ensure system correctness and quality need to offer a high automation and coverage of the tests also at the *system level*, the module level alone is insufficient.

Since the *validation* of the *module requirements* must be performed in the *system context* anyway, and since the test and validation must be done in the end at *system level* also, a natural question to ask is: how much of the tests should be developed / performed independently at module level, and how much using the *system context*. This question has a pragmatic and economic motivation: the cost of developing and maintaining tests for (many) complex modules is often overwhelming. This is especially striking for complex modules, i.e. modules having *hundreds* of input and output signals. Instead of specifying test vectors and sequences with (plausible) value combinations for hundreds of signals, it is easier to record such vectors in a c system. In other words, the system can act as a *test harness* for the module tests – a “cheap” harness, if the system model is be built for system test and validation anyway. We conclude here: one should have the possibility to decide based also on pragmatic reasons how to best *split and shift* the testing effort between the module and the system level.

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, cf. [14]. 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, cf. [16]. 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. implementation

Compare simulation traces from MiL or SiL to traces that use the target embedded 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 electronic 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 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. This 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 <postcondition>

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's post-condition is evaluated to true or false as the test success/failure condition. A particular weakness of this approach is that it ignores the general nature of the requirements: the requirements are valid in the whole state space (when taking the preconditions into account), but the above model cannot test requirements in arbitrary points of the state space. A better strategy implements *requirement checking* as *system invariant observers* - this allows a continuous tracking of all requirements at all times during any simulation scenarios.

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 predefined 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 very useful information, we argue that they still do not measure sufficiently well the requirements of a "good" *system test*. A good coverage measurement at system level should assess also 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 indicate meaningful *system state variables*, the coverage measurement shows which states (and transitions) have been reached during a test - see figure 3 for an example.

III. 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 correctness and safety requirements* as system invariants - this allows to detect problems in *any* simulation scenario. In a similar way observers of quality criteria can be defined.
- *Investigation of the system behavior in a large space of situations*. This is possible in a systematic and economic way only by using automated methods for intelligent generation and classification of simulation scenarios. This follows the goals to: (a) increase the coverage of

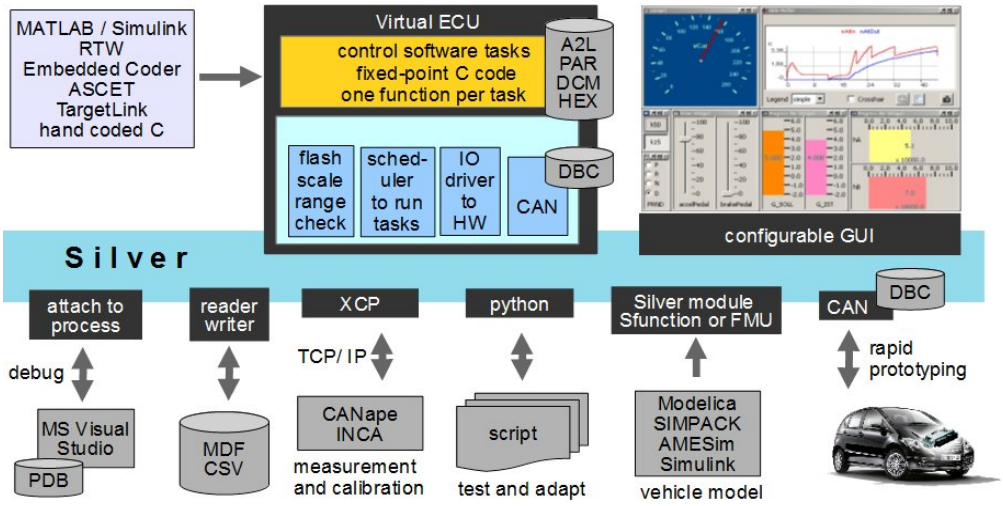


Fig. 1. Silver: virtual integration, simulation and rapid control prototyping on Windows PC. Silver co-simulates several compiled modules: virtual electronic controllers (ECUs), plant model components and other utility modules.

the system state space, and (b) detect requirement violations and worst-cases of quality criteria.

- Integration of automatically generated scenarios, human-specified test scenarios and scenarios defined by measurements of test drives. This pays tribute to potential weaknesses of automatic methods in particular cases.
- *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 root cause is identified.
- Customizable, easy to understand, visualization of state coverage measurements in addition to the commonly used requirement coverage and source-code coverage measurements.
- Black-box test: access to the source code of the controllers and plant models is not mandatory - this accounts for the fact that systems integrate modules from different suppliers and source code is often not shared across organizations.
- *Verification of the configuration and calibration parameters.* The simulated system must use exactly the same controller configuration and calibration parameters as the real systems. The configurable parameters of a system release must be tested as well, since *a single bad parameter value can have catastrophic effects*.

IV. 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 the SiL platforms, in particular using conventional PCs, will play a much more prominent role for system test and validation in future. Consider this simple argument: *copy-and-paste* can be done with a SiL simulation to conduct experiments in parallel on several PCs, but it cannot be applied to "multiply" HiL or other simulation platforms that rely on special hardware. PCs are *highly available* and provide *astonishing computational power* at *low costs*. Often enough, simulations can be conducted much *faster than real-time* or, at the other extreme, with quite *sophisticated physical simulation* plant models that *cannot* be used for real-time simulation. Of course, while functional aspects can be in general validated using purely virtual simulation environments, certain low-level communication and time-critical operations must be tested as well using HiL and physical prototypes.

Figure 1 depicts Silver, the virtual integration platform developed by QTronic based on SiL on Windows PC - for details see [3] and [4]. Silver can integrate several compiled modules representing virtual controllers and plant model components. Controller code can be imported from Simulink, Embedded Coder, TargetLink, Ascet or hand-coded C. Virtual ECU integration (vECU) is supported by the Silver Basic Software: this schedules time-triggered or event-triggered controller tasks and supports parameterization, measurement and calibration automotive standards, such as: DBC-configured CAN communication, ASAP2/A2L, PAR, HEX, DCM configured calibration via XCP. Single or multiple vECU setups can be built on a single PC. Complex setups use more than 400000 signals. 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 from several modeling tools: Simulink, GT-POWER, Dymola, MapleSim, AMESim, SimulationX, Axisuite, SIMPACK and others. Moreover, the new standard *FMI for Model Exchange and Co-Simulation* [5] is supported. Other features contribute to providing a 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 in the automotive industry at AMG, Mercedes-Benz, BMW, Volvo, IAV, Continental and others.

V. Automatic Exploration of System States with TestWeaver

Whether we are aware about it or not, intelligent search techniques help solving a multitude of problems around us nowadays: information retrieval, routing, optimization, planning, games - to name just a few. Also a rich literature describing various search-based techniques applied to automatic test generation can be found: local optimization, evolutionary, combinatorial, constraint-based techniques, combined with various structural coverage and goal oriented heuristics(cf. [1], [9]).

Our method combines simulation and intelligent search to investigate system behavior in a large space of situations automatically with the following goals: *(a)* increase coverage, i.e. drive the system in as many qualitatively differing states as possible, and *(b)* find requirement violations and worst cases for quality indicators. Functional and safety requirements, as well as other quality criteria, can be implemented as observers that monitor 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 operation can be integrated, but are optional. They are complemented by the large number of automatically generated scenarios. The method is supported by our tool TestWeaver and is in industrial use since several years.

Figure 2 depicts TestWeaver. TestWeaver connects to a SiL, MiL or HiL system simulation. It 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

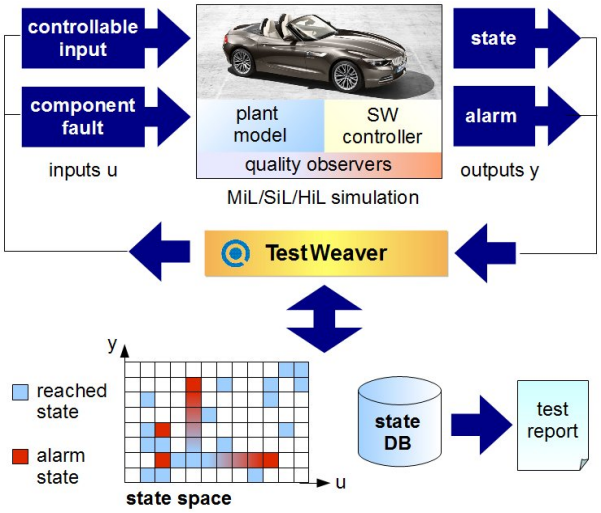
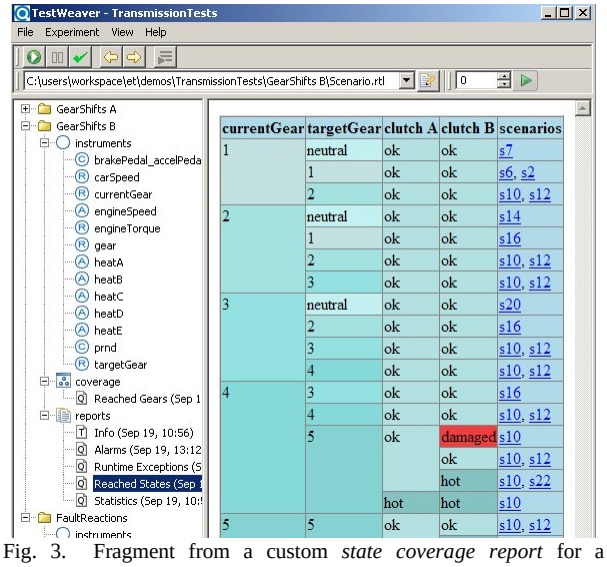


Fig. 2. TestWeaver: autonomous generation and evaluation of test scenarios with the goals to: (a) drive the system into states that have not reached before, and (b) find correctness violations and worst-cases of quality indicators.

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) - since 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 (huge) system



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* for a transmission system. It shows the gear shifts reached during the test, associated clutch temperature classes (ok/hot/damaged) and links to scenarios that can be used to drive the system those states. The engineers can select the variables to be added / removed to a coverage report in order to build meaningful system state measurements for their test objectives - for instance, one could decide here to add also street slope or engine torque regions to the system state classification. The measured variables can stem from the software controller, from the plant model or from the quality observers. The table rows are filled in by TestWeaver based on the values from the state database.

Requirement watchers - overview

name	FAILED tests	Failed example	Tests with SUCCESS	SUCCESS example
Watcher_EGS_001	51	s17.17	70	s2.84
Watcher_EGS_002	8	s14.38	9	s2.82
Watcher_EGS_003	0	(none)	2	s2.5

The gear shift from 4 to 5 must not take longer than 1.2 seconds. Furthermore, during this gear shift, heat of clutch B must not exceed CL_B_TEMP_MAX

Fig. 4. Fragment from requirement coverage report. For each requirement examples where the requirement was satisfied and / or was violated are reported.

state space on different smaller but meaningful sub-spaces for a better overview. See figure 3 for an example.

The *state 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 test engineer is interested in. The states that are referenced by a coverage report are marked as "interesting". Since TestWeaver remembers the path to any reached state, it can drive the system to an "interesting" state many times and can continue the simulation from those states with differing input settings. Each alternative simulation requires the restart of the system simulation from the initial state. From an interesting system state TestWeaver can try to simulate all possible singleton changes to the inputs - for instance using an exponent from each input partition. This is a simple but quite powerful strategy: by trying to change the inputs in qualitatively new system states, the probability to drive the system into further, not yet discovered, system states increases. As in the "classical" search pattern cf. [12], new interesting states discovered during the search are added to the *agenda* for future exploration. The states of the agenda are ordered according to a heuristic measure of their expected relevance and according to the clusters where they belong in the coverage reports. For instance, for the coverage report from figure 3, the classification according to the *currentGear* (the first column / first dimension) has more priority than the classification in sub-clusters according to the *targetGear* (the next dimension in the defined report). In effect, states that reach new *currentGear* values will be investigated before states that reach new value combinations for *(currentGear,targetGear)*, and so on.

Besides the observed states and transitions the state database (cf. figure 2) stores, also the *duration* of each observed state transition. By post-mortem processing of the state database further correctness, quality and statistical information can be extracted: such as longest and average duration of certain state transitions, 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 and SimulationX a Modelica library is provided, for Silver and HiL systems a Python library is provided, for other environments a C library is provided.

```
(state_classes, agenda)=({}, {initial_state})
while(not_finished)
  best_state = remove_best_state (agenda)
  for input in allowed_inputs(best_state)
    scenario=resetSUT_and_drive_to(best_state)
    scenario=append(input, scenario)
    scenario=continue_simulation(scenario,max_time)
    state_classes=classify(scenario,state_classes)
    add_new_states(state_classes, agenda)
```

Fig. 5. Generation of scenarios with system state and input domain coverage. In *classify* the classification of the system states according to the instrument partitions and the user defined coverage reports is done, as for instance in figure 3. Also software code coverage measurements can be used for the classification as well as states that reach minimal / maximal values for certain specified measurements. New interesting states after the classification are added to the *agenda*. In *continue_simulation* the scenario simulation is continued for a configurable maximal time - the inputs can be kept constant here or can be randomly changing after a specified latency.

While MiL and HiL simulation platforms can be used as well, the most proficient system simulation platforms used by our customers are purely virtual Software-in-the-Loop (SiL) integration platforms - for instance using Silver, cf. [2], [3].

Obviously the method used by TestWeaver is not a formal method - it can be classified as a semi-formal method. Therefore, although the search-based test helps unveil existing hidden problems in a system, there is *no completeness guarantee*: we can discover faults, but cannot prove their absence. There is no general objective criteria to decide when a system test is "complete" and when the scenario generation can be stopped. All (informal or semi-formal) testing methods are *incomplete* in this respect. In practice a test engineer will *decide to terminate* the state exploration based on a combination of:

- *objective coverage measurement*s, such as: system state coverage, software code coverage, requirement coverage, possibly other measurements, and
- *pragmatic considerations*, such as: restrictions on the time and other resources available for a test cycle.

A reasonable test coverage is achieved in many practical cases with simulations that run over night or during a week-end - practical applications are reviewed in section VII.

VI. Example - Validation of a Transmission System

In order to illustrate the method assume we are required to test the automatic transmission system of a passenger car. In a first approximation this system is composed of:

- *the electronic transmission controller (TCU)* - microcontroller running the transmission control software and having digital/analog I/O connections to sensors and actuators in the transmission hardware and a CAN-bus interface to other electronic devices
- *the automatic transmission hardware* - composed of a hydraulic and a mechanical subsystem. The hydraulic receives the control currents for electro-magnetic valves from the TCU and it distributes and applies pressure to mechanical actuators, such as clutches and brakes. Based on the opened and closed mechanical connections

different discrete transmission ratios can be achieved.

- *the rest of the car system* - this contains, most notably, the combustion engine, connected to the transmission input, and the connection from the transmission output to the wheels, including the brake.
- *the driver and the environment* - this includes, most notably, the driver controls (buttons, pedals, gear-lever) and the street (characterized by street-wheel contact properties and the street slope).

The role of the automatic transmission system is to select the most appropriate transmission ratio between the engine and the wheels, based on the driver controls, the car speed and the other context information available, while maintaining efficiency, comfort and safety of operation. A realistic transmission controller may contain about 150 modules and thousands of internal variables, say 6000-10000 or more. Beside the software itself, the behavior of the controller is also massively influenced by the *configuration and calibration parameters*. A TCU may have more than 10000 such parameters. If some have bad values the effects can be very bad, ranging from loss of comfort / efficiency, to loss of function in certain situations, therefore the parameters *must* be incorporated in the system test. Fortunately, a vTCU setup with Silver is able to deal with the level of complexity and accuracy required here for the system simulation including: *(i)* the TCU *software tasks*, *(ii)* the flashed TCU *parameters*, *(iii)* the *plant model* for the rest of the vehicle and the environment. A *realistic* plant model fit for the purpose of validating a TCU was reported in [15] to contain about 2800 variables, including 34 continuous state variables used by the algebraic-differential equation solvers. We can expect the complete closed-loop system simulation for a real transmission system to roughly run in real-time on a laptop, cf. [2],[15].

A. Objectives: test coverage

We want to test the system requirements in many differing driving situations. Here is an example of how we might define the system test coverage goals. *Operational state coverage*:

- all *gears* should be reached; furthermore, all possible *up/down shifts* should be exercised; furthermore, all gears and shifts should be exercised with differing *street slopes*: uphill, downhill, flat
- the *gear lever* can be moved from D to N and back while forward driving at various *car speeds* and differing *street slopes*
- the *shifts* should be exercised with differing *engine torques*: engine brake, low, medium and high torques.

Source code coverage: system tests are usually required to demonstrate at least 100% *function coverage*. But we may choose to measure and analyze also more detailed levels of coverage, such as *statement* or *decision coverage*. In order to measure the source code coverage it is enough to compile the controller code executed within Silver with a code-coverage measurement tool such as CTC++.

Requirement coverage: we intend to test all requirements, that will be formulated later, in all situations. Every requirement must be tested in at least one scenario, i.e. in case a requirement is defined using preconditions that restrict the activation of its pass condition, those preconditions must be satisfied in at least one scenario (of course, with TestWeaver the requirements will be tested usually in many scenarios, not only in one, but we want to easily detect if a requirement was not tested using the automatic test generation).

All of the above coverage measurements can be reported in TestWeaver by customizing report templates.

B. Instrumenting the system inputs

Although the system contains hundreds of modules and thousands of variables, after all modules are integrated there remain only a handful of free variables that can be controlled. These are basically the driver's controls (pedals, buttons, etc.) and some relevant environment parameters. Fault events, for instance for sensors and actuators are another category of controllable inputs, but we neglect these here. For every instrumented variable TestWeaver expects a description of the value domain as a finite set of relevant partitions. The most important system inputs for our transmission system are: acceleration pedal, brake pedal, gear lever and street slope. Their description using the Python syntax accepted by Silver is given below:

```
accPed = Chooser(tw, Variable('accPed'),
  unit ='%',
  # partition definition (min,max,name)
  [[ 0,   0, '0%'      ],
  [  0, 100, '0..100%' ],
  [100, 100, '100%'   ]])
brkPed = Chooser(tw, Variable('brkPed'),
  unit ='%',
  [[ 0,   0, '0%'      ],
  [  0, 100, '0..100%' ],
  [100, 100, '100%'   ]])
prnd   = Chooser(tw, Variable('prnd'),
  [[0,'P'], [1,'R'], [2,'N'], [3,'D']])
slope  = Chooser(tw, Variable('slope')
  unit ='%',
  [[-20,   0, 'downhill' ],
  [   0,   0, 'flat'     ],
  [   0,  20, 'uphill'  ]])
```

For every variable a set of relevant domain partitions with optional associated names are given. For the pedals we have chosen for simplicity only three domains, 0%, 100% and the values in-between. The default strategy used by TestWeaver will pick the middle value from an interval as partition exponent, therefore, when generating scenarios based on the above instrument description TestWeaver will generate for the pedals sequences of values from the set {0%, 50%, 100%}. It is easy to use finer granularity partition definitions. The recommended strategy is to start with a conservative number of partitions for the inputs and to refine these if necessary after examining the experiment results. In

real projects about 4-6 partitions for pedal values proved to be enough in most cases. For the *slope* we chose three qualitative regions, corresponding to flat, uphill and downhill streets.

The partition definition for the gear lever *prnd* is quite clear, for discrete inputs all values are used as independent partitions because they activate differing operating modes.

Based on the above description TestWeaver is allowed to freely change the values of the four variables at discrete time points (e.g. every 100 milliseconds). Instead of generating only one very long scenario, the strategy used by TestWeaver is to generate many short scenarios: short scenarios are easier to reproduce and analyze. The scenario length is chosen by the test engineer when configuring an experiment. For our case we can choose a value between 60-90 seconds, which is enough to accelerate a car to its highest speed and to brake it to standstill.

In certain cases the inputs are not completely independent and some restrictions among their values must be considered. Also in our example we will formulate such restrictions:

- the acceleration and the brake pedals should not be pressed at the same time
- the gear lever is in P only at car start, afterwards it should take values from R,N,D. Moreover, it is not allowed to reverse the direction of movement: above 5km/h forward car speed the gear lever should not be moved to R; analogously, below -5km/h no move to D.

In Python these restrictions can be formulated as functions that constrain the set of allowed partitions of a chooser at a given time depending on the values of other variables:

```
def brkPed_Partitions():
  if accPed.Value > 0:
    brkPed.partitions('0%')
  else:
    brkPed.partitions(Partitions.All)

def prnd_Partitions():
  if time.Value < 0.5:
    prnd.partitions('P')
  elif carSpeed.Value > 5:
    prnd.partitions('D','N')
  elif carSpeed.Value < -5:
    prnd.partitions('R','N')
  else:
    prnd.partitions('D','R','N')

def slope_Partitions():
  if time.Value == 0:
    slope.partitions(Partitions.All)
  else:
    slope.partitions(Partitions.Last)
```

For the *slope* we have chosen to change its values only at the beginning of a scenario and to keep it constant afterwards - for the simplicity of the definitions and analysis and because it satisfies our operational coverage goals. Note however, that if the user chooses to add measurements or self defined scenarios to TestWeaver's test database, he is not required to satisfy any of the constraints of the instrument definitions. Since no complete analysis method is known to exist for complex systems, in practice we use a combination of methods. In particular, if the automatic generation fails to reach some coverage goals in a given time, the test engineer can manually define additional scenarios using Python scripts or measurements to close the remaining gaps.

TestWeaver is in general allowed to abruptly change the values of a chooser during a scenario, e.g. from 0% to 100%. Some inputs might require, however, smooth changes - for instance the street slope is not "jumping". Two basic strategies are used in such cases: *(a)* insert rate of change limiters between the chooser and the system input or *(b)* make use of continuous signal generators (e.g. sinus, spline, etc.) controlled by discrete parameters. See [8] for an example where TestWeaver controls the 3-d street profile as well as wind profiles with smoothness constraints.

C. *Instrumenting system internal variables*

Let us assume our system has about 10000 variables, say, 7000 variables in the software controller and 3000 variables in the plant model. Of course they are not independent state variables, most of them are tightly coupled. In practice it is enough to instrument between 10 and, say, 50 variables if they are good indicators of the operational system states. In our example, if we have a look at the operational state coverage goals formulated before, we find out that the *gears*, the *shifts*, the *car speed* and the *engine torque* define important operational properties. These will be part of the monitored state variables by TestWeaver. The "gears and the shifts" can be measured via two variables of the software controller that define the *current gear* and the *target gear* of the controller; the car speed and the engine torque are variables computed in the plant model. There is basically no hard restriction on the number of instrumented state variables, they can be arbitrary continuous or discrete variables. For each instrumented variable domain we associate a finite number of partitions used for the state coverage measurement.

```
crtGear  = Reporter(tw,Variable('crtGear'),
  # partition definition
  [[ -1,'R'], [0,'N' ],
   [1],[2],[3],[4],[5],[6],[7]])
tgtGear  = Reporter(tw,Variable('tgtGear'),
  [[ -1,'R'], [0,'N' ],
   [1],[2],[3],[4],[5],[6],[7]])
carSpeed = Reporter(tw,Variable('carSpeed'),
  unit='km/h',
  [[-100,-5],[-5,5],[5,10],[10,20],[20,30],
   ...
   [100,130],[130,160],[160,200],[200,250]])
engTorque=Reporter(tw,Variable('engTorque'),
  unit='Nm',
  [[-1000,-5, 'brake'],
   [-5,    5, 'aroundZero'],
   [ 5,   20, 'low'],
   [20,  100, 'medium'],
   [100,1000, 'high']])
```

Note that, while the inputs can be (almost) freely chosen by TestWeaver, the internal states are not *controllable* at all - TestWeaver cannot simply *set* these values in order to get to

a certain state. What TestWeaver can do, and does, is to monitor the values taken by the states during differing simulation scenarios, to classify these values and to register all partition changes as interesting events, respectively interesting operational state regions. Once a region has been reached, TestWeaver learns the scenario leading there. It can later drive the system to that state many times and continue exploration with changed inputs, for instance as indicated in figure 5.

Once the inputs and some relevant states are instrumented, TestWeaver can immediately start the scenario generation. Two more steps are usually performed before the experimentation starts: *(i)* definition of report templates that depict the system state coverage (and help also TestWeaver to achieve it) and *(ii)* definition of correctness / quality requirements that will be monitored and reported during an experiment.

D. Reporting state coverage

TestWeaver has an SQL-like language with which very complex properties can be queried, e.g. minimal-maximal duration of states. However, simple reports can be built directly from the user interface by clicks on the table headers and by adding or removing instrumented variables to the table. This will display the cross product of partitions that were reached during the simulation for the selected variables, associated with one or more scenarios that can reach those states or state regions. For the table from figure 3, the engineer just selected the 4 variables from the table's header; the query is built, run and displayed as HTML table by TestWeaver automatically. In order to display the reached system states according to the aforementioned coverage objectives the following tables should be configured before the experiment starts:

1: *(slope, crtGear, tgtGear)*, 2: *(slope, carSpeed, prnd)*,
3: *(engTorque, crtGear, tgtGear)*.

Note, the engineer does not have to "fill in" the table rows of the tables, e.g. like the one from figure 3. The engineer only has to name the variables. The table will be filled in by TestWeaver based on the values from the state database.

E. Defining and monitoring system requirements

The complexity of correctness / quality requirements can range from very simple to arbitrarily complex. Let's start with some simple cases. Certain common requirements are automatically monitored by TestWeaver:

- software errors that lead to *crashes* of the simulation process, such as *division-by-zero*, *access violation*, etc.
- *min-max* and *range monitoring* for all vTCU variables documented in ASAP/A2L configuration files. In our case, we assume all the 7000 software variables have range descriptions in the A2L file. Their values will be monitored and violations will be reported automatically together with scenarios that can reproduce them.

Beside automatically monitored properties there almost always exist several "easy catches" that can be monitored. We can expect that our TCU integrates sophisticated condition monitoring functions. For instance, in [15] over 200 signals are used to detect *symptoms* of abnormal conditions. When simulating the nominal system behaviour (i.e. without fault injection) no symptom or diagnostic trouble code should be raised. This is an easy condition to monitor. Any reporter can associate a *severity* value with a partition that is not-nominal. Values from non-nominal partitions, called "*alarms*", can be reported automatically.

Let's now have a look at more complex requirements, ones that involve temporal dependencies among variables. The TestWeaver Python instrumentation defines a template *Watcher* class that can be instantiated and configured to model a wide range of common relations used by requirements. A few examples are illustrated. Below, `lambda` is a Python keyword, used to define small anonymous predicates/functions:

Req-001: *Once started, the engine speed should never get below 400rpm (engine stalled). Moreover the speed should never get above 7500rpm (over-speed).*

```
Watcher(tw,'Watcher-001', description='...',
  wait_event      = (lambda:
    engSpeed.Value>400),
  fail_condition  = (lambda:
    engSpeed.Value<400 or
    engSpeed.Value>7500))
```

Req-002: *If the pedal positions and the gear lever are not changing the target gear should be constant for at least 1s.*

```
Watcher(tw,'Watcher-002', description='...',
  while_condition = (lambda:
    constant(prnd, accPdl, brkPdl)),
  wait_event      = (lambda:
    tgtGear.Value!= tgtGear.prevValue),
  pass_condition  = (lambda:
    tgtGear.prevDuration > 1))
```

Req-003: *Whenever a gear shift starts, a request to reduce the engine torque must be issued within 20ms.*

```
Watcher(tw,'Watcher-003', description='...',
  wait_event      = (lambda:
    crtGear.Value!= tgtGear.Value),
  pass_condition  = (lambda:
    engTrqReq == -1), # torque reduction
  tolerance_time  = 0.02)
```

Many requirements can easily be configured using the Watcher template. If necessary, the full expressivity of the Python language can be used. The coverage of the requirements will be reported by TestWeaver with successful and failing scenarios, as illustrated in figure 4.

VII. Real World Applications

TestWeaver is in use for repeated system test and validation during development at several automotive OEMs and suppliers. We review here application results for several complex power-train and driver assistance systems.

In [7] a highly automated process used by Mercedes-Benz for the systematic validation of over 200 transmission

variants built in the Sprinter Vans is described. The control software of the 7G-TRONIC *automatic transmission* runs on PC under Silver in closed-loop with a vehicle simulation. A combination of scenarios stemming from real-life measurements and scenarios dynamically generated by TestWeaver are used by the validation process. The simulation is conducted on several PCs in parallel. For all 200 vehicle variants the simulation can be configured and conducted using an automatic process. A similar process is used by AMG and Mercedes-Benz for testing AT and DCT transmission systems built in passenger cars, cf. [2], [13], [15]. ZF reports in [11] about the use of TestWeaver for testing *hybrid transmissions*.

In [6] TestWeaver and Silver have been used for the 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 [10] 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 [8] 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 [8], 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 seconds 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 concluded, the TestWeaver approach seems extremely well suited for the validation of complex controllers during all stages of development.

VIII. Conclusion

We have discussed the challenges raised by the system test and validation of complex embedded systems, a task that requires to consider the interactions among very complex controllers, physical subsystems, operators 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.

References

[1] A. Blass, Y. Gurevich, L. Nachmanson, M. Veanes, “Play To Test", *Microsoft Research Technical Report* MSR-TR-2005-04, 2005.

[2] H. Brückmann, et al., “Model-based Development of a Dual-Clutch Transmission using Rapid Prototyping and SiL", *Proc. Intern. VDI Congress Transmissions in Vehicles*, Germany, 2009. Available: http://www.qtronic.de/doc/DCT_2009.pdf

[3] A. Junghanns, "Virtual Integration of Automotive Hard- and Software with Silver". Available: http://www.qtronic.de/doc/SilverIntro.pdf

[4] A. Junghanns, R. Serway, T. Liebezeit, M. Bonin, "Building Virtual ECUs Quickly and Economically", *ATZelektronik* 03/2012.Available: http://www.qtronic.de/doc/ATZe_2012_en.pdf

[5] Functional Mock-up Interface for Model Exchange and Co-Simulation, *specification*. Available: http://www.functional-mockup-interface.org/

[6] M. Gäfvert, et al., "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

[7] S. Gloss, et al., "Systematic Validation of over 200 Transmission Variants", *ATZelektronik* 04/2013. Available: http://www.qtronic.de/doc/ATZe_04_2013_en.pdf

[8] 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

[9] P.McMinn, "Search-based Software Test Data Generation: A Survey", *Software Testing, Verification and Reliability*, vol. 14, no. 2, 2004.

[10] Th. Neubert, M. Tatar, "Extensive Test of Heavy-Machinery ECU on a NI VeriStand HiL using TestWeaver. *14th ITI Symposium*, Dresden Germany, December 2011. Available: http://www.qtronic.de/doc/TestWeaver_NIVeristand_ITI2011.pdf

[11] 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, 2011. Available: http://www.qtronic.de/doc/TestWeaver_AutoReg2011.pdf

[12] P. Norvig, "Paradigms of Artificial Intelligence Programming", Morgan Kaufmann, ISBN 1-55860-101-0, 1992.

[13] 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

[14] Schilling, Alam, "Integrate static analysis into a software development process". Available: http://www.embedded.com/design/prototyping-and-development/4006735/Integrate-static-analysis-into-a-software-development-process

[15] M. Tatar, R. Schaich, T. Breitinger, "Automated Test of the AMG Speedshift DCT Control Software", in *Proc. 9th Intern. CTI Symp. Innovative Automotive Transmissions*, Berlin, Germany, 2010. http://www.qtronic.de/doc/TestWeaver_CTI_2010_paper.pdf

[16] W. Visser, et al., "Model Checking Programs", in *Proc. 15th IEEE Intern. Conf. on Automated Software Engineering*, Grenoble, 2000.