Module Test in System Context

Tjark Kiefer, Ingo Matheis

Abstract

Module tests are well proven methods to assure software quality. But with raising complexity of the code this method is not without its problems. Assume having a module with 20 inputs, each having 5 values to be stimulated. So you would have to define 5 to the power of 20 (=95,367,431,640,625) tests. Considering that in real life modules use counters and thresholds, the numbers get even worse. Thus the costs for maintaining and defining module tests especially for complex modules limit its use.

Regarding the V-Model by Barry Boehm the module test is followed by the integration test. During the development of the new 9G-Tronic by Daimler a SiL was built containing all functional code and including a fully automated build process. Because it was so easy to use the design engineers tested any changes using the SiL. Thus the integration test was done before the module test.

From that context the idea was born to derive the module test from the integration test. The mock objects needed to run the module test could be generated automatically from the integration test of the SiL. This method was used to test modules with high complexity (40-100 inputs, including analog sensor values) and it turned out that the effort to define and maintain such tests was very low. The module test in system context was well accepted.

1. Motivation

The development of the 9G-Tronic came with lots of challenges. The time given was 3.5 years which corresponds to 7 software development cycles, the developers were short of prototypes and since the 9G-Tronic replaces the 7G-Tronic there was little time between the production launch and producing high quantities.
Thus the key for making the control software reliable was to make its development highly efficient.

1.1 The Development Process

At Daimler there are two development cycles (aka VA-cycles) a year. While the V-part is following the V-Model by Barry Boehm, the A-part is about tuning the parameters such that the car performs best. In the following we restrict to the V-cycle.

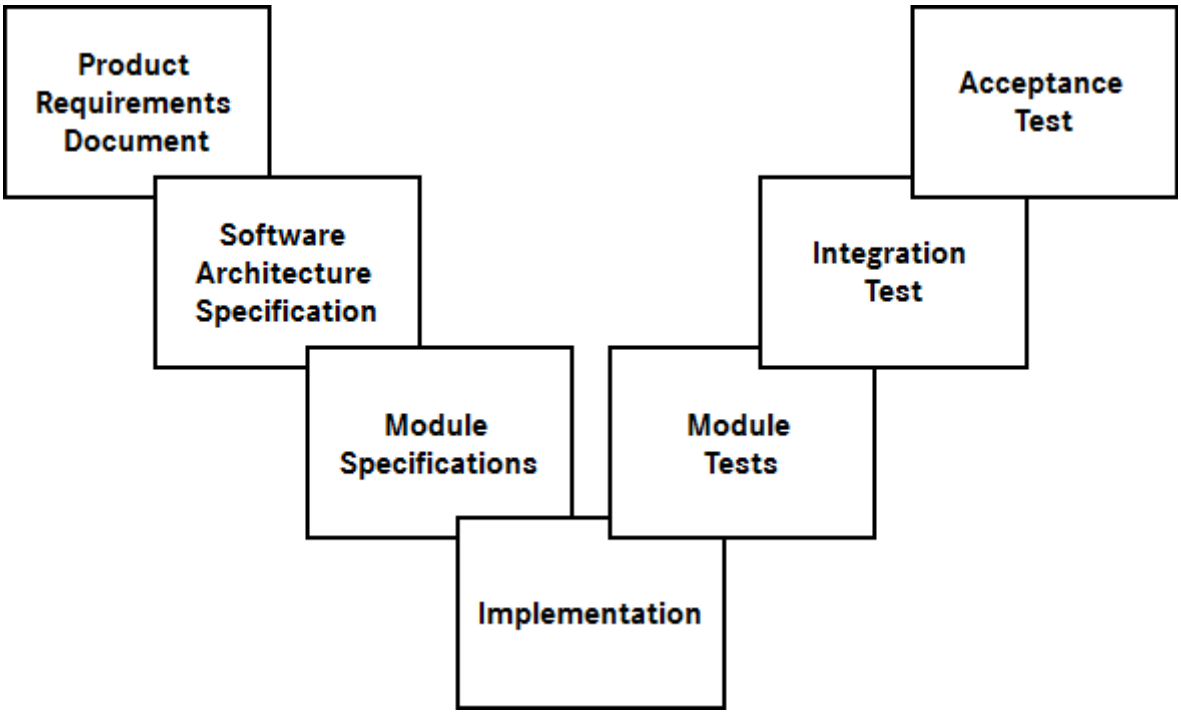

Diagram 1: The V-Model

From the function developer's point of view this process is:

- Specify your function,
- wait until it is implemented by the software engineers,
- flash the test software to the HiL or car, test it and
- report the test result.

This process turned out to be a bottleneck.

- Each loop takes its time because at least two engineers are involved until you can test.
- The specification of the design engineer is often misinterpreted by the software engineer. Thus adding more loops.
- Every idea the design engineer develops causes an interrupt in his workflow because every time he has to wait until the software engineer provides new software. This causes him to change context all day and therefore making him work ineffectively.

1.2 Pre-testing with the SiL

The SiL is a co-simulation environment that contains the complete function code, a plant model simulating the hardware, providing interfaces for actuators and sensors, a rest bus simulation and a virtual cockpit so you can do virtual test drives.

The SiL comes with a full featured build environment. All function software is included and any model change by the design engineer can be imported to the SiL by just a click of a button. The build process itself is completely automated and replaces the integration engineer.

Furthermore the co-simulation tries to emulate the target ECU: It provides connections for XCP and CAN so that you can connect with your standard measurement and calibration tool like CANape or INCA. Thus the engineers can reuse their tool

chain from the car and use it with the SiL. And because in the virtual world there is no bandwidth limitation of the CAN, you can measure tens of thousands of measurements at once.

If a design engineer wants to test his idea, he changes the model, pushes a button and within minutes he gets the SiL with his modifications, ready for an integration test. After testing intensively he can decide for the best option and formulate the one specification for the software engineer.

Because the SiL pre-testing is easily done software defects can be found as early as possible. Note that both the design engineer and the software engineer benefit from the SiL.

2. The Module Test

While the integration test evaluates the system behavior, especially how the modules of the system interact with each other, the module test focusses on a modules internal logic. For the sake of software quality both levels of detail must be covered.

In theory modules should be small and have less than 10 inputs and outputs. The more complex modules are well structured and highly modular from the inside. And the modules are easy to understand, well documented and some parts can even be reused.

In practice modules are rarely developed from scratch. For some years you add functions, fix bugs and refine if statements of state machines. To improve the transmission in terms of

- fuel economy,
- fast and comfortable shifting,
- driver and environment dependent shifting strategy and
- safety and durability

existing modules are reused and extended. Add some time pressure to this process and your modules most likely will become more complex than they should be.

We made a little statistics on the code for the 9G-Tronic:

- More than 100 modules with a total of 5000 inputs are used.
- About 20 modules have at least 50 inputs.
- About 5% of the inputs are continuous signals like revolutions per minute, torque or speed.
- Most of the 20 modules with 50 inputs have high complexity, using counters and thresholds.

The problem to do module testing for complex modules can be described as follows:

- The more complexity, the higher the costs defining a module test.
- The more complexity, the more module tests are needed.
- Time and money are limited resources.

Therefore: Module tests scale badly with the complexity of a module: Costs are high for creating and maintaining the tests. And using more human resources does not solve the problem.

3. Module Test in System Context

The challenge is to set up module testing for complex modules such that

- the costs implementing a test is independent of the complexity of the module,
- the test is backward compatible to classic module testing and
- the test is robust in terms of interface changes of the module.

This is the co-simulation that is used for the integration test.

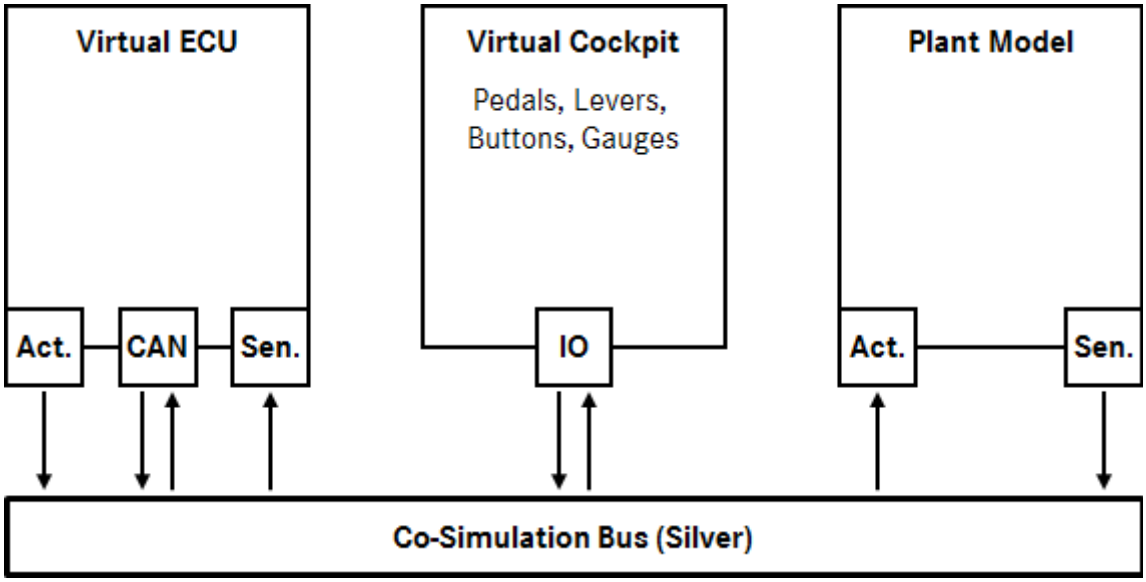

Diagram 2: The co-simulation

The ECU comes from the build process that automatically integrates all modules and emulates the XCP- and CAN-communication. The plant model is created by Dymola simulating the hardware of the transmission. The virtual cockpit offers buttons and sliders to interact with the virtual car. All is integrated using the co-simulation bus.

The diagram above is equivalent to the following one.

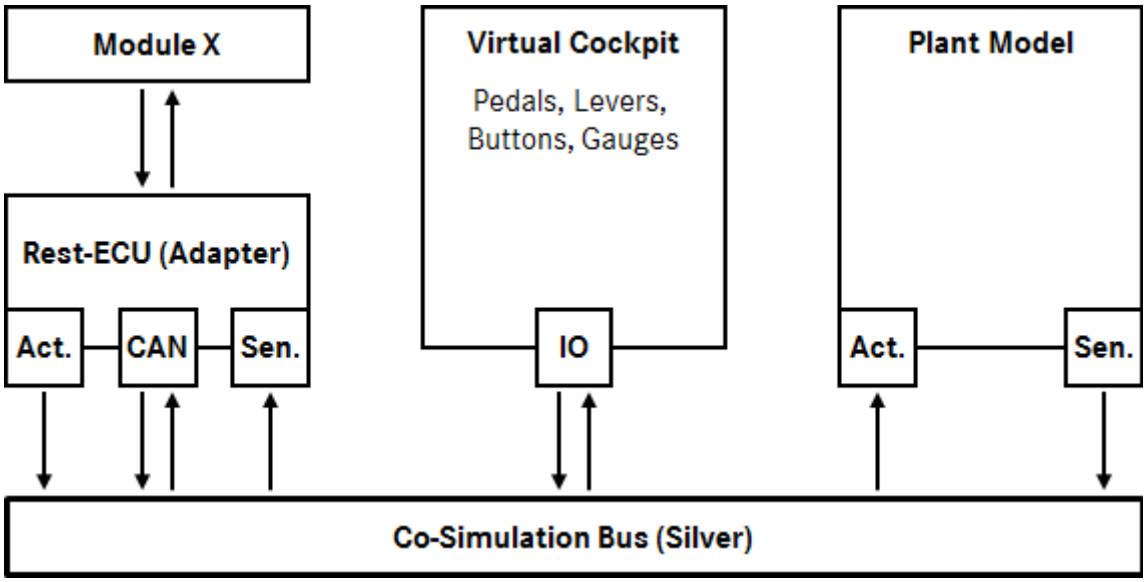

Diagram 3: The co-simulation from a different point of view

What has changed is the representation of the control software of the ECU. If we isolate one module then the rest of the control software becomes an adapter between the module X and the co-simulation bus. Since we already have solved the problem how to integrate all modules, the adapter comes for free.

After automatically determining the inputs of module X we add an appropriate bypass control panel to the cockpit of the SiL. Thus the SiL becomes the module test in system context.

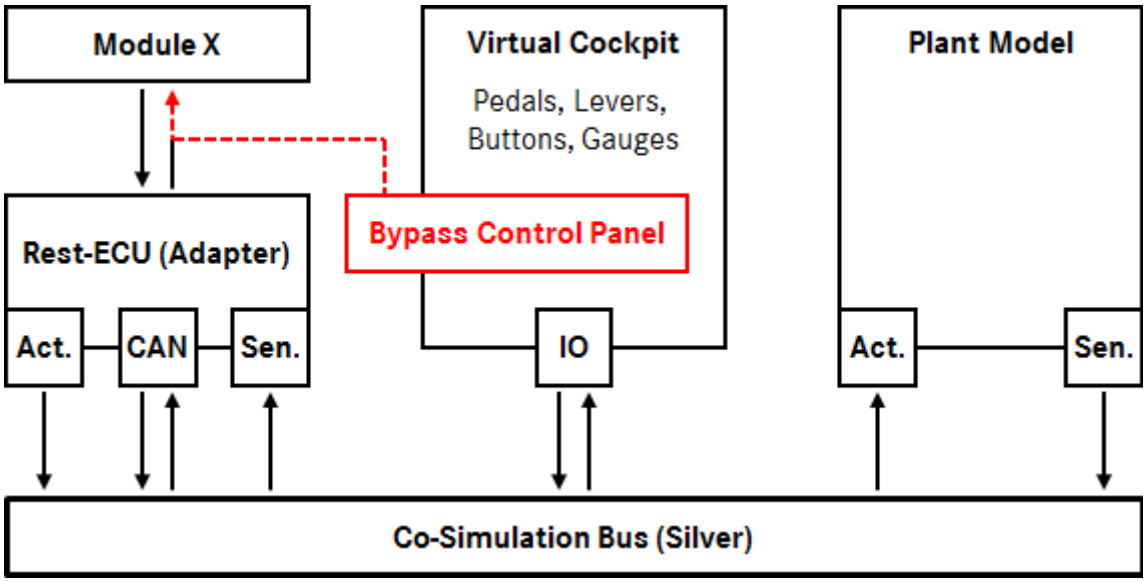

Diagram 4: The Module Test in System Context

If you record the outputs of the virtual cockpit and the bypass control panel you can replay any stimulus. Thus you can record the stimulus by simply driving the virtual car and using the bypass control panel at the right point of time.

The requirements mentioned above hold true:

- The costs to implement the module test are independent of the complexity of the module because most of the time you use the virtual cockpit as input and the bypass control panel that is automatically generated.
- The module test in system context is compatible with the classic module test. From the very beginning of the simulation use the bypasses only to stimulate the module. In this case, all other modules still are executed, but they are out of the test loop.
- If the interface of module X changes most of the stimulus recorded will still work since the adapter automatically changes too. One can construct counter example for this, but in practice they turned out to be rare.

The remaining part is to evaluate the module behavior. To do so we use so-called watchers. They have the following properties:

- Watchers are assigned to one or more stimulus.
- Watchers have a Boolean expression to determine when the evaluation starts. In the case of classic module testing this would be checking the time.

- Watchers have a Boolean expression to determine if the test succeeded. For this check you can define a tolerance time in which the test has to succeed or you can define that the success state has to stay true for some time.
- You can concatenate watchers to a list of watchers.

In practice defining watchers takes from less than a minute (simple check) up to five minutes (concatenated watchers with complex Boolean expressions). Again the requirements mentioned above are still valid.

During the test the code coverage is measured using the CTC-tool by Testwell. After your tests finished you get a report visualizing which lines of the c-code have been executed and which still miss. From this analysis you gain knowledge how to increase the code coverage and how the next stimulus has to be defined.

4. Conclusion and Results

This method has been applied to the 9G-Tronic development for a set of very complex modules having up to 100 inputs. This work was delegated to a test engineer who created test stimulus using the documentation of the control software. After one week of testing a code coverage of about 70% could be achieved and the tests were presented to the design engineer in charge. After discussing some details and one more week of testing the code coverage raised to about 90%. The module test in system context was well accepted because the handling was easy and fast.

For very simple modules the classic module test was better to use since all you have to do is fill up a spread-sheet with inputs and outputs. But with raising complexity the module test in system context was much more efficient in terms of test depth and time costs.

This last point still could be solved because the module test in system context is backward compatible to the classic module test. Using the spread-sheet with inputs and outputs one could generate everything needed to run the same test as a module test in system context.

The Authors

M.Sc. Tjark Kiefer
Daimler AG
Strategy and Coordination – Competence-Center Powertrain
(Strategie und Koordination – Kompetenzcenter Triebstrang)
Stuttgart

Dr. rer. nat. Dipl.-Math. Ingo Matheis
QTronic GmbH
Simulation and Emulation of Control Devices of the Drivetrain
(Simulation und Emulation von Steuergeräten im Bereich Antriebsstrang)
Stuttgart