embeddedworld2018
Exhibition&Conference
... it's a smarter world

Continuous Integration and Test from Module Level to Virtual System Level

Johannes Foufas, Martin Andreasson
Volvo Car Corporation
Gothenburg, Sweden

Michael Hartmann, Andreas Junghanns
QTronic GmbH
Berlin, Germany

***Abstract*— Software-in-the-Loop (SiL) is a test strategic sweet spot between Model-in-the-Loop (MiL) and Hardware-in-the-Loop (HiL) tests. We show in this paper how to use automatic C-code instrumentation to harness the superior properties of SiL technology for Module Tests even when the C-code is generated in a few, large controller functions combining the modules to be tested.**

Furthermore we show how to re-use module test specifications in integration and system tests by separating the test criteria from the test stimulus. We call these test criteria requirements watchers and define them as system invariants. This powerful technique, combined with efficiently handling large numbers of controller variants by annotating watchers and scripts, allows the automatic validation of hundreds of requirements in module, integration and system tests improving the software quality dramatically very early in the software development process.

Last but not least, we extend the idea of continuous integration to continuous validation to leverage all of the above to reach high levels of software maturity very early in the software development process. That will also benefit later test phases – like HiL system and system integration tests – by dramatically reducing commissioning efforts.

Keywords— Software-in-the-Loop; continuous integration

I. MOTIVATION AND CHALLENGES

Engineers are under pressure to deliver improvements at a growing pace while satisfying an increasing amount of regulatory pressures concerning performance, safety, reliability and ecology. The combination of more functionality and smaller turnaround times between new versions requires new methods of test and validation to keep software quality up to par. While traditional testing on the target hardware maintains a role in integration testing and satisfying strict safety norms, it is too slow, resource intensive and late with feedback for earlier phases of the control-software development cycle to increase robustness in a meaningful way.

Common Unit/Module test approaches rely on MiL which is prone to failure when looking for certain classes of bugs. SiL simulation can alleviate these concerns by providing a testable system that is much closer to the C-code reality: using the generated C-code, the target integer variable scaling and the (variant-coded) parameter values for the target system, often even including parts of the basic-software and communication stacks [1,2]. And despite being so close to reality, SiL is still offering all the strong points of MiL: cheap and early available execution platform (PC), determinism, flexibility when integrating into different simulations tools for example as FMUs, fully accessible and debuggable internals, easy automation for all system variants and many more benefits.

But moving to SiL is not without challenges. First and most obviously, hardware-dependent parts of the control software cannot be included and suitable SiL-abstractions have to replace the missing code. Recent, standardized software architectures, like AUTOSAR or ASAM MDX, ease such replacement and IO connectivity considerably as standard APIs can be provided by the SiL platform or standard description formats can be used to generate the connection layers like SiL-AUTOSAR-RTE generation from .arxml files. Even for pre-AUTOSAR ECUs this task can be handled quite efficiently these days: A limited number of tier-1-suppliers produced a limited number of vendor-dependent RTOS (inspired) architectures that allow for high-levels of reuse[3].

Another challenge is dealing with generated C-code for module test. The generated code is optimized for target use and may fuse many software modules into one large C-function (task). Stimulating individual software modules from the outside is not possible. Regenerating individual modules is out of the question, because changing the code generation process would lead to different C-code and therefore defeating the purpose of SiL: to test exactly the code that will be compiled for target without changes. The solution: we will instrument the generated C-code to gain control over all input variables to the module(s) under test.

Ideally one would like to reuse tests from MiL to SiL to HiL. However, the different levels of simulation detail, restrictions on measurement bandwidth, availability of the execution platforms, setup cost for different variants,… requires a more sophisticated test strategy than "simple reuse". Focusing on the strength of each platform and running each test as early as possible will frontload, as one example, application layer function and integration tests to SiL, while leaving hardware related diagnostic tests on the HiL platform. And optimizing control strategies as early as possible will move these tasks to MiL simulations. Re-using test definitions is therefore limited by the different test goals and platform-related restrictions.

However, module and system-level tests can still share the same requirements, if not the same test focus. The solution to high levels of reuse for test specifications is separating the

implementation of the requirements tests from the stimulus. While classic test automation combines test stimulus and requirement tests into the same script, we define requirement watchers as formal, stimulus and system-state independent invariants: conditions that must always hold. Engineers need to spend more time and care in writing such requirement watchers, but the payoff justifies this extra effort: Requirement watchers can be tested with any kind of stimulus be it scripted, field measurements, short test vectors, hour-long load-collective simulations or auto-generated test stimulus (e.g. by TestWeaver)[4]. Here we will show how to reuse module requirements defined for module testing in system-level testing when written as requirement watchers.

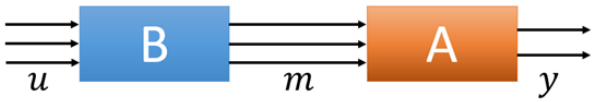
Increasing number of variants of control systems requires special measures during test and validation to reduce manual matching of test cases to variants of the control software. We show how annotating requirement watchers and stimuli with filter properties enables automatic selection of relevant test

Continuous Integration is a state-of-the-art method to detect integration problems. Combining CI with more than rudimentary tests is difficult if the target binary is the test object. Using SiL as execution platform allows high levels of automation for large numbers of tests because they can run on the same platform as the build process: the PC. Extending the idea of Continuous Integration (nightly builds) to Continuous Validation (nightly test) improves early detection of large classes of software problems considerably.

II. Testing at Volvo Cars Corporation

At Volvo Cars Corporation (VCC), SiL testing is at the core of a new Continuous Integration strategy. Through increasing the frequency of integration points and corresponding tests, control software reaches a higher level of maturity when final acceptance tests are carried out close to production. In order to achieve this, a large number of tests need to be defined and used throughout the development process.

One concern so far has been the incompatibility of test cases and stimuli between MiL and SiL setups. The structure that is designed by a developer in modeling tools is often disregarded during code generation. This means testing is limited to module level, with modules growing in scope over time. Developers on the other hand design around smaller units represented through subsystems.



$$y(t) = A\left(B\left(u(t)\right)\right)$$

Fig. 1: Basic module with subfunctions

The difficulty in test design for large models can be illustrated by the simple example in Figure 1. Subfunction A is defined by a set of requirements that define the behavior of the outputs (y) as a function of the intermediate signals (m). Historically, testing these requirements in anything other than MiL simulation would require the engineer to invert Subfunction B in order to design the correct set of inputs (u) for the test.

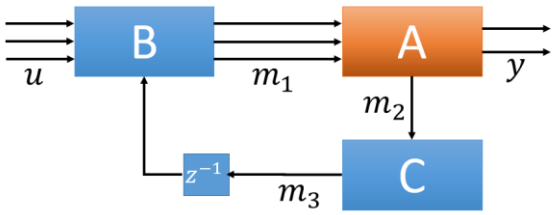


Fig. 2: Function requiring transient stimuli

For more complex modules, this approach is very costly and error-prone. As loops inside functions and state diagrams are introduced, tests for simple functionality require increasingly complicated transient stimuli.

We aim to present an instrumentation approach that offers the opportunity to bypass parts of a function and allows developers to define stimuli and test criteria around arbitrarily small subfunctions of a module.

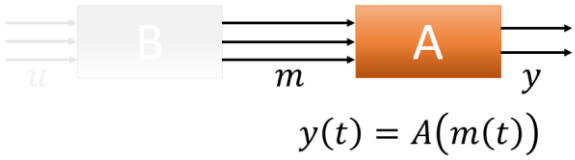


Fig. 3: System under test in with bypassing: Test stimuli can be definded as m(t)

The requirements that are defined using this process shall remain independent of the stimulus and usable throughout all levels of testing, up to integration and robustness tests.

III. Instrumentation Approach

Modelling tools like Simulink allow developers to structure their models into subsystems which can be used like atomic blocks. The subsystem can be copied or moved freely across models and can be tested independently in MiL. When generating code from a model using TargetLink, the complete model is represented by a single C method. Statements that are generated from blocks within a subsystem are spread across the entire compilation unit. This means a subsystem cannot be executed on its own, preventing any kind of meaningful unit testing. To remove this limitation from SiL tests, we analyze the resulting C-code and inject bypass opportunities wherever a measurable signal is written.

The injected code remains inactive unless the source is compiled for a SiL target and the user enables bypassing for

the respective variable. This way, MISRA compliance of production software is ensured even if the instrumented code makes it into release builds on accident.

As code generators tend to use temporary, local variables where signals are not specifically made measurable, further analysis of the generated code is necessary. In cases where such a temporary variable is always equal to a measurable signal, it has to be set to the correct value as well. This specifically applies to signals transcending subsystem borders, which can be represented by two different variables in code.

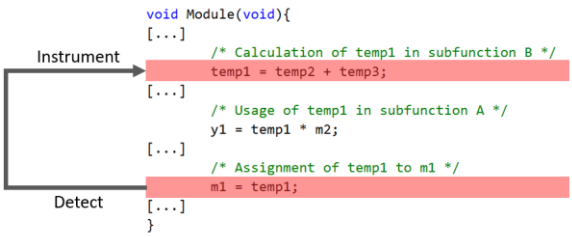


Fig. 4: Instrumentation of temporary variables

State Machines can by bypassed entirely so no transitions are necessary to provide the system under test with the correct state and/or corresponding flags.

After the code is instrumented, the virtual basic software is automatically set up with regards to task scheduling and supplier-dependent modifications. Compilation results in a virtual ECU containing the entire OEM-part of the control software which can be coupled with a plant model and/or other ECUs for system-level simulation.

Without recompilation, engineers can trim the V-ECU to fit their use-case. Depending on a specification file provided by the user, the Virtual ECU will reconfigure its scheduler to only execute a subset of the included functions. The same specification can be extended by a detailed interface specification listing the ports of a subsystem. If this specification is present, all bypasses on the input side are activated and the variables are overwritten by stimuli during simulation.

IV. Design of Stimulus-Independent Tests

The instrumentation method described reduces the effort in test design significantly. Unit-Tests of small subfunctions can be created through traditional scripting and deployed as part of an automated test framework. While this method can produce comprehensive results in regards to verification and coverage, it relies heavily on developers being able to foresee all possible problems.

During the specification phase, requirements are written in a broad scope. Often a requirement will define a certain behavior that shall be true under certain conditions. In essence:

Condition A => Behavior B

Defining test cases around such requirements would be difficult, especially if the condition contains several continuous signals. The widespread approach of testing by creating a stimulus and checking for a specific reaction fails to capture a large number of possible scenarios as engineering hours and therefore the number of defined test cases are limited.

In addition, a stimulus-reaction based test becomes obsolete once the object under test is integrated into a system, as the previously defined stimulus often cannot be reproduced due to its artificial nature.

Side effects that appear based on the interaction of several components cannot be tested. A developer might cover all the expected combinations of outputs from another module or sub-function, but faulty signals as result of a bug in this module might not be considered.

TestWeaver by QTronic provides the means to define requirements in a way that closely resembles the original specification. The test for a requirement is defined by precondition and expected behavior instead of stimulus and reaction.

The definition of a requirement watcher entails conditions to activate the instrument and the criteria to be tested. A watcher intended to test the simple example above would remain inactive until Condition A is met and once becoming active check for the Behavior B.

For more complex cases additional options such as tolerance times can be specified. Inverse usage, i.e. the specification of unwanted behavior is also supported.

Each requirement can be tested at every point in time during a simulation. Requirements defined at subsystem level remain valid in system context and vice versa and can be tested in regardless of scope. As requirement watchers do not require write-access to any signals, the definitions implemented for unit tests are still applicable in larger contexts where the code instrumentation might be omitted. Module and integration tests can thus be executed on final production code.

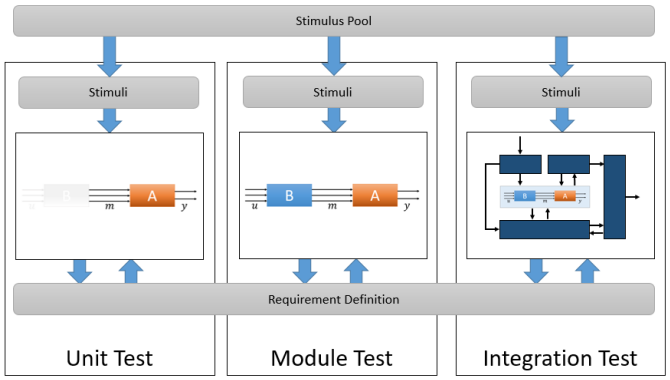


Fig 5: Requirement watchers can be reused throughout and refined based upon different scopes. All requirements are tested against at every point. Stimuli are selected from a pool where applicable.

Code coverage is measured with Testwell's CTC++. The decoupled requirements described above provide the option to use any input vector to increase coverage. Any scripts or measurements that are available can be added to the stimulus pool and simulated. This way, high code coverage can be

achieved without specifically designing additional tests. The requirement definitions can also be reused with TestWeaver's scenario generation for focused explorative tests, further increasing coverage and robustness.

V. Continuous Integration and Verification

At VCC Powertrain, code is deployed to a Jenkins-based continuous integration system. Pipelines are defined to automatically build virtual ECUs and run applicable tests. Commits by function developers into the common model base trigger the execution of interface verification and module tests as well as integration tests relevant to the Module in SiL and HiL.

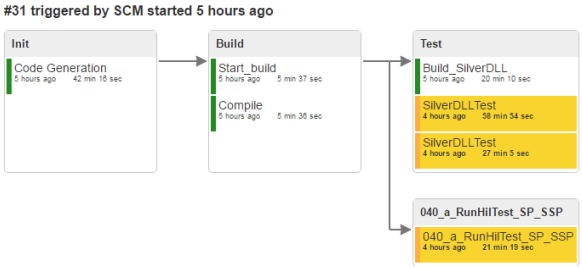


Fig 6: A typical Jenkins Pipeline.

As a result, function developers get quick and reliable feedback about the behavior of their models in the context of a wider system. To verify and keep track of code quality and open issues, the full test suite is executed nightly.

VI. Conclusion

In this paper we present a number of critical building blocks necessary to improve software maturity early in the software development process. Software-in-the-Loop (SiL) allows test execution in a Continuous Validation process of the target C-code. Instrumentation of the target C-code allows manipulation of any input of the software module enabling module tests even if target code generation merges many modules into larger C-functions (tasks).

When expressing module and system requirements as requirement watchers we can reuse these more easily in most of the test stages more than compensating for the extra effort defining requirements as invariants.

Annotating requirement watchers and stimulation scripts with variant information allows automatic filtering to matching ECU configurations. This way, a single test database can be used to handle a multitude of variants and at the same time ensuring all relevant requirements will be tested on all variants during all test stimuli reaching code-coverage and requirement-coverage goals more quickly and more easily than with traditional test methods.

As the virtual ECU can be reconfigured within Silver to include or exclude any function in the entire application software, build times are kept to a minimum.

In order to reduce the amount of work needed to design tests even further, closed loop simulations including detailed plant models will be integrated into the VCC CI and CT toolchain. Reusing the existing requirement watchers, TestWeaver's scenario generation will be employed in order to increase robustness and test coverage even further.

[1] Brückmann, Strenkert, Keller, Wiesner, Junghanns: Model-based Development of a Dual-Clutch Transmission using Rapid Prototyping and SiL. International VDI Congress Transmissions in Vehicles 2009, 30.06.-01-07.2009, Friedrichshafen, Germany

[2] Rui Gaspar, Benno Wiesner, Gunther Bauer: Virtualizing the TCU of BMW's 8 speed transmission, 10th Symposium on Automotive Powertrain Control Systems, 11. - 12.09.2014, Berlin, Germany

[3] René Linssen, Frank Uphaus, Jakob Maus: Software-in-the-Loop at the junction of software development and drivability calibration, 16th Stuttgart International Symposium (FKFS), 15. - 16.03.2016, Stuttgart, Germany

[4] Mugur Tatar: Enhancing the test and validation of complex systems with automated search for critical situations, VDA Automotive SYS Conference, 06. - 08.07.2016, Berlin, Germany