



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 validtws

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

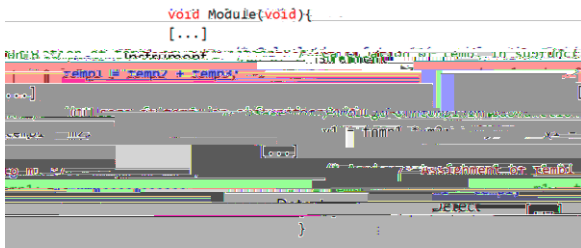
offering all the strong points of MiL: cheap and early available execution platform (PC), determinism, flexibility when integrating into different simulations tools for example as FMUs, fully accessible and debuggable internals, easy automation for all system variants and many more benefits.

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.

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.



```
void Module(void){
    [...]
    temp1 = temp1 + temp2;
    [...]
}
```

Fig. 4: Instrumentation of temporary variables

State Machines can be 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

achieved without specifically designing additional tests. The 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-