Virtual Hardware “In-the-Loop”: Earlier Testing for Automotive Applications

Abstract
This whitepaper is the first one in a series of publications that will describe the concept of virtual hardware “in-the-loop” (vHIL). The goal of vHIL is to frontload the testing process by enabling software teams to create and run their software tests before the actual ECU hardware is available. Higher quality tests, higher quality software and a more streamlined “in-the-loop” flow is the intended outcome of this solution. Besides describing how vHIL fits in the general Model-In-the-Loop (MIL) – Software-In-the-Loop (SIL) – Hardware-In-the-Loop (HIL) process, this first whitepaper describes in more detail how a virtual prototype model created with Synopsys Virtualizer can be integrated with a MathWorks Simulink plant model. This integration is fundamental for enabling a vHIL solution.

The Testing Challenge
Embedded software quality is one of the biggest concerns of the automotive industry. There are thousands of known errors when the integration of the embedded software stacks on the final ECU starts. This is just the beginning, before new errors are uncovered during the subsequent testing phases. With the development timelines shortening and the time-to-market becoming increasingly important in automotive, software testing is becoming the biggest bottleneck in the development cycle. Improving the quality of the embedded software by increasing the number of tests run during the testing phases conflicts with the fact that testing is already taking most of the resources and time available in a project. Testing is 75% of the software development cost and this number is increasing. In most automotive companies hardware/software integration happens during a customer project, known as a variant. Variant testing and bug fixing horizontally influence the complete software platform (i.e. common software libraries shared across an organization) and introduce combinatorial complexity to testing. For instance, a single vehicle can have tens of variants (low/mid/high end, country specific, etc.) that can lead to hundreds of thousands of tests for a single software platform. Fixing a bug in one variant can have trickle-down effect on multiple other configurations. The problem is overwhelming.

Automotive companies integrate and test the system functionality of a new application (e.g. powertrain domain) by connecting the ECU to a physical prototype of, for instance, the engine (in this context the engine is also known as the plant). The ECU and the plant are connected in a closed loop, where the ECU actuates over the different elements of the engine (e.g. spark plugs) and at the same time monitors the response of the plant via sensors (e.g. crankshaft speed). Following this approach, ECU testing (mainly the application software) can only start after a very expensive prototype of the plant is available. Moreover, if this is the first time when all the pieces are integrated with each other, there is a high risk that the plant could be damaged if the software running on the ECU encounters a bug. In this way, testing happens very
late and requires a huge effort. In order to overcome these limitations the automotive industry developed the concept of Hardware-in-the-Loop (HiL) simulation, where the physical plant is replaced by a simulation model (a.k.a. the plant model). To connect the ECU and the plant model in a closed-loop, the plant model is executed on a HiL simulator box. This HiL simulator box speeds up the simulation of the plant model to real-time. Today’s more advanced testing environments use HiL, hence avoiding the risk of damaging the plant, but more importantly allow testing to happen earlier.

This approach, however, is hardly sufficient to cope with the increasing requirements in quality, development time and cost of today’s automotive applications. Breaking the dependency on physical plant prototypes moved the testing bottleneck to two other areas. First, software teams have limited access to HiL environments. Due to cost, most HiL environments are time-shared among multiple integration and testing teams. Every team will require the HiL with a slightly different configuration, which is error prone and time consuming to setup. Second, testing has to wait until an ECU prototype is available and the software is integrated and up and running to a level where testing can start. Fortunately, the dependency on physical ECU prototypes can be removed as well by using fast simulation models of the ECU hardware that enable execution of unmodified embedded software. Such fast simulation models are also known as virtual prototypes (VP). A great advantage of virtual prototypes for software testing is the fact that they can scale easier across multiple teams and locations, while reusing most of the software tools and plant models used in the HiL environments. Virtual prototyping is not intended as a replacement for HiL testing, but as an upfront testing milestone that can help increase the quality of the tests as well as the quality of the embedded software itself.

**Conventional “In-the-Loop” Methods**

HiL testing has gained acceptance in lock step with the increased adoption of Model-Based Development (MBD) in automotive. MBD and in-the-loop methods are widely used today in automotive to develop and test control strategies. A complete in-the-loop flow is composed of several chronological steps as illustrated by Figure 1. First, on the left side of the V cycle, Model-in-the-Loop (MIL) is applied, followed by Software-in-the-Loop (SIL). Second, on the right side of the V cycle, Hardware-in-the-Loop (HIL) is applied. Sometimes, Processor-in-the-Loop (PIL) is applied after SIL, although it is not that common.

![Figure 1: V-cycle (MIL, SIL, HIL)](image)

MIL consists of mixing the physical plant model (including mechanical, electrical, thermal, etc., effects) with the control strategy at the algorithmic level (typically using state machines). This creates a complete mechatronic model that can be simulated together to find out whether the behavior of the control logic is correct. Very often a tool that is used for this is Simulink from The MathWorks. Initial tests can be created at
this level. These tests should be reusable for the next subsequent phases. Once the MIL simulation shows that the results are correct, the control part is implemented (typically in software). Implementing the control model can be done manually by writing C code or automatically by using code generation tools.

SIL simulation is then used to validate the implementation of the control logic. With SIL, the C code is simulated together with the same physical plant model used in the MIL phase. The same tests can be reused in this phase to assure that the results are equivalent (back-to-back testing). Typically, code coverage at the unit testing level is measured in this phase. The problem with SIL is that the C code is compiled for the host PC instead of the target microcontroller (MCU). This may introduce differences in precision due to different data types (saturation and overflow effects), as well as other problems due to resource limitations on the MCU (memory and processing power).

PIL is used to overcome some of the limitations of SIL. PIL still focuses on the control algorithmic implementation, but this time the C code is compiled for the targeted processor architecture. The execution of the software code is synchronized with the simulated physical plant in order to reuse the same tests. Memory allocation and execution time can be measured at this point. The main problem of PIL is that in reality a control function is not targeted to a single MCU architecture, but too many different MCUs depending on the specific requirements and variant of the end product. Combined with the fact that people doing SIL are not familiar or keen to work with hardware environments this approach has seen limited adoption.

Next step in the flow is HIL. This however does not happen as a smooth continuation of the previous phases. Before HIL testing can start, the control function has to be integrated with many other control functions, plus many other different software layers (OS, drivers, middleware). This has to a big extent been standardized by AUTOSAR and now software integration is a more structured process than it was before. Although the process is more structured, it does not mean that it has become trivial. Having a first instance of the software stack that can be compiled is still a daunting task. And even if the software stack compiles, this does not mean that it will actually work for the intended MCU and ECU. There are quite a few iterations (and loops with the previous phases) that must be done by the software integration teams before the ECU testing using HIL can start. Some of the previous tests created during the MIL and SIL phases can still be applied here (sometimes with some modifications), but the big bulk of test creation and execution can only happen at this point.

**Benefits of virtual hardware “in-the-loop”**

A VP is a fast simulation model of the digital ECU/MCU hardware. This model enables fast simulation, while being able to execute exactly the same binary software as the target MCU. Because a virtual prototype is a software simulation model it provides extra controllability and visibility capabilities, which are extremely useful for debugging, analysis and automated testing. VPs are easier to use, to distribute across multiple teams and to archive compared to real hardware.

Virtual hardware “in-the-loop” (vHIL) using a VP can add multiple benefits to a standard automotive development flow. In the context of an “in-the-loop” methodology, a VP removes the dependency on the ECU hardware and enables earlier software integration and testing. This creates a smoother path between MIL, SIL and HIL. With a vHIL environment, a virtual ECU model is connected in closed-loop to the same plant model that runs, for instance, on Simulink. The same tests applied on the MIL and SIL phases can be now executed on the vHIL environment. More importantly new tests can now be created by the software testing teams without having to wait for an ECU prototype or having to allocate time on a HIL environment. This is sometimes called offline test creation. Frontloading the creation and execution of the tests allows for developing more tests, but has also immense benefits in terms of increasing the quality of the tests. Together with code coverage measurement on the VP, running the tests on the vHIL environment can help to identify what code the tests are covering and what part of the code is not yet exercised by the tests. Code coverage is a standard capability of the VP tools. Moreover, since a VP is more controllable and offers more visibility than hardware, directed tests covering corner cases are now easier to create and will reproduce the exact same conditions every time they run.
All in all, increasing the number of tests and the quality of the tests has a direct influence on the quality of the embedded software itself. This means that a much more mature software stack can be integrated with the real ECU, reducing the bring-up time from weeks to days. As a result more time can be allocated to perform ECU and system testing in a HIL environment. As indicated before, vHIL is not intended as a replacement for existing “in-the-loop” methods but as a complementary new milestone to create a more mature and efficient development process.

The rest of this whitepaper will focus on how a VP can be linked to a Simulink plant model to create a vHIL environment.

**Virtualizer-Simulink Integration Library**

In this chapter the integration between our virtual prototyping toolset (Virtualizer™) and Simulink using the Virtualizer-Simulink Integration (VSI) library is explained. Official documentation for the VSI library can be found on the Virtualizer tool installation package.

Virtualizer is the tool used to create a simulation model of the digital ECU hardware. The main digital element of an ECU is the MCU. An MCU is composed of the processor core(s), on-chip memories and bus interconnect, as well as internal and external peripherals. Software executes on the MCU and communicates with other blocks in the ECU or outside the ECU through dedicated interfaces (e.g. CAN, SPI, ADC, PWM, etc). Virtualizer libraries contain off-the-shelf MCUs from the main semiconductor companies in the automotive space (e.g. Renesas, Freescale, Infineon). The electrical, analogue and mixed-signal parts of an ECU (e.g. signal conditioning) must be either abstracted away or modeled in a different tool (e.g. Simulink).

Simulink is used to create the plant model representing the physical system (e.g. engine, transmission, vehicle dynamics, etc.). The electrical parts of the ECU have to be modeled in Simulink as well. This can however typically be done at a fairly high level. This section is not intended to explain how to use Simulink to create a plant model, but instead how to modify an existing one to establish the link to Virtualizer.

There are two options to connect Virtualizer and Simulink models: co-simulation and export.

- With the co-simulation option, both tools need to be open simultaneously and executing in a synchronized fashion, communicating values based on a fixed-step or variable rate (depending on the solver used by the Simulink plant model). The current integration library implements the co-simulation method using a shared memory scheme that is extremely efficient, with the caveat that both tools have to obviously run on the same host.

- The second option is exporting the Simulink model to be used inside Virtualizer. With this option, a SystemC model that contains the plant model will be generated from Simulink using the Simulink Coder tool (former Real Time Workshop). This SystemC model can be imported in Virtualizer without effort, since all the required scripts and files are automatically generated by Simulink Coder. Once the plant model is imported, it can be connected to the rest of the ECU modeled in Virtualizer. A single simulation will be created out of the two types of models.

When to use co-simulation and when to use export depends on many factors. As a rule of thumb, we propose users to first use co-simulation, since any problems that may arise can be debugged more easily in each model’s original tool. Once the integration works as expected, the export option can be used to generate a single simulation that will run faster and will be easier to use in a testing environment. Fortunately the same plant model modified to connect to Virtualizer can be used in both co-simulation and export mode without further changes.

**The Simulink Side**

In order to connect Virtualizer and Simulink, the VSI library blocks have to be used. There are two sets of equivalent VSI blocks: one set in the Simulink toolbox libraries Figure 2 (left) and one set in the Virtualizer libraries Figure 3.
The Simulink VSI library blocks contain six blocks. What block to use depends on the data type and direction of the communication. These blocks are:

- (Double/Vector/Bit)Out: These blocks will receive a value with (double/unsigned int/boolean) data type from Virtualizer.
- (Double/Vector/Bit)In: These blocks will send a value with (double/unsigned int/boolean) data type to Virtualizer.

When using the co-simulation flow, there is nothing else to do from the Simulink side besides instantiating and connecting the VSI blocks to the rest of the plant model.

When using the export flow, however, there are few more things to configure in the “code generation” options Figure 2 (right). More specifically the “system target file”, which has to be set to one of the SystemC options (depending on the host platform). This will automatically configure the other makefile options. After generating the code and building the model library, the plant model is ready to be imported on Virtualizer.

The Virtualizer Side

In the co-simulation flow, the VSI library is also used on Virtualizer. This library also contains six different blocks, depending on the data type and direction of the communication. These blocks are:

- vsi_(double/vector/bool)_in: These blocks will send a value of (double/unsigned int/boolean) data type to Simulink.
- vsi_(double/vector/bool)_out: These blocks will receive a value of (double/unsigned int/boolean) data type from Simulink.

After instantiating the blocks and connecting them to a SystemC design, the last step is to enable the link to Simulink. This is done per instance by configuring the “SimulinkPath” property of the instance to match the instance name of the Simulink VSI block. More details are explained with an example in the next section.
When using the export flow, the VSI library is not needed on the Virtualizer side. Instead the complete plant model will be imported as a SystemC block. This block will have a SystemC connection port (sc_in or sc_out with the appropriate data type) per VSI block used on the Simulink plant model. More details are explained with an example in the next section.

Case Study: Automatic Transmission Controller
The example used in this case study has been modified from one of the original examples provided with the Simulink installation. This example is called “Modeling an Automatic Transmission Controller” and it can be found as part of the Simulink automotive applications demo. The top level view of the original model is shown in Figure 4. This model includes four subsystems: Engine, Transmission, Vehicle and ShiftLogic. Moreover, the driver input (throttle and brake) is modeled as a ManeuversGUI block. Let's, from a high level point of view, look at how this model works. The Engine block receives the throttle and impeller torque information and calculates the output RPM (revolutions per minute) of the engine. The Transmission block receives information regarding the engine RPM, the selected gear and the RPM of the transmission (part of the Vehicle dynamics block) and generates the impeller torque information for the engine and the output torque to the wheels. A closed-loop exists between the Engine and the Transmission so changes to the engine RPM will affect the impeller torque (depending on the gear), which in turn will affect the engine RPM. Finally, the automatic gear selection happens on the ShiftLogic block, which has been modeled using Stateflow. This control logic is typically implemented as software and runs on the TCU (Transmission Control Unit).

In our case study, we will modify the Simulink model by removing the ShiftLogic block and replacing it with a connection to Virtualizer. In Virtualizer, we model the digital part of the TCU (i.e. the MCU) that runs the embedded software implementation of the control logic algorithm. Both co-simulation and export flows will be discussed in the case study.

Modifying the Simulink Model
Adapting the Simulink model requires several steps:

- First step is to remove the ShiftLogic block from the Simulink model and replace it with VSI connector blocks. Instead of making the connection directly at the top level, a hierarchical block named vECU (virtual ECU) is added. The engine RPM value is connected to the vECU block as an input and a gear output that connects to the Transmission block is also created. This is illustrated in Figure 5.
Second step is to adapt the engine RPM and gear signal from the “real” data types used in Simulink to the “digital” data types used in Virtualizer. Basically, the signal conditioning blocks that exist on the real ECU are modeled at a very high level, abstracting away the detailed electrical and analogue-mixed signal effects. We could have chosen to model those effects using Simulink, but to simplify the amount of effects to consider in this case study we decided not to. The engine RPM, which is a real value in the range 0 to 6500, is normalized to the 12-bit (0 to 4096) output of the ADC analogue part. The ADC digital part is modeled in Virtualizer as part of the MCU. This means that all voltage information and conversion effects are in fact abstracted away. The normalized engine RPM signal is then sent to Virtualizer through a VectorOut VSI block, called rpm_digital. On the other side, the gear information will arrive from Virtualizer to Simulink through three BitIn VSI blocks (gear_0, gear_1 and gear_2). These BitIn blocks will each carry a boolean value, coming from a GPIO pin on the MCU. The ConversionB_D block will assign different weights to each bit and create a single real value that connects to the output. This is all shown in Figure 6.
Third step is to configure the options for the simulation time (0 to 5 seconds), solver (fixed-step with sample time 0.001 seconds) and select (or create) an input scenario on the ManeuverGUI block. For this example the created input scenario is shown in Figure 7. In this scenario the break (up) is not pressed, but the throttle (down) is increased from around 70% to 100% in 2.5 seconds, then released to 0% and pushed again to 100% until the end of the scenario (5 seconds).

![Figure 7: Input scenario](image)

With these three steps the co-simulation flow is ready. We can at this point also generate a SystemC model that contains the plant model (export flow). For that case, a Simulink Coder option is needed. Exporting the plant model is similar to what was described in the previous chapter.

**Creating a virtual ECU in Virtualizer**

As explained before, in Virtualizer only the digital part of the ECU needs to be modeled. First a hierarchical block named “ECU” is created, as it was done in Simulink. The main block of the ECU is the MCU, where the software will run on. For the MCU, a model of the Freescale MPC5643L “Leopard” dual-core device is used. The model can be dragged and dropped from the Virtualizer libraries in the canvas belonging to the ECU block. This virtual MCU model mimics the “Leopard” architecture, including the two e200z4 processor cores, FLASH and SRAM memories, on-chip peripherals (e.g. INTC, STM, etc) and the off-chip peripherals (e.g. ADC, SIUL, FlexCAN, etc). The grade of fidelity of the model is such that the same embedded software that runs on the actual hardware can run on the virtual model and will result in the same functionality. Figure 8 shows the Leopard block diagram (left) and the MCU top level view in Virtualizer (right). Besides the main MCU block, a clock generator block emulating the external clock (EXTAL) and a reset block emulating the external reset are added to the ECU block.
Co-simulation flow

For the scenario where we want to connect Virtualizer to Simulink using co-simulation, we follow these steps:

- First, the VSI library (named SIMULINK_CONNECTOR_BLOCKS) is opened.
- To communicate with the Simulink plant model we need to:
  - Drive the `rpm_digital` signal in Simulink to one of the ADC converter pins in the Leopard MCU. We choose the `ADC_1 input 0` that correspond with the one of the pins on the MCU model. A "vsi_vector_out" block is instantiated within the ECU hierarchy level next to the Leopard model and connected to the pin.
• Drive three GPIO pins (part of the SIUL peripheral in Leopard) from Virtualizer to Simulink for the gear value. We select the pins PAD_A[0..2] for the gear_[0..2] signals on the Simulink side, which are connected to the SIUL. Three “vsi_bool_in” blocks are added and connected in Virtualizer.

![Figure 9: Setting up the co-simulation](image)

The last step to setup the co-simulation is to configure the VSI blocks to find the correspondent blocks in Simulink. This is done by configuring the “SimulinkPath” property on each block with the name of the corresponding block in Simulink. For instance, the three blocks connected to the SIUL (independently of how they are named) must configure the “SimulinkPath” property to “gear_0”, “gear_1” and “gear_2” respectively. This is show in Figure 9 for one of the VSI blocks.

Once this is done, the design can be exported and built. The result will be a simulation executable that will look for connecting to Simulink before starting.

**Export flow**

In case of the export flow the VSI library blocks are not needed. Instead the generated plant model will be imported in Virtualizer. For that, Simulink Coder will create a Virtualizer import script. The script can be run from the Virtualizer console or through the “Tools->Execute Script” menu. Figure 10 shows the imported model connected to the MCU. After connecting the plant model to the MCU, the design can be exported and built. The result will be a simulation executable that will directly run the two models together.

![Figure 10: Imported plant model](image)
System Analysis

In this section the results of running both the Simulink plant model and the Virtualizer ECU together are explained. Let’s first look at what the software running on the Leopard MCU is expected to do. Figure 11 shows the Freescale CodeWarrior compiler and the main “control_plant” function that interacts with the plant model. The “control_plant” function is triggered periodically every 6 milliseconds. This is controlled by the STM block (system timer) part of the MCU. The first step in the code is to read the RPM value out of the ADC_1 channel 0. After reading the digital sample, the value is adapted from the 12-bit digital representation to physical RPM information. So the signal is now back to the range between 0 and 6500 RPM. After doing some basic error checking (diagnostic code) the ADC value is compared to two value thresholds. If the value is above 4000 RPM the gear will be increased (if the system is not in the higher gear already). If the value is below 2000 the gear will be decreased (if the system is not in the lower gear already). After the gear is modified, the “control_plant” function is not triggered anymore until a safe margin time is expired. This time is controlled by the calibration parameter ITR_PAUSE_AFTER_SHIFT: switching too quickly might damage the transmission gearbox, switching too slowly might keep the engine too long at high revolutions. This calibration is very important for safety reasons; plus, of course, for finding the right spot that offers the intended driver experience.

![Figure 11: Embedded software in CodeWarrior](image1)

The high level results of the interaction between the embedded software running on the virtual ECU and the plant model can be observed in Simulink. Figure 12 shows the impeller RPM results (up) correlated with the gear output (down) of the vECU block. The closed-loop effects are observed on the quicker reduction of RPM when a gear is increased and vice versa.

![Figure 12: System analysis results in Simulink](image2)
This view does however not help to understand what the software and the MCU hardware are exactly doing when running the scenario. This is especially important in case the system does not behave as expected and the plant model, MCU hardware and embedded software need to be analyzed together. Figure 13 shows the type of correlated system analysis that Virtualizer is able to provide. The impeller RPM digitalized signal can be traced together with the output pins that select the gear. The software functions can also be traced together and the effects of the ITR_PAUSE_AFTER_SHIFT calibration parameter can be observed by looking at the spaces in the trace.

![Figure 13: System analysis results in Virtualizer](image)

More hardware information like, for instance, timer interrupts, ADC status, ADC register traces, etc can be traced and added to the view. Figure 14 shows how after the STM interrupt happens the “control_plant” function is called, which in turn calls the “read_from_ADC_1_channel_0” function. This function, as observed on the ADC register trace and status, interacts with the different ADC_1 registers to begin the reading of a sample. In this specific case the value has reached the upper threshold value and the gear is increased (from 1 to 2) through the “set_gear” function.

![Figure 14: Zoomed in view with STM and ADC hardware information](image)

Virtualizer can act as a virtual logic analyzer that can trace software and hardware events with unlimited buffer size or number of events. This is extremely useful to debug low level hardware/software integration issues across the multiple layers of the software stack and the MCU hardware.
Summary
This whitepaper describes the mechanisms to integrate Virtualizer and Simulink models. Connecting together a model of a digital MCU (in Virtualizer) to a plant model of the physical system (in Simulink) is the first step towards a virtual Hardware-in-the-Loop solution. Being based on simulation models, vHIL removes the dependency on the ECU hardware to start integration and testing. Earlier test creation, automated test execution, code coverage measurements, etc., are some of the benefits of vHIL using Virtualizer. Moreover, its great debugging capabilities allows for finding bugs on failed tests with less effort and in shorter time than with conventional methods. Overall, vHIL is the natural step to smoothly bridge the MIL/SIL design phases to the latter HIL testing phase.

References
http://www.synopsys.com/Systems/Pages/default.aspx


3 "This Car Runs on Code", R. Charette, IEEE Spectrum, February 2009
(http://spectrum.ieee.org/green-tech/advanced-cars/this-car-runs-on-code#)


5 MPC564xL: Qorivva 32-bit MCU for Chassis and Safety Applications,
http://www.freescale.com/webapp/sps/site/prod_summary.jsp?code=MPC564xL