Towards Virtual Validation of ECU Software using FMI

Lars Mikelsons[1] Roland Samlaus[1]

[1]Robert Bosch GmbH

Abstract

Connected, Automated, Electrified. These three trends in the automotive industry require rethinking of the use of simulation respectively models. The use of models for evaluation of new concepts or stimulating the unit-under-test (in HiL testing), already firmly rooted in the development process of software functions, will not be sufficient to realize visions like autonomous driving or update-over-the-air. One key enabler for such technologies is virtual validation, i.e. the validation or release of software functions in a pure virtual setup. That is, simulation is not only a tool to shorten the development cycle, but one of the key technologies to release future software functions, e.g. highly automated or autonomous driving. In this contribution a feasibility study for the validation of FMI-based virtual ECUs (vECUs) in a co-simulation setup is presented. Thereby, the powertrain and the vECU are represented by FMUs, while the tool CarMaker is used for vehicle dynamics. On the base of the gained experience requirements for the FMI standard are formulated that would allow to go for virtual validation of future software functions. *Keywords: FMI, virtual validation, ECU, vECU, autonomous driving*

1 Introduction

The use of models and simulation is firmly rooted in the development process of automotive software as well as hardware components. However, in the development of software functions typically simulation is mostly used to evaluate new concepts or to stimulate the unit-under-test, e.g. HiL in testing. More precisely, the model of a a software or hardware component is typically used during development (Junghanns et al., 2014). Models and simulation are rarely used for virtual validation, i.e. validation or even release of a software function in a pure virtual setup. There exist examples where software validation was done virtually, e.g. ESC homologation (Holzmann et al., 2012). However, the validation of ECU software is mostly performed using real prototypes. In fact, although in many cases models are exchanged between OEMs and suppliers, it is not a standard workflow to use them for the application of software functions. While for "‘old fashioned"’ software functions not using existing models may lead to a more costs, not using models is not an options when it comes to concepts like autonomous driving or update-over-the-air. According to (Wachenfeld and Winner, 2015) and (Winner et al., 2010), following basic statistics, between 100 million and 5 billion kilometers of test driving are required in order to ensure that software for autonomous driving is at least as save as a human driver. Note that, the test procedure has to be repeated after every single update or modification. Clearly, it is not possible to use real prototypes for those test drives due to required time and costs (Google states that its 20 self driving cars drive 16.000 kilometers per week (goo, 2015)). The same argumentation holds for update-over-the-air except that typically the problem arises from the number of variants and configurations that need to be tested. Thus, here virtual validation has to be employed. Typically, for technologies like autonomous driving one has to couple models from different domains (xDomain vehicle simulation), e.g. powertrain, vehicle dynamics or powernet. One approach for xDomain vehicle simulation is to use Modelica in order to model all involved domians in the same tool respectively language. Though, in big companies the models for the different domains are generated in different business units that prefer different simulation tools (best suited for their specific problems). Hence, co-simulation is typically the way to go. Designing such a co-simulation setup for virtual validation leads to several challenges. Typical questions that arise are

- What is the required level of detail for my models?
- How do I parametrize my models?
- How to validate a model?
- How big is the discretization error?
- How big is the coupling error?
- How can I integrate the software code into the simulation?
- Which portions of the ECU code do I have to integrate (where to cut)?

In this contribution only the last two questions are focused. In fact, this contribution presents a feasibility study for integrating ECU code as an FMU into a co-simulation setup. Thus it shows a possibility to integrate ECU code (including the formulation of further requirements on FMI) and discusses the problem of identifying the portion of the software stack required for a specific validation task. Note that, the used software function is part of a function for highly automated driving (HAD). Future work aims at treating this HAD function as sketched in this paper.

In industry there are different meanings for vECU. Some people just mean cross-compiled application code, others mean ECU code running on a virtual OS and last but not least a vECU can also include virtual hardware. In section 2 a brief overview is given and the used approach for the vECU used in section 3 is described. In section 3 the co-simulation setup and generated results are discussed. Starting from a yaw rate controller implemented in ASCET a vECU is generated. This vECU, is then integrated in a co-simulation setup consisting of Model.CONNECT from AVL as a co-simulation middleware, an FMU containing a powertrain model generated with GT Suite from Gamma Technologies and CarMaker from IPG for vehicle dynamics simulation. Moreover, required additional features in the used tools and standards are discussed. The paper closes with a summary and an outlook.

2 Virtual ECUs

Virtual ECUs (vECU) aim at running target ECU code on standard x86 systems by virtualization. This section introduces use cases for virtual ECUs supporting the ECU developer in creating software with higher quality faster then with regular development processes. The basic software architecture for ECUs is explained and it is distinguished between three types of virtual ECUs that differ in the extent of the re-used target code. Finally the virtual ECU used in the feasibility study is presented.

2.1 The AUTOSAR software architecture

The AUTOSAR (Automotive Open System Architecture) standard defines an architecture (see figure 1) for embedded software on ECUs. The idea is to "‘cooperate on standards - compete on implementation"’. AUTOSAR systems can be divided in six main components (see 1):

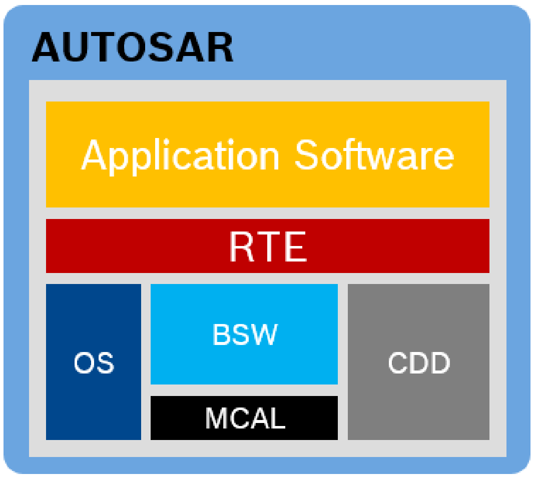

Figure 1. The AUTOSAR software architecture

1. **Application Software (ASW)** is the software implementing the unique features of an ECU, e.g., the behavior of the electronic stability program (ESP) or HAD functions.

2. **Runtime Environment (RTE)** is the communication layer which distributes the signals directly between ASW components or using the base software's (BSW) communication stack. The idea of AUTOSAR ASW components is that they can be distributed freely on different ECUs. The RTE will then either dispatch the data from one component directly to another component, if they are deployed on the same ECU, or the data is send via the communication stack in the base software.

3. **Base Software (BSW)** is software that provides basic functionality of an ECU. Typical software components are communication stacks such as CAN, automotive ethernet, or flexray. Other examples are memory access and diagnosis functions. The extent of the used base software on an ECU depends on the use-case, e.g., ECUs for wipers need less functionality than engine control units.

4. **Operating System (OS)** is a real-time system that is responsible for executing code at the right time and with a defined maximum duration. Therefore, runnables that contain the executable code, are scheduled using scheduling tables. The runnables are assigned to recurring tasks of a defined maximum duration. For simulation it is often desired to accelerate the execution of code. Therefore the tasks are executed as fast as possible. Furthermore, the OS is responsible for handling interrupts, e.g., when data is received from a sensor. This pauses the execution of runnables until the interrupt has been handled.

5. **Microcontroller Abstraction Layer (MCAL)** is the driver layer and specific for the used ECU hardware. This should be the single software component which is hardware dependent. For simulation on x86 systems the MCAL layer has to be exchanged.

6. **Complex Device Drivers (CDD)** contain special code which is not commonly used and this not part of the AUTOASR specification, e.g., drivers for magnetic valves.

2.2 Categories of virtual vECUs

vECUs can roughly be classified into three categories:

1. vECUs that contain only the ASW and RTE (and optionally an OS). This aims at quick testing of basic functions of the ASW without using base software components like communication. If the ASW code is AUTOSAR compatible, vECUs for this use case can be easily created since there are no hardware specific parts. However, no realistic estimation of the execution behavior on real ECUs can be derived with this kind of vECUs.

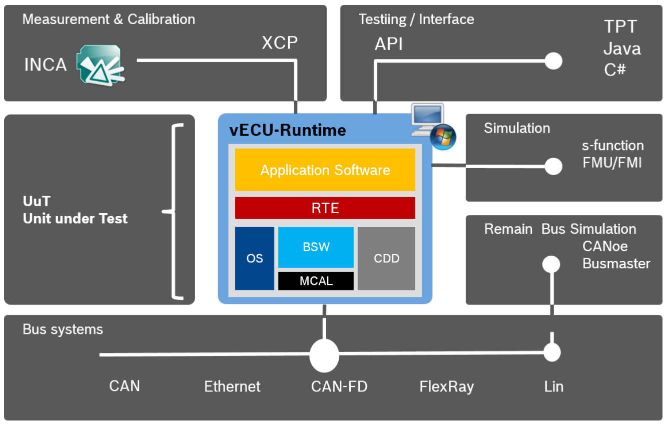

Figure 2. Use cases for vECUs

2. vECUs that consists of ASW, RTE, BSW, OS and a virtual MCAL (for x86 systems). A more realistic behavior of the real ECU can be simulated with this kind of vECU. The scheduling of tasks is considered and the functionality of the BSW can be tested. As an example rest-bus simulation can be performed to mock additional ECUs in the network to test for correct reaction of the simulated vECU.

3. If a more realistic behavior of the vECU is desired, virtual hardware can be used. All software components of the real ECU can be re-used, including the target MCAL layer. The MCAL is used with detailed hardware models which simulate real timing behavior. Another benefit is the ability to perform fault injection which can be problematic when done with real hardware since the injected faults could cause damage to the devices. A drawback of using hardware models is reduced simulation speed since the models are usually highly detailed.

2.3 Use cases for virtual ECUs

Virtual ECUs can be used for faster test of application software. Based on the vECU category used, BSW functionality and timing behavior can also be considered. Typical use-cases for vECUs are displayed in figure 2.

The XCP protocol is used by tools like ETAS INCA to measure and calibrate parameters of the ECU software, e.g. to optimize the gasoline injection for a certain engine type. This can also be done virtually with vECUs. Test APIs allow for automatic testing of the ECU software. Examples for commonly used testing tools are TPT and ECU-TEST. It is also possible to write custom tests with arbitrary programming languages like Java. The vECU can be exported as an FMU or S-Function for co-simulation with physical and plant models. Virtual busses (CAN, LIN, ...) can be connected to virtual ECUs using the MCAL layer. This allows to analyze messages on the busses and to perform rest-bus simulation with tools like CANoe or Busmaster to simulate additional ECUs in a network.

2.4 vECU for feasibility study

For the feasibility study a category 1 vECU has been used. The vECU consists of an OS, the RTE and application software. The application software has been generated as AUTOSAR 4 compatible code from an ASCET model. Based on the application's AUTOSAR description the RTE has been generated. No BSW or MCAL has been integrated at this point. This will be done as a next step in order to send messages via CAN. This will enable to investigate how a software function can be deployed on more than one ECU.

2.5 vECU tool evaluation

Several tools of different vendors including ETAS , QTronic, Dassault, dSPACE and Mentor Graphics have been evaluated for the creation of vECUs at Bosch. For this contribution ETAS EVE is used since it is best suited for the use case presented here (e.g. best integration into the existing ECU build tool chain).

3 Feasibility study

In this section it is shown how FMI is used to integrate and finally validate a software function in a co-simulation setup. The vision is to validate HAD functions or software for autonomous driving in the future. In this contribution, not a complete function consisting of e.g. object recognition, trajectory generation and follow-up control is used but only the lateral control since the goal is demonstrate the use of FMI to integrate ECU software into a simulation for virtual validation (and not to investigate the numerical properties).
In section 3.1 the co-simulation architecture is described, while 3.2 gives a brief overview on the used models and simulation results. In section 3.3 further requirements on the FMI standard and the used tools are derived.

3.1 Co-Simulation Architecture

In order to setup a co-simulation one of the first things to do is to define the integration platform, i.e. the tool that executes the master algorithm. In some cases, especially when not all involved tools offer an FMI export, it may be the case that there is not one defined master algorithm. Moreover, direct tool couplings (that do not rely on FMI) written by different tool vendors tend to have different numerical properties and to produce out-of-sync signals. An approach to face those issues is to a co-simulation middleware, that does not contain a model but only serves as a master and coordinator. Consequently, the co-simulation has a clean architecture with tool couplings that are consistent with each other(see figure 3). Moreover, it is easily extendable and typically offers more options to configure the co-simulation than simulation tools do. There exist several open-source (e.g. PyFMI) as well commercial (e.g. TISC from TLK Thermo, Cosimate from ChiasTek or Silver from QTronic that also includes vECU generation) co-simulation middlewares. In this contribu-

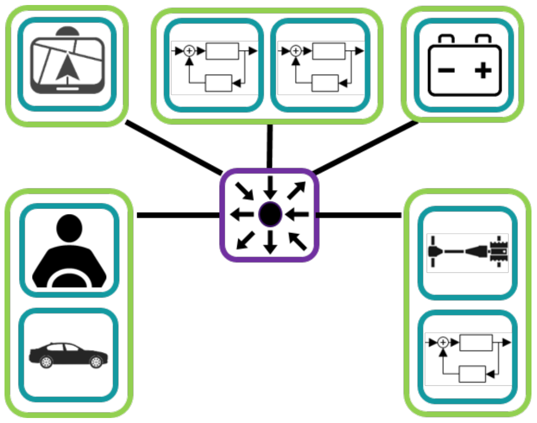

Figure 3. Co-simulation architecture with typical vehicle domains using a co-simulation middleware (purple rectangle)

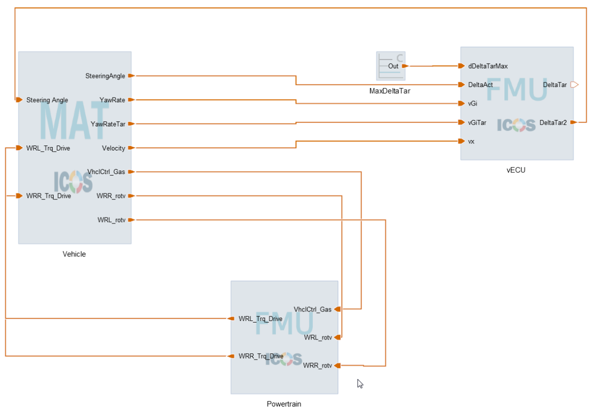

Figure 4. Setup of the co-simulation (screenshot from Model.CONNECT from AVL) with two FMUs (vECU from ETAS EVE and powertrain from GT Suite) and CarMaker

tion Model.CONNECT from AVL is used.

3.2 Co-Simulation Setup

As already described above the co-simulation consists of the following participants (see figure 4)

- CarMaker from IPG: Vehicle dynamics and driver model for longitudinal control
- FMU: Powertrain model generated from GT Suite from Gamma Technologies (see **??**)
- FMU: vECU generated with EVE from ETAS GmbH

Throughout this section it is assumed that the software under consideration can be validated by using the ISO double lane change as maneuver. The software shall be validated if the deviation in the yaw rate is rate is not bigger than 0.021/s and the deviation in the position in the y-direction is not bigger than 30cm. The vehicle model (and accordingly the software function) is parametrized according to a luxury car, however details are neglected here. Advanced co-simulation algorithms were not used, i.e. the models communicate using a parallel scheme using zero order hold extrapolation. It is expected that this has to be changed when using a more complex powertrain model and a more complex software function. Input for the software function is the actual yaw rate, the desired yaw rate, the vehicle velocity and the steering angle. Note that, the desired yaw rate is read from a table, that will be replaced by a trajectory generator in the future.
Figure 5 indicate the the simulation result lies within the specified error bounds. In fact the maximal error in the yaw rate is 0.0151/s and 29.8cm in the y-direction of the position of the vehicle. Thus (under the assumptions stated above), the software function can be judged as validated.

3.3 Derived Requirements for Tools and Interfaces

While it can be seen from the previous section that a virtual validation of ECU software using FMI is possible in principle, there are some issues that prevent or will (in the case of more complex functions) prevent FMI from being suited for that use case. Beside issues described in (Link et al., 2015) the following requirements were derived:

- For complex software functions lots of physical signals have to be connected to the vECU. Thus, FMI should support vectors for easier workflows and better models (w.r.t. clarity).
- When it comes to software functions that are distributed over multiple ECUs, signals have to be exchanged between them. In many cases these signals are not just scalars or vectors (see above), but structs. A typical example is the ADASIS protocol (Ress et al., 2008) that is used to transmit the e-horizon from an e-horizon provider to an e-horizon reconstructor. Thus, in order to use FMI for vECUs it should support structs.
- In cases where not only the functionality, but also the timing shall be validated the communication has also to be modelled, e.g. using virtual CAN. Currently, the user (FMU generator and/or integrator) has to care about the communication between FMUs (at least for virtual busses etc.). In future versions it would be desirable to have the communication mean as part of FMI. Note, that this will be the case for the Advanced Co-Simulation Interface (ACI) (Krammer et al.).

Beside the requirements on FMI there are also some issues on the tool side. Among these are

- According to the list above tools (simulation tools and co-simulation middlewares) should support vectors and structs.
- When more complex models are used and especially in cases where numerically demanding couplings are in place it is desirable to use a master algorithm that

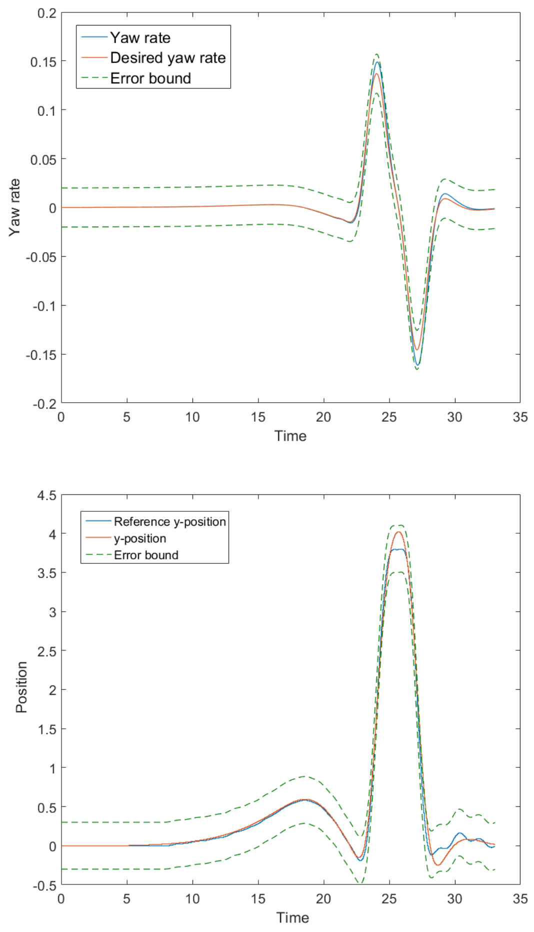

Figure 5. Variables used for validation including error bounds for the double lane change including error bound

can iterate, i.e. repeat macro steps. This is currently not supported by the used vECU, but will be supported in the future. However, many simulation tools do not support this (optional) FMI feature. This situation should be changed.

- Tool: If common open source implementations of virtual MCALs would reduce development overhead and enable switching between different vECU tools.

4 Summary & Outlook

This paper presents a feasibility study for the use of FMI for the validation of future ECU software. Besides a brief overview over the concept of vECUs different use cases for the use of vECUs are considered. For a functional validation (without timing) it is shown that the integration of ECU Code into a co-simulation works in principle. However, for more complex functions than the yaw rate controller used here, some issues arise that were collected in section 3.
In future work a more complex (HAD) function will be used. Consequently, more complex models have to be used. Moreover, it is desired to do also timing investigations. Thus, virtual CAN will be used for signal exchange. Therefore, the vECU has to include the communication stack. Last but not least an interface to connect calibration software (e.g. INCA from ETAS) will be created.

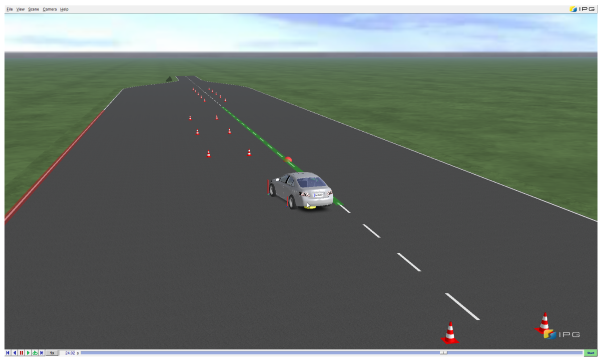

Figure 6. Screenshot showing the double lane change maneuver in CarMaker

References

Google self-driving car project monthly report. August 2015.

Henning Holzmann, Karl Michael Hahn, Jonathan Webb, and Oliver Mies. Simulation-based esc homologation for passenger cars. *ATZ worldwide*, 114(9):40–43, 2012.

Andreas Junghanns, Jakob Mauss, and Michael Seibt. Faster development of autosar compliant ecus through simulation. *ERTS-2014, Toulouse*, 2014.

Martin Krammer, Nadja Marko, and Martin Benedikt. Interfacing real-time systems for advanced co-simulation - the acosar approach.

Kilian Link, Leo Gall, Monika Mühlbauer, and Stephanie Gallardo-Yances. Experience with industrial in-house application of fmi. In *Proceedings of the 11th International Modelica Conference, Versailles, France, September 21-23, 2015*, number 118, pages 17–22. Linköping University Electronic Press, 2015.

Christian Ress, Dirk Balzer, Alexander Bracht, Sinisa Durekovic, and Jan Löwenau. Adasis protocol for advanced in-vehicle applications. In *15th World Congress on Intelligent Transport Systems*, page 7, 2008.

Walther Wachenfeld and Hermann Winner. Die freigabe des autonomen fahrens. In *Autonomes Fahren*, pages 439–464. Springer, 2015.

H. Winner, G. Wolf, and A. Weitzel. Freigabefalle des autonomen fahrens/the approval trap of autonomous driving. *VDI-Berichte*, (2106), 2010.