Fully Virtual Rapid ADAS Prototyping via a Joined Multi-domain Co-simulation Ecosystem

Róbert Lajos Bücs, Pramod Lakshman, Jan Henrik Weinstock,
Florian Walbroel, Rainer Leupers and Gerd Ascheid
Institute for Communication Technologies and Embedded Systems, RWTH Aachen University, Germany

Keywords: ADAS Prototyping, Driving Simulators, Virtual Platforms, Multi-domain Simulation, Near Real-time Execution.

Abstract: Advanced Driver Assistance Systems (ADAS) have evolved into comprehensive hardware/software applications with exploding complexity. Various simulation-driven techniques emerged to facilitate their development, e.g., model-based design and driving simulators. However, these approaches are mostly restricted to functional prototyping only. Virtual platform technology is a promising solution to overcome this limitation by accurately simulating the entire ADAS hardware/software stack, including functional and non-functional properties. However, all these tools and techniques are limited to their individual simulation environments. To reap their combined benefits, this paper proposes a joined frameworking to facilitate ADAS prototyping via full virtualization and whole-system simulation. For this purpose, an advanced automotive-flavor virtual platform was also designed, ensuring detailed, near real-time simulation. The benefits of the approach and the joined frameworks are shown by prototyping two ADAS applications in various system configurations.

1 INTRODUCTION

Vehicles have evolved from mainly mechanical to complex *hardware* (HW) and *software* (SW) driven systems in the last decades. Vehicular SW covers vast areas, ranging from infotainment, over safety-critical control tasks, heading all together towards highly automated driving. Furthermore, recent HW trends reflect strong integration, i.e., fewer *Electronic Control Units* (ECUs) with multi and many-core CPUs. The resulting architecture of *Advanced Driver Assistance Systems* (ADAS) consist of up to 100 networked ECUs with 250 embedded and graphic processors (Georgakos et al., 2013), executing an estimated 100 million lines of code (Charette, 2009).

Moreover, frequent factory recalls have been observed in the past years in case of various manufacturers, most of which were caused by failures in the electronic system (NHTSA, 2017). This called attention to secure HW/SW design, test and validation for which the automotive industry agreed on functional safety standards, e.g., the (ISO 26262, 2011). The standard defines strict rules to ensure functional safety throughout the complete lifecycle of the HW/SW system and recommends full-system simulation to provide early guarantees for this purpose. However, with the sheer complexity of the electronic and electrical system and the demanding requirements posed by functional safety standards, ADAS development and validation has become immensely difficult.

Model-Based Design (MBD): Utilizing MBD tools is advised to tackle these challenges, offering early algorithmic prototyping, in-tool testing and often ISO-certified code generation. MBD tools accelerate the *design-validation-refinement* cycle of ADAS via a simulation-driven approach. Yet, MBD tools are limited to *(i)* rigid artificial *inputs* and *outputs* (I/O) and *(ii)* model only algorithmic aspects of ADAS.

Driving Simulators: Such frameworks address the first limitation of MBD tools by providing realistic I/O via their virtual driving environment. This enables ADAS evaluation/refinement via virtual test drives, while experimenting with various traffic and environmental conditions and vehicle types (e.g., cars, trucks). Moreover, driving simulators ensure deterministic test repeatability, which is crucial if spurious errors occur. They can also include a driver in the ADAS evaluation, thus capturing real human reactions. Yet, driving simulators are restricted to model only functional aspects of ADAS. Moreover, they are limited to their simulation environment, making a joined usage with further tools mostly unfeasible.

Bücs, R., Lakshman, P., Weinstock, J., Walbroel, F., Leupers, R. and Ascheid, G.
Fully Virtual Rapid ADAS Prototyping via a Joined Multi-domain Co-simulation Ecosystem.
DOI: 10.5220/0006665900590069
In *Proceedings of the 4th International Conference on Vehicle Technology and Intelligent Transport Systems (VEHITS 2018)*, pages 59-69
ISBN: 978-989-758-293-6

Virtual Platforms (VPs): This technology addresses the second limitation of MBD tools by extending the simulation to the ADAS HW/SW system. VPs consist of simulation models of HW blocks and their interconnection, created via electronic system-level standards, e.g., *SystemC/TLM* (Accelera, 2012). These enable modeling and simulation of full systems including, e.g., CPUs, buses, memories and various peripheral devices. Moreover, VPs enable HW/SW co-design by simulating the whole ADAS SW layer as executed by the platform. Conversely to MBD tools, VPs enable exploring non-functional system properties, e.g., HW/SW partitioning, task mapping, schedulability and dynamic worst-case execution time analysis. VPs also facilitate system prototyping by providing full HW/SW visibility, debuggability and non-intrusive monitoring, while ensuring execution determinism. However, VPs are generally limited by either their simulation speeds or modeling accuracy. Moreover, similarly to driving simulators, VPs are limited beyond their simulation environment.

Multi-Domain Co-Simulation: This technique extends the boundaries of specialized simulation tools and models by providing means for cross-domain interconnection and joined control. The multi-domain approach can be used to fulfill the ISO 26262 requirement of full-system validation by co-simulating various vehicular subsystems. This would allow to connect the HW/SW simulation with the virtual environment of a driving simulator. Tool-agnostic standards are defined for such purposes, overcoming the inflexibility of point-to-point connections. However, target tools/models have to be made compliant to multi-domain co-simulation standards for joined utilization.

Objectives: To reap their combined benefits, this paper proposes joining the preceding tools and techniques to facilitate and accelerate ADAS prototyping via full virtualization and whole-system simulation. Putting this in practice, a joined frameworking is presented, composed by carefully chosen tools and standards, addressing the previous limitations, as follows:

 # 1. Overcoming the artificial I/O limitation of MBD tools by using the environment of driving simulators.
 # 2. Extending the functional simulation capability of MBD tools and driving simulators by ensuring beyond functional modeling and exploration via VPs.
 # 3. Providing compliance for all target tools to a selected multi-domain co-simulation standard.
 # 4. Addressing the speed/accuracy trade-off of VPs via an advanced automotive-flavor platform that ensures detailed and fast simulation at the same time.
 # 5. Pursuing near real-time whole-system co-simulation execution, to be able to involve the developer in the virtual ADAS test driving process.

The proposed frameworking allows ADAS testing in a closed-loop, i.e., *(i)* capturing the environment of a driving simulator via virtual sensors *(ii)* inputting the gathered data and executing the target ADAS on a VP and *(iii)* applying regulatory actions on the virtual vehicle within the driving simulator. Lastly, to highlight its advantages, two ADAS applications were prototyped using the proposed full-system simulator.

2 BACKGROUND

The previously presented ideas pose strict requirements on simulation ecosystems. Thus, various tools and standards were carefully compared to select the most suitable combination fulfilling the prerequisites.

In this work, Simulink (MathWorks, 2017) was chosen as MBD tool in the ADAS design automation flow, as it provides advanced modeling semantics, a vast block set and in-tool simulation features. Moreover, its certified code generator ensures safe and continuous ADAS integration onto target HW devices.

The requirements for a driving simulator in the proposed approach are availability, adaptability and a realistic virtual driving environment. After carefully examining numerous driving simulators, an open-source racing game, *Speed Dreams 2* (SD2, 2017), was selected as it supports various traffic and environmental conditions (e.g., precipitation, visibility), different car types and configurable vehicle dynamics (e.g., component dimensions). In this work, the tool was extended with urban traffic simulation support and ADAS virtual test driving in such environments. Lastly, compliance to a selected multi-domain co-simulation standard was also added, ensuring connectivity beyond its simulation environment.

The virtual platform technology is concerned with the strictest requirements in the proposed methodology and has been the main focus of this work. The envisioned framework needs to accurately model and simulate the complete ADAS HW/SW stack. On the HW side, this requires assembling a scalable distributed system, consisting of a configurable number of modular subsystems connected over a vehicular communication bus. Moreover, the envisioned platform needs to execute the complete ADAS SW stack, including the target algorithms, and ideally a full-fledged automotive *Operating System* (OS).

Due to its magnitude and complexity, the VP is expected to be the performance bottleneck of the whole frameworking. Thus, to avoid slowing down the full-system simulation, the platform needs to achieve execution speeds close to real-time. Considering these serious requirements, numerous SystemC-

based simulation technologies were carefully examined. From these, the GreenSoCs (GreenSoCs, 2017) framework emerged as the best choice. At its heart, GreenSoCs uses the QEMU (Bellard, 2005) fast emulation technology to overcome the strongest performance bottleneck of a VP, the CPU model. In this regard, QEMU can be also configured to contain multiple instances of the CPU module in a single package, still maintaining high execution efficiency. Its performance lies in the just-in-time compilation engine, translating and caching target CPU instructions to code blocks of the simulation host at run-time. However, due to the nature of system emulation, QEMU completely lacks timing annotation, which degrades the overall simulation accuracy. To address this issue, the GreenSoCs technology provides a dedicated SystemC wrapper around QEMU, thus integrating the CPU models into the timed SystemC environment. Herewith, GreenSoCs-based platforms can leverage both, fast simulation and temporal accuracy at CPU instruction level. The necessary timing management is achieved by synchronizing the global simulation time whenever QEMU is ahead of the current SystemC time by more than an amount called *quantum*.

Based on GreenSoCs, this work presents an advanced automotive-flavor VP that supports distributed multi-core setups while maintaining high execution efficiency, as detailed later. Lastly, to ensure connectivity beyond its environment, the VP has been made compliant to a multi-domain co-simulation standard.

In this regard, various standards were compared to join the previously presented distinct simulation ecosystems by co-simulating arbitrary vehicular subsystems beyond tool and domain boundaries. A centralized orchestration was envisioned for joined co-simulation control. After thorough examination, the *Functional Mock-up Interface* (FMI, 2017) was selected, a light-weight, open-source, tool-agnostic specification, considered the de facto multi-domain co-simulation standard for automotive applications. FMI defines a master/slave approach, where a central master is responsible for synchronization and data exchange between multiple slaves. Although the standard states the responsibilities of the master, it does not propose a reference implementation. In this work an in-house FMI master has been utilized for orchestration, as it achieves highly efficient co-simulation execution by implementing a parallel control engine.

On the other hand, FMI slaves, or *Functional Mock-up Units* (FMUs), encapsulate target tools/-models following the light-weight FMI C-API. This prescribes FMUs to implement functions for model creation/deletion (e.g., `fmi2Instantiate()`), run-control (e.g., `fmi2DoStep()`) and data exchange (e.g., `fmi2Get/Set()`), among others. The resulting FMUs are bundles containing *(i)* the model implementation (e.g., source code, pre-compiled binary) *(ii)* a description file defining its external interface *(iii)* other resources (e.g., third-party tools, documents).

Lastly, this work presents a generic method to ensure compliance of domain-specific simulation tools to FMI. The resulting co-simulation system allows coupling arbitrary FMUs to it, thus overcoming the inflexibility of direct, point-to-point connections.

3 JOINED MULTI-DOMAIN SIMULATION SYSTEM FOR RAPID ADAS PROTOTYPING

The selected simulation ecosystems are promising high-performance base technologies for the envisioned whole-system simulator. This section presents details to their adaptation to establish a fully virtual ADAS rapid prototyping environment. The resulting framework system is described step by step next.

3.1 Generic FMI-based Tool-coupling

First, an FMI-based, generic coupling technique was designed for the target simulators and models. The FMI standard defines two possible connection options for FMUs: *standalone* or *tool-coupling*. The former FMUs are self-contained, including the model, the simulator and all run-time dependencies. Conversely, tool-coupling FMUs are simple communication adapters, interacting with a simulator that contains the target model. Although standalone FMUs are more straight-forward to implement, research indicates severe problems with multiple instantiation and library co-dependencies (Bücs et al., 2015). Thus, in this work the tool-coupling technique was favored, as it separates the adapter and the target simulator into isolated processes, which is a more reasonable approach for the envisioned frameworking.

Depicted in Fig.1, first a Generic FMI Tool-Coupling Adapter was designed for this purpose. For simplicity, the adapter was constructed as a standalone FMU. Thus, the FMI master can load the module directly (without further dependencies) into its own host OS process. At instantiation, the adapter prepares an inter-process communication channel for interacting with a target simulation tool. So that the adapter remains independent from simulators and can be reused for arbitrary subsystems, a user-layer command protocol was defined (inspired by (GDB, 2017)), named *FMI Inter-process commu-*

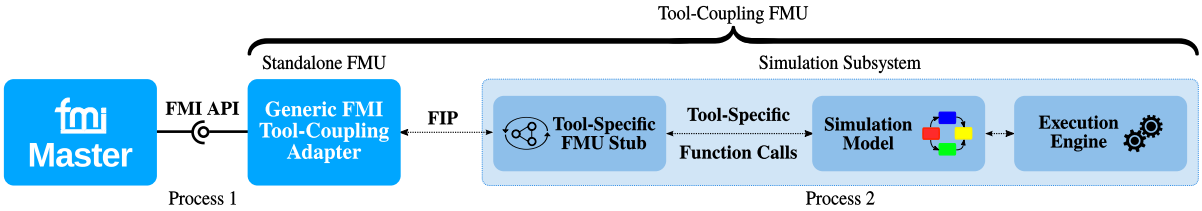

Figure 1: Generic FMI-based tool-coupling layer.

Start of frame	Command/ Acknowledgment	Comma Separated Arguments	End of frame
$	5 characters	0 to n	#

FMI Request	FIP Request Packets	Example
Initialize model	cinit	$cinit#
Request model step	cstep(StepSize)	$cstep1#
Get all integer model parameters	cgtin(n),vr[0],vr[1], ...vr[n-1]	$cgtin2,3,4#

FMI Response	FIP Response Packets	Example
Acknowledge initialization request	ainit	$ainit#
Acknowledge model step request	astep	$astep#
Return all integer model parameters	dgtin(n),vr[0],vr[1], ...vr[n-1]	$dgtin0,5,6#

Figure 2: Examples of FMI function to FIP frame mapping.

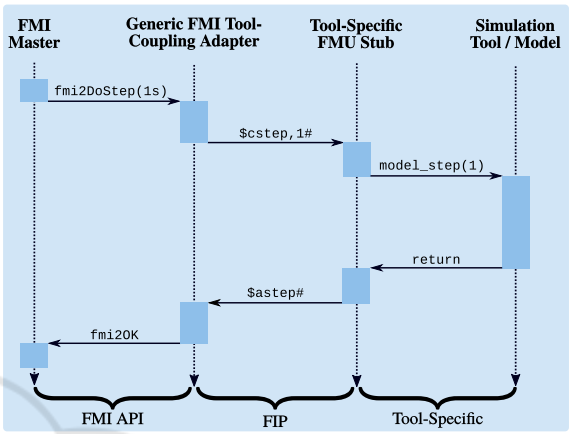

Figure 3: Sequence diagram of FIP-based communication.

nication Protocol (FIP). This implements a remote procedure call mechanism, where FMI API functions are translated to commands / acknowledgments of FIP, as exemplified in Fig.2. Herein, first the frame format can be observed, including the packet delimiters `$` and `#`. Below, several examples of FMI function to FIP frame mappings are demonstrated for various requests and responses. For instance, the `fmi2DoStep()` API function with $t_{step} = 1$ s translates to the FIP message `$cstep1#` and later to the acknowledgment `$astep#`, sent to the FMI master.

On the other side of the interaction, individual, Tool-Specific FMU Stubs need to be designed, that receive the FIP packages and execute the initial FMI request on the target model, as shown in Fig.1. To clarify the complete communication mechanism, Fig.3 depicts a sequence diagram. Following up on the previous example, the FMI master invokes again the `fmi2DoStep()` API function with $t_{step} = 1$ s. This call is translated to the FIP command `cstep` by the Generic FMI Tool-Coupling Adapter and sent to the Tool-Specific FMU Stub. The packet is then interpreted, and a tool-specific function call is invoked by the stub, in this meta-code example `model_step()`. After completing the model step, the stub sends the matching acknowledgment `astep` to the adapter, which passes the call status to the master.

The FIP-based adapter-stub mechanism addresses the **#3** objective of this work (Sec.1), providing a generic approach for target tools to comply to FMI. The corresponding tool extensions are presented next.

3.2 Adaptation of Speed Dreams 2

As mentioned previously, the original SD2 racing game was extended within the scope of this work to support urban traffic simulation and ADAS virtual test drives. First, city and country-side maps were created, and support for two-lane roads, intersections, two-way traffic and buildings was added, as depicted in Fig.4. Furthermore, the robots were adapted to follow traffic regulations, e.g., speed limits and yield the right of way, which is crucial in intersections.

Next, SD2 was converted into a tool-coupling FMU based on the mechanisms detailed in Sec.3.1. Shown in Fig.4, first a tool-specific Speed Dreams 2 FMU Stub was created to process FIP commands, sent by the FMI master over the adapter. The stub executes control operations via the Simulation Run-Control module, including stepping vehicle dynamics calculations. The stub can also access and expose parameters of the driving simulator, required for data exchange with other FMUs. Herein a Parameter Access module was built, granting safe data access for properties of Vehicle Dynamics and the Virtual Driving Environment, as indicated in the figure.

These adaptations address the **#1** and **#3** objectives of this work (Sec.1). First, the virtual environment of the driving simulator overcomes the limitation of artificial I/O. Moreover, the FMU compliance

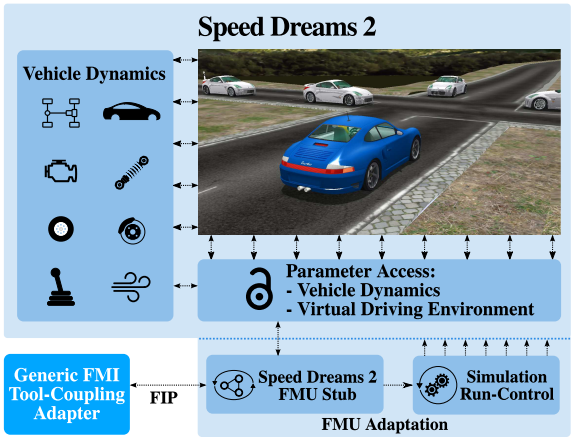

Figure 4: Block diagram of the modified SD2 simulator.

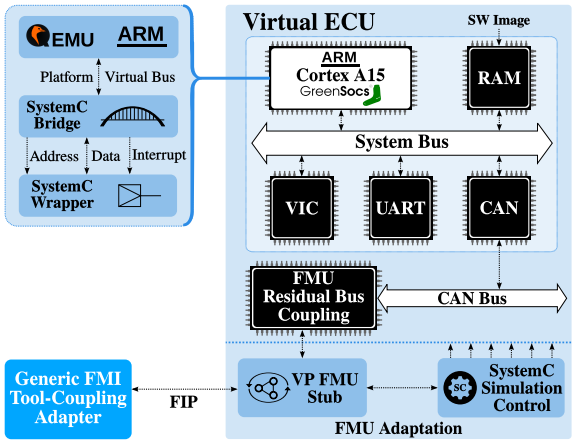

Figure 5: Block diagram excerpt of the proposed VP.

allows for external connections and control, required for multi-domain co-simulation. These traits can be exploited to achieve ADAS modeling beyond the functional level by utilizing a VP, detailed next.

3.3 Automotive Virtual Platform Design

Following the strict requirements discussed in Sec.2, an automotive-flavor VP was designed to extend the scope of ADAS testing. Shown in Fig.5, the VP consists of *virtual ECUs* (vECUs), modeling a modified ARM Versatile baseboard (ARM, 2017). At the heart of such vECUs a high-performance GreenSoCs-based ARM Cortex-A15 CPU model (ARM, 2012) is embedded. Depicted in Fig.5, this module consist of *(i)* the QEMU-based ARM processor simulator, *(ii)* a SystemC Bridge, capturing access commands of QEMU's Platform Virtual Bus, converting them to SystemC transactions/signals (e.g., address, data, interrupt) and *(iii)* a SystemC Wrapper, encapsulating the whole package as a SystemC module, providing all defined model interfaces (e.g., TLM sockets).

Furthermore, vECUs contain various in-house designed peripheral devices, e.g., an on-chip memory, a system bus, serial communication modules (UART) and a *Vectored Interrupt Controller* (VIC), among others. The particular set of HW peripherals and virtual devices was chosen so that vECUs are able to execute full-fledged OSs. Moreover, the platform architecture was designed to be scalable by instantiating a configurable number of interconnected vECUs, so to resemble a distributed multi-core system. Hence, each vECU was also equipped with a *Controller Area Network* (CAN) transceiver for inter-vECU communication over a single CAN-bus, as shown in Fig.5.

After its construction, the platform was converted into a tool-coupling FMU (recalling Sec.3.1). Similarly as for the driving simulator, first a VP FMU Stub was created to process FIP commands. Depicted in Fig.5, the stub also executes control operations via the SystemC Simulation Control component, which is responsible for creation/deletion of SystemC modules and run-state control. To expose the internal HW/SW state towards the co-simulation, an FMU Residual Bus Coupling unit was designed, acting as a bridge between SystemC and FMI. This module is capable of buffering/updating certain user-selected messages via their unique CAN identifiers as they are transmitted over the CAN bus. Each of these can correspond to specific HW/SW parameters (e.g., SW variables, HW signals). If the FMI master requires reading a parameter, a lookup is performed in the buffer for the corresponding CAN identifier, and the message is fetched, decoded and interpreted. Conversely, when the master sends data to the VP, the buffer embeds the parameter into a CAN message and injects it to the bus.

However, initial VP performance tests shown a notable simulation slowdown in multi-core setups, linearly with the number of added CPU models. As this issue contradicted the scalability and performance requirements, a more effective simulation technology was required alongside the GreenSoCs approach. Thus, the idea arose to exploit the inherent parallelism of the VP due to its modular structure. This resulted in executing each CPU model on a separate host thread and the remaining peripherals of the vECU on another, as they are less performance-critical. The resulting parallel segments run decoupled, ahead of the global simulation time, but are synchronized whenever their own local time reaches the quantum (recalling Sec.2). Platform parallelization achieves major performance gains, reaching execution speeds beyond real-time, as presented later.

The resulting performance-optimized VP addresses the **#2**-**#5** objectives of this work (Sec.1). Firstly, it extends the simulation spectrum to the ADAS

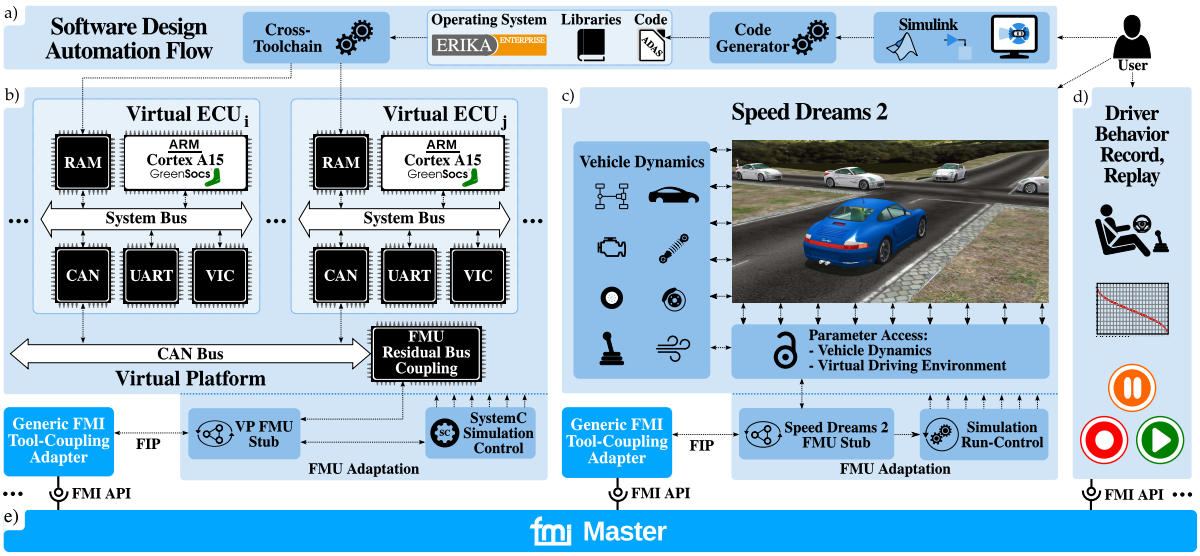

Figure 6: Joined framework system, co-simulation interconnection and design automation flow for rapid ADAS prototyping.

HW/SW system, covering analysis of non-functional properties (e.g., HW/SW partitioning). Moreover, the FMU adaptation allows for external connections and control, required for multi-domain co-simulation. Lastly, the presented performance improvements address the speed/accuracy trade-off and the pursuit for near real-time whole-system execution.

3.4 The Virtual Platform SW Stack

Next, the SW layers were designed for integrating ADAS prototypes as embedded code onto the created VP. At first, the integration can be performed in form of bare-metal SW. For this purpose, base SW components were created, consisting of low-level utilities (e.g., cross-toolchain, linker script, bootloader), a standard C library and a driver layer for the peripherals. The bare-metal setup enables early functional validation of ADAS properties, as well as HW/SW design space exploration, partitioning and distribution.

However, safety-critical ADAS applications have far stricter requirements, e.g., hard real-time execution guarantees and HW/SW protection mechanisms. To manage such applications, the automotive industry laid down *Real-Time Operating System* (RTOS) standards, such as OSEK/VDX (OSEK Group, 2005) and its successor AUTOSAR (AUTOSAR, 2014). Both impose portability of SW components, priority-based multi-tasking, standard communication layers, safe resource management and real-time scheduling, among others. Since AUTOSAR implementations were not available at the point this work was conducted, OSEK/VDX was chosen instead. Herein several variants were examined, and *ERIKA Enterprise* (EE) (Evidence, 2017) was selected among them, a comprehensive, open-source OSEK/VDX implementation with full tool support. EE is especially favored in open-source RTOS-based research activities, since it is developed towards the AUTOSAR specification.

In this work, EE was ported to the designed VP following the guidelines outlined in (Evidence, 2014). Herein various mechanisms were adapted, e.g., for fixed-priority scheduling, nested *interrupt* (IRQ) handling and inter-task communication beyond vECUs. However, since all details would go beyond the boundaries of this work, a full description of the porting instructions can be found in (Evidence, 2014).

3.5 The Full Design Automation Flow

After having all subsystems in place, the joined frameworking can be used as follows. Shown in Fig.6a, Simulink is the starting point for ADAS development and early functional testing. Once a prototype is mature enough, Simulink's built-in code generator can be invoked to obtain embedded code for it. The resulting ADAS C-modules can first be integrated onto the VP as bare-metal SW for early functional validation and HW/SW design space exploration. To support this, the VP can be configured to include as many vECUs as desired for the current development stage (Fig.6b). Moreover, SW integration and validation is supported by the possibility of run-time debugging on the VP, as the GreenSoCs technology allows attaching SW debuggers to the virtual CPU models. In more mature development stages, ADAS applications can be integrated as EE tasks onto the VP for refinement of non-functional properties, e.g., execution times and

Algorithm 1: LKA optimal steer angle control.

Input: w_{road}:road width [m], w_{lane}:lane width [m], $\sphericalangle_{veh\_abs}$:vehicle-road angle [rad], θ_{rel}:segment-relative yaw [rad], *lane*:ongoing/incoming
Output: $\sphericalangle_{steer}$: steer angle

```
// Update current and optimal vehicle position in lane
1:  d_left ← get_dist_from_left(...)
2:  d_right ← get_dist_from_right(...)
3:  d_opt ← w_lane/2

// Get target lane for the LKA to operate in
4:  if lane == ongoing then
5:      d_diff ← d_right - d_opt
6:  else
7:      d_diff ← d_opt - d_left

// Correct steer angle ∢_steer
8:  ∢_steer ← ∢_veh_abs + θ_rel − d_opt/w_road

// Normalize ∢_steer to −Π...Π
9:  while ∢_steer > Π do
10:     ∢_steer ← ∢_steer − 2∗Π
11: while ∢_steer < −Π do
12:     ∢_steer ← ∢_steer + 2∗Π
```

and scheduling. For all such design iterations, a full co-simulation can be set up via the FMI master, loading the tool-coupling FMUs of the VP and SD2 (Fig.6e). In this setup, the user can engage virtual test driving in SD2, while the target ADAS is executed on the VP, regulating the behavior of the virtual vehicle. In case of spuriously occurring errors, a Driver Behavior Record/Replay FMU was added to the co-simulation (Fig.6d), so to ensure exact test repeatability. The most notable feature of the presented system in terms of productivity is that it can achieve the complete ADAS exploration cycle (adjustment, code generation/integration, test drive) in minutes. To present its detailed capabilities, two driver assistance applications were prototyped with its help, as presented next.

4 RAPID ADAS PROTOTYPING

This section presents the refinement process of the created ADAS benchmarks. It must be noted that the focus here is not on the designed ADAS itself, but on the capability of the assembled frameworking to explore system properties on various abstraction levels.

4.1 Lane Keep Assistant (LKA)

This ADAS performs a closed-loop steering control of the vehicle, positioning it inside the driving lane. The LKA was designed for two operating modes, *(i)* automatically keeping the mid-point of the lane *(ii)* actively avoiding lane departure, while the driver is still mainly in control. Algorithm 1 presents a meta-code description of the former mode. Herein, first the road and lane widths are used to calculate the optimal and actual vehicle positions in the driving roadway (lines 1-3). Next, according to the selection of the incoming/ongoing lane, the distance error is calculated (lines 4-7). Subsequently, the steer angle is corrected using the current vehicle-road angle and the segment-relative yaw to compensate the distance error (line 8). The segment-relative yaw is defined as the rotational movement along the axis which is perpendicular to the chassis plane. This value is computed as the angular difference between the x-y plane of the car chassis coordinate system and the x-y plane of the road segment within the world coordinate system. Lastly, the updated steer angle is normalized to the range $\{\sphericalangle_{steer} \in \mathcal{R} \mid -\Pi \leq \sphericalangle_{steer} \leq \Pi\}$.

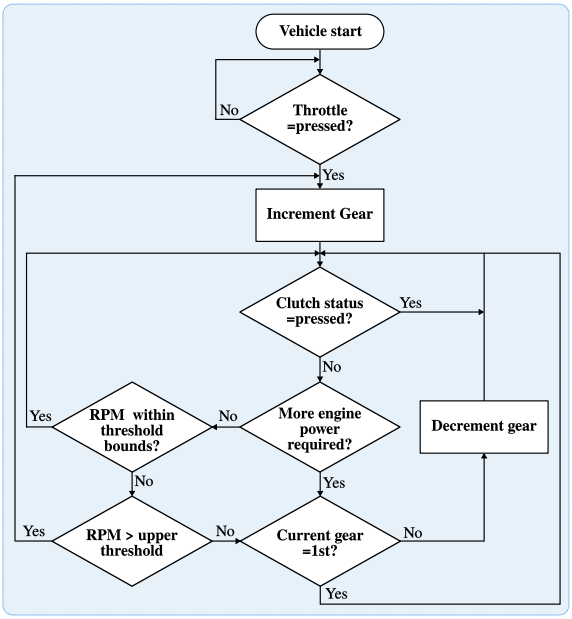

Figure 7: Flow chart of the automatic transmission control.

4.2 Automatic Transmission Control (ATC)

This algorithm was created as a closed-loop, finite state machine, periodically providing the gear control output after evaluating its inputs: the engine speed, the state of the virtual clutch release sensor and the user throttle control. The ATC supports various use-cases, e.g., regular up and downshift with respect to the current engine speed, vehicle start / stop and overtake assistance, shifting one gear down once it is recognized that more engine power is required. The operation of the automatic transmission control is best described by a flow chart, as depicted in Fig.7:

SC_A	Both ADAS integrated on 1x vECU as bare-metal SW.
SC_B	Both ADAS integrated on 1x vECU as EE tasks.
SC_C	Two ADAS separated on 2x vECUs as bare-metal SW.
SC_D	Two ADAS separated on 2x vECUs as EE tasks.

Figure 8: VP system setup used for benchmarking.

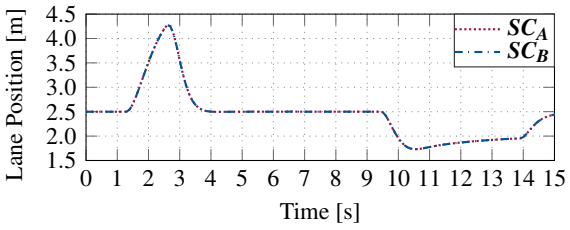

Figure 9: Stage transitions of HW/SW system design.

Figure 10: LKA lane position in various SCs.

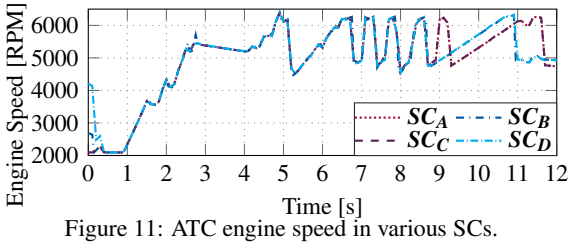

Figure 11: ATC engine speed in various SCs.

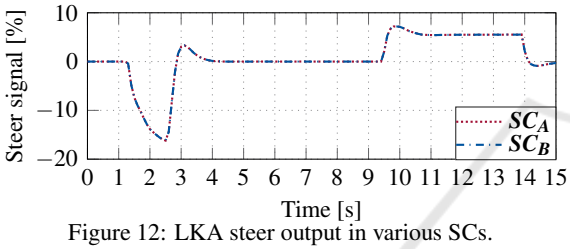

Figure 12: LKA steer output in various SCs.

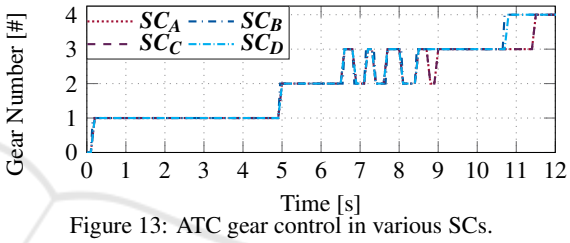

Figure 13: ATC gear control in various SCs.

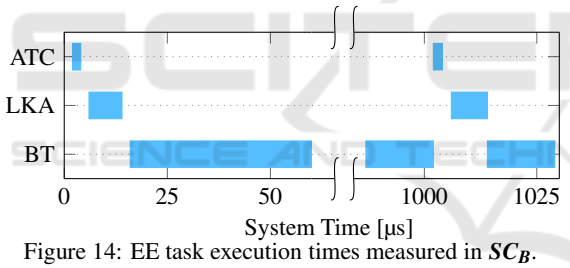

Figure 14: EE task execution times measured in SC_B.

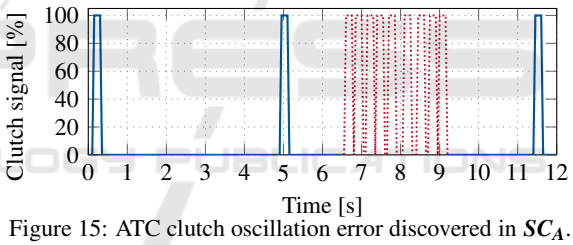

Figure 15: ATC clutch oscillation error discovered in SC_A.

1. From the initial vehicle start condition, the gear is incremented upon changes in the user throttle input.

2. Next, the clutch status is evaluated. If the vehicle is in the middle of a transmission, the algorithm enters a wait state until the clutch is released again, allowing for further possible gear changes.

3. Then, additional engine power request is checked. If the engine speed is lower but the user throttle control higher than certain predefined thresholds, the gear value is decremented so to provide more power, e.g., in a vehicle takeover scenario.

4. Lastly, the engine speed is evaluated with respect to predefined upper and lower thresholds. If the engine speed remains within these bounds, the algorithm enters a wait state, else an increment or decrement of the gear value is performed accordingly. Herein a final check is performed to avoid unrequested switching to natural gear.

4.3 HW/SW System Configurations

Both ADAS were integrated on the VP in the *System Configurations* (SCs) shown in Fig.8. The corresponding transitions in system design stages are detailed in Fig.9. This order was established to continuously explore and refine new HW/SW properties by lowering the design abstraction level as follows:

1. SC_A: algorithmic prototyping and integration
2. SC_B: timing analysis, RTOS support, inter-task comm.
3. SC_C: refining spatial distribution
4. SC_D: adding standard inter-vECU communication

The novelty of the frameworking is to enable such a design space exploration of non-functional traits (e.g., timing behavior, HW/SW partitioning), while stepwise refining the level of system abstraction.

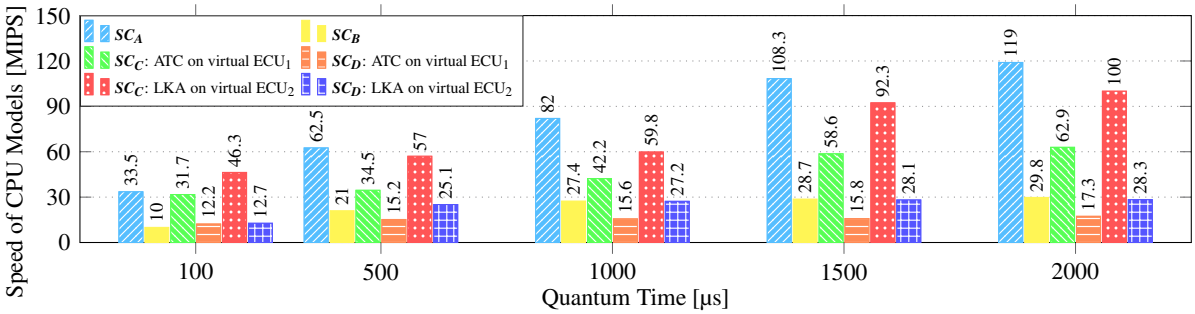

Figure 16: Execution speeds of the CPU model(s) within the VP, using various quantum settings, measured in various SCs.

4.4 Functional and Temporal Analysis

Numerous co-simulation runs[1] were executed in the setup shown in Fig.6 for ADAS test and refinement. All of these were orchestrated by the FMI master applying a fixed step size of 2 ms on every subsystem.

First, signals of the LKA were captured. Fig.10 shows the lane position of the test vehicle around the optimum mid-point of 2.5 m. The resulting steer control is shown in Fig.12, where negative values indicate right, positive values left steering. The reaction of the LKA in auto mode can be observed between 1.5-3.5 s in a sharp right, and between 9.5-15 s in a longer left curve. Only minor differences can be noted in both plots between $\boldsymbol{SC_A}$ vs. $\boldsymbol{SC_B}$ which can be explained by the short execution time of the ADAS. Thus, even as EE task, less system events (e.g., IRQs) disturb its execution. Data was also gathered in $\boldsymbol{SC_C}$ and $\boldsymbol{SC_D}$, but as only minor changes were noted (due to CAN messages transmit times), the results are not shown.

More variance was found refining the ATC. Fig.11 depicts the engine speed between 1-5 s while steadily accelerating, and between 5-12 s while alternating the throttle. The resulting gear control is shown in Fig.13. Functional and temporal deviations can be noted in both plots. As before, the minor timing differences between $\boldsymbol{SC_A}$ vs. $\boldsymbol{SC_C}$ and $\boldsymbol{SC_B}$ vs. $\boldsymbol{SC_D}$ are caused by CAN message delays induced by distribution. Yet, bare-metal vs. EE task implementations ($\boldsymbol{SC_{A,C}}$ vs. $\boldsymbol{SC_{B,D}}$) vary strongly, causing even extra gear shifts (around 9 s). The reason is that in $\boldsymbol{SC_{A,C}}$ gear control occurs more frequently, triggered by ad hoc activations via CAN message IRQs. Conversely, the refinements of $\boldsymbol{SC_{B,D}}$ grant a scheduler-based execution with activations at discrete time stamps. Although this occurs less often in the current setup, it provides real-time execution guarantees, that $\boldsymbol{SC_{A,C}}$ lack.

Moreover, Fig.15 shows the captured clutch signal, where an unexpected oscillation was observed between 6.5-9.2 s. Herein an algorithmic error was identified, violating a mandatory wait time between gear shifts, found and corrected by attaching a debugger to the vECU. This finding further affirms the advantages of the co-simulation system for ADAS prototyping.

Lastly, ADAS execution times were measured in $\boldsymbol{SC_B}$. Herein tasks were assigned fixed priorities from highest to lowest as follows: ATC, LKA and the idle *background task* (BT). As shown in Fig.14, EE schedules the tasks with the period of 1 ms, following their priorities. Moreover, the runtimes of both ADAS can be observed: around 2 µs for the ATC and 8 µs for the LKA. Herein, the ATC required less time to finish as it was in a wait state, checking only for the expiration of the mandatory stay in gear time. Although no timing violations were detected due to the relatively short application runtimes, the measurements demonstrated the potential of the proposed frameworking to provide dynamic execution time and schedulability analyses.

4.5 Co-simulation Performance Evaluation

After profiling the co-simulation system, the VP was found to be the main performance bottleneck. To analyze its impact, the number of executed *Million Instructions Per Second* (MIPS) was captured for all CPU models, shown in Fig.16. To quantify the speed of the full VP, the *Real-Time Factor* (RTF) was used, i.e., wall-clock time divided by simulated time, where RTF$\leq$1 corresponds to real-time. As the CPU models were clocked at f_{clk}=100 MHz, the goal was that they reach 100 MIPS (RTF=1). To achieve this, the quantum was also increased, with the trade-off of losing accuracy. Here it must be noted, that even higher quanta did not cause disturbances in OS scheduling. As shown in Fig.16, bare-metal vs. EE-based SCs achieve higher performance ($\boldsymbol{SC_A}$ vs. $\boldsymbol{SC_B}$ and $\boldsymbol{SC_C}$ vs. $\boldsymbol{SC_D}$), as the OS induces extra events (e.g., IRQs) causing simulation overhead. Fig.16 also indicates that multi-vECU setups are less efficient than single-vECU ones ($\boldsymbol{SC_C}$ vs. $\boldsymbol{SC_A}$ and $\boldsymbol{SC_D}$ vs. $\boldsymbol{SC_B}$), as the

[1]Simulation host: 6x AMD Phenom II 1100T x86_64, f_{clk}=3.3 GHz, 64K L1D and L1I, 512K L2 and 6144K L3 caches, 12 GB RAM, using Scientific Linux 6.8.

Table 1: Achieved co-simulation speeds in various SCs.

System Configuration	SC_A	SC_B	SC_C	SC_D
Average Co-Simulation Execution Speed [FPS]	31.7	14.49	19.06	7.63

simulation effort scales linearly with added vECUs but parallel efficiency does not. In SC_C and SC_D it can also be seen that the vECU executing the LKA is faster than the one running the ATC. This is explained by the relatively simple implementation of the LKA, leading to faster runs and more simulation idle time.

Lastly, the efficiency of the full system (Fig.6) was evaluated, with the quantum set to the overall co-simulation step size of 2 ms. The average *Frames Per Second* (FPS) rendered by the SD2 graphic engine was chosen as measure, as it reflects the real-time behavior of the full simulator. Table 1 holds the achieved results. As the VP dominates the whole system efficiency, the same trends can be noted as for the VP with regards to various system configurations.

Summary: Sec.4.4 highlights the potentials of the frameworking for system exploration and validation of functional and non-functional ADAS traits. The results in Sec.4.5 indicate that the co-simulation system can include the developer in virtual test driving. Herein certain setups (e.g., $SC_{A,C}$) reach near real-time whole-system execution, as frame rates over 20 FPS are considered adequate for human perception. This final achievement covers the **#5** objective of the paper, thus accomplishing all predefined goals.

5 RELATED WORK

Several approaches emerged for simulation-driven ADAS design by joining distinct tools and techniques.

The authors of (Schneider and Frimberger, 2014) present an FMI-based co-simulation framework, joining a driving environment with a purely functional ADAS simulator to develop a dynamic headlight adjustment unit. Herein the HW/SW simulation was abstracted away by static test benches, excluding closed-loop ADAS testing and non-functional evaluation.

To bridge this gap, authors of (Bücs et al., 2015) present techniques to couple SystemC-based HW/SW simulators with FMI. Although the work covers many aspects of HW exploration and ADAS design automation, the rudimentary vehicle models used for evaluation strongly limit test capabilities. Moreover, benchmarking implies RTF=22.4 for VP execution, making it impossible to include the developer in test driving.

Authors of (Wehner and Göhringer, 2013) prototype an adaptive cruise control via a Xilinx Zynq VP (Cadence, 2012) and a driving simulator. Herein the used VP is constrained by not allowing to create multiple platform instances, required for distributed setups. Authors also recognize that the used embedded Linux lacks real-time guarantees needed by safety-critical ADAS. As the VP achieves RTF=5.3, it heavily limits interactive ADAS test driving. Lastly, an ad hoc coupling is used between the subsystems, restraining the scalability of the simulation system.

Authors of (Raffaëlli et al., 2016) use a joined simulation framework to validate a lane departure warning and an emergency brake system. They couple a driving simulator over a test automation server with a fast QEMU-based VP, named Rabbits (Gligor et al., 2009). Since the VP is not parallel, a linear slowdown is expected with each added CPU model. Although performance numbers remain undisclosed, as the VP was used in an octa-core setup, it very likely excludes a human for ADAS test driving. Thus, although the setup is conceptually similar to the presented system, it can not reach the same efficiency.

The frameworking proposed in this paper combines the benefits of model-based design, driving simulators and VPs to accelerate ADAS prototyping. Herein FMI is used for tool coupling to overcome the inflexibility of ad hoc connections. The main novelty lies in the developed VP, used to explore non-functional traits. In contrast to previous works, the VP includes advanced technologies for accurate and fast simulation. Thus, the overall co-simulation system can reach near real-time speeds, allowing to include the developer in ADAS test driving. As to the best knowledge of the authors, these aspects have not yet been addressed by other scientific efforts.

6 CONCLUSIONS

This work presented a multi-domain co-simulation framework system to establish a fully virtual ADAS rapid prototyping environment. Putting this in practice, first the SD2 driving simulator was adapted for virtual test driving, urban traffic and multi-domain simulation. At the heart of the system, a high-speed VP was presented supporting distributed, multi-core vECU configurations and FMI coupling. To provide real-time guarantees for safety-critical ADAS, the ERIKA Enterprise RTOS was ported to the VP as part of a full SW design automation flow. Lastly, the capabilities of the joined tools were shown by prototyping an ATC and an LKA application in various system configurations. Regarding future work, the design of further ADAS is planned, heading towards highly automated driving.

REFERENCES

Accelera (2012). SystemC Language Reference Manual. *IEEE Std. 1666-2011 (Revision of IEEE Std. 1666-2005)*, pages 1–638.

ARM (2012). *ARM Cortex-A15 MPCore Technical Reference Manual*. Available at http://arm.com.

ARM (2017). Versatile board. Avail. at http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.dsi0034a/index.html.

AUTOSAR (2014). AUTomotive Open System ARchitecture Operating System Standard v4.1. Available at www.autosar.org/.

Bellard, F. (2005). QEMU, a fast and portable dynamic translator. In *USENIX Annual Technical Conference*, ATEC '05, pages 41–41.

Bücs, R. L., Murillo, L. G., Korotcenko, E., Dugge, G., Leupers, R., Ascheid, G., Ropers, A., Wedler, M., and Hoffmann, A. (2015). Virtual hardware-in-the-loop co-simulation for multi-domain automotive systems via the functional mock-up interface. In *Forum on Specification and Design Languages*, pages 1–8.

Cadence (2012). Virtual Platform for Xilinx Zynq-7000 EPP User Guide.

Charette, R. (2009). This car runs on code. IEEE Spectrum. spectrum.ieee.org/transportation/systems/this-car-runs-on-code.

Evidence (2014). Porting procedure for ERIKA Enterprise. http://erika.tuxfamily.org/wiki/index.php?title=Porting_ERIKA_Enterprise_and_RT-Druid_to_a_new_microcontroller.

Evidence (2017). ERIKA Enterprise Website. erika.tuxfamily.org/.

FMI (2017). Modelica Association - Functional Mock-up Interface (FMI) Standard Official Website. www.fmi-standard.org/.

GDB (2017). Remote Serial Protocol. https://sourceware.org/gdb/onlinedocs/gdb/Remote-Protocol.html.

Georgakos, G., Schlichtmann, U., Schneider, R., and Chakraborty, S. (2013). Reliability challenges for electric vehicles: From devices to architecture and systems software. In *Design Autom. Conf. (DAC)*, pages 1–9.

Gligor, M., Fournel, N., and Pétrot, F. (2009). Using binary translation in event driven simulation for fast and flexible MPSoC simulation. In *IEEE/ACM International Conference on HW/SW Codesign and System Synthesis (CODES+ISSS)*, pages 71–80.

GreenSoCs (2017). Virtual Platforms. www.greensocs.com/.

ISO 26262 (2011). *Road vehicles - Functional safety*. ISO, Geneva, CH.

MathWorks (2017). Simulink. http://mathworks.com/help/simulink/.

NHTSA (2017). National Highway Traffic Safety Administration, U.S. Dept. of Transportation - SafeCar Website. www.safercar.gov.

OSEK Group (2005). OSEK/VDX Operating System Standard v2.2.3. Available at www.osek-vdx.org/.

Raffaëlli, L., Vallée, F., Fayolle, G., Souza, P. D., Rouah, X., Pfeiffer, M., Géronimi, S., Pétrot, F., and Ahiad, S. (2016). Facing ADAS validation complexity with usage oriented testing. *Computing Research Repository (CoRR)*, arXiv:1607.07849.

Schneider, S. and Frimberger, J. (2014). Significant reduction of validation efforts for dynamic light functions with FMI for multi-domain integration and test platforms. In *International Modelica Conf. 2014*.

SD2 (2017). Speed Dreams 2 Website. www.speed-dreams.org/.

Wehner, P. and Göhringer, D. (2013). Evaluation of driver assistance systems with a car simulator a virtual and a real FPGA platform. In *Design and Architectures for Signal and Image Processing*, pages 345–346.