Abstract

The engineering processes for the design of Cyber-Physical Systems (CPS) are often suboptimal due to - among others - inefficient collaboration between the involved engineering domains. This may lead to increased development time and cost, oversized hardware, suboptimal product performance, etc. One of the factors contributing to this inefficient collaboration is that the integration of product components in the final stages of the development process was not sufficiently specified at design time. In this paper, we propose the Embedded Platform Simulator, EPSim, enabling the virtual integration of the control and the embedded domain in early stages of the development process. We elaborate on the architecture of the EPSim-tool and validate its precision with traditional control-embedded Hardware-in-the-Loop simulations.

Keywords: Control-Embedded Co-design, Cyber-Physical Systems, Virtual Integration, Co-simulation.

1 INTRODUCTION

There is an increasing demand for safer, more performant, feature-rich, and intelligent systems in various areas such as manufacturing, machine construction, automotive, military & defense, retail, and consumer electronics. Simultaneously, development engineers have to cope with the growing complexity of systems due to new concepts such as over-the-air updates, predictive maintenance, and product variability. This is addressed by the increased use of embedded platforms consisting of hardware (the microcontroller) and middleware (that includes an operating system), assuring the real-time execution of the application software.

In particular, for the design of Cyber-Physical Systems (CPS), multiple engineering domains make use of these embedded platforms. The related engineering processes are often suboptimal due to - amongst others - inefficient collaboration. This typically leads to increased development time and costs, oversized hardware, suboptimal product performance, etc. One of the factors contributing to this inefficient collaboration is that the integration of multiple software components in the final stages of the development process was not sufficiently specified earlier at design time.
This specifically applies when designing the control logic for a CPS. Here, control engineers focus on the design of the application software, while the embedded engineers are responsible for the selection and configuration of the embedded platform. The integration testing of both engineering domains often appears quite late in the development cycle. This is due to the fact that this kind of testing requires that the application software (i.e., the control software) is deployed on an embedded platform. The deployment activity involves multiple steps: the generation of application code from a model (in case the software is designed following a Model-Driven Engineering (MDE) approach), the configuration of the middleware including the (real-time) operating system and its tasks, the mapping of the application software on those tasks, and the configuration of communication channels. In development processes, this deployment step is referred to as the building phase and precedes the (integration) testing phase.

One approach to solve the limitations of late integration (and integration testing) is to facilitate virtual integration (and virtual integration testing) of multiple domains through simulation in earlier stages of the development process. Therefore, we propose to introduce a virtual integration test phase preceding the traditional build and test phases of MBSE processes. This is visualized in Figure 1, which shows the example case of the inclusion of the Virtual Integration Testing in the V-model for software development. It is clear that this approach allows to front-load different kinds of (real-time) embedded analyses, such as resource usage and time delay analysis. This way, integration errors related to the deployment will be detected earlier, which saves costly re-engineering in later phases.

To endorse this goal, we developed EPSim, an Embedded Platform Simulator allowing engineers to model the embedded platform using a library of configurable blocks. Both hardware (e.g., converters) and middleware (e.g., operating system) related components are part of the library and can be selected to compose the embedded platform. The underlying Discrete EVent System specification (DEVS) formalism provides the semantics of our simulator and allows the embedded platform model to be co-simulated with (a model of) the application software to be deployed. Our co-simulation approach addresses one of the problems in systems engineering: the simulation of application software under the assumption of unlimited computational resources for the CPS. In the literature, this is often referred to as Zero Execution Time (ZET) (Derler et al. 2013): in a single simulation time step, it is assumed that (a) inputs are sampled, (b) outputs are computed, and (c) instantaneously written. However, when deployed on an embedded platform those steps do take some computation time. Therefore, the simulation of application software is better approximated by the Logical Execution Time (LET) principle (Derler et al. 2013), for which (a) inputs and (c) outputs are sampled or set at a guaranteed interval. This implies that (b) the computation time of application is taken into account during simulation. EPSim facilitates the LET principle by enabling the virtual deployment of the application software on a (simulated) embedded platform. It can therefore also be called virtual Hardware-in-the-Loop (vHiL) testing.
Vanherpen, De Meulenaere, Vanommeslaeghe, and Maes

The paper is structured as follows. In Section 2, related work regarding the simulation of deployment effects is discussed. Section 3 elaborates on the architecture of our embedded platform simulator. In Section 4, we presented an adaptive cruise control case study, which we use to validate the performance of EPSim in Section 5. We summarize and discuss this paper in Section 6 and present some use cases for which this research can be valuable.

2 RELATED WORK

Different approaches have been proposed to obtain LET behavior in simulations to allow engineers to evaluate the temporal behavior of their application after (simulated) embedded deployment and its impact on the functional behavior of the application. This is especially important in the development of Cyber-Physical Systems, where correct, real-time operation of the system is often critical.

One possible approach is to annotate the application software with information retrieved from the platform. Ciccozzi et al. describe in (Ciccozzi et al. 2013) a solution that back annotates extra-functional properties to the model of the application software. However, their back annotation consists of a textual description that needs to be interpreted by engineers. In (Vanherpen et al. 2015), the authors present a (semi-)automated approach that enables the back annotation of extra-functional properties by introducing platform-related blocks in the model of the application software. However, these techniques only provide a static approximation of the temporal behavior of the application on the embedded platform. As such, they do not allow engineers to discover, e.g., scheduling-related issues and their impact on the application under development.

Another method to obtain LET simulation behavior is by co-simulating a model of the embedded platform with a model of the application software. El-khoury and Törngren were one of the first to present such an approach in which platform-related blocks were introduced in the model of the application software (El-khoury and Törngren 2001). A similar approach is observed with TrueTime (Henriksson et al. 2003, Cervin et al. 2003), a MATLAB Simulink toolbox for modeling parts of the embedded platform. However, TrueTime requires the (control) application to be implemented as MATLAB scripts or C++ code, complicating its integration with existing models. The T-Res framework presented in (Morelli and Di Natale 2014, Cremona et al. 2015) extends the usability of the TrueTime approach by using triggered subsystems, facilitating the integration with Simulink models. In that sense, it is similar to the MATLAB toolbox SimEvents, which can also be used to model and simulate the embedded platform, as demonstrated in (Li et al. 2016, Mertens et al. 2019). However, while these approaches integrate well within the MATLAB Simulink, integration with models from other tools (e.g., from different engineering domains) is not straightforward.

Finally, we highlight commercially available tools that share some similarities to what is presented in this paper and in previous work (Vanommeslaeghe et al. 2020). Both dSPACE and Vector provide solutions to simulate the embedded platform with their tools SystemDesk (dSPACE) and vVIRTUALtarget (Vector), respectively. However, the analysis of how the embedded platform configuration affects the execution of the application software is not considered. A tool that provides an environment to couple tools from different domains, mainly focusing on requirements and architectural analysis but also claiming virtual HiL testing, is FERAL (Kuhr et al. 2013). The tool focuses on modeling and simulating x86 and ARM architectures and provides insights into how task response times affect the behavior of the application software.

3 EMBEDDED PLATFORM SIMULATOR

This section gives an overview of our Embedded Platform Simulator (EPSim) and its major constituent parts. Figure 2a shows a high-level view of the EPSim: a model of the embedded platform is co-simulated with (a model of) the application software, which typically controls the physics of the CPS. The input signals are,
for example, the sensor signals, and the output signals represent the application software’s simulation traces with the embedded platform effects (vHiL simulation).

When going into more detail, the Embedded Platform Simulator is constructed of five components, as depicted in Figure 2b: (i) the Library, (ii) the Co-Simulation Interface, (iii) the Simulation and Analysis module, (iv) the Application Programming Interface (API), and (v) the User Interface. In the following subsections, we elaborate on each component’s functionality and how they relate to each other.

![Figure 2: Embedded Platform Simulator (EPSim).](image)

3.1 Library

To facilitate the modeling of the embedded platform, EPSim’s library comprises models of hardware and middleware components at different levels of abstraction. This allows engineers to model those aspects of the embedded platform that affect the execution of the application software. We categorize them under (real-time) operating system, communication, and peripherals related models. For the latter, EPSim currently provides blocks to model Analog to Digital Converters (ADC), Digital to Analog Converters (DAC), and Pulse Width Modulation (PWM). The (real-time) operating system sub-library contains various task models (fixed-priority, periodic, event-based, etc.), an interrupt model, two processor models (single-core and multi-core), and a fixed-priority preemptive scheduler model. As for the communication interfaces, the library provides models of a Controller Area Network (CAN), a Serial Peripheral Interface (SPI), and a Universal Synchronous Asynchronous Receiver Transmitter (USART) communication interface. For each model, one can opt for the most appropriate level of abstraction, depending on what aspects of the embedded platform require more or less detailed analysis. For example, suppose the deployed application software is sensitive to communication delays. In that case, selecting a detailed CAN communication model that incorporates the send and receive buffers of a CAN node, message arbitration, and bit stuffing makes sense.

An embedded platform is considered a discrete-time system as resources are driven by a hardware clock. As such, each component can be represented as a set of states processed in a pre-defined order. As an example, Figure 3 shows the states of a software task running on an embedded processor. When booting an embedded system, tasks are Disabled. Once enabled by the Real-Time Operating System (RTOS), the task is Suspended and waits to be resumed by the RTOS. When resumed, the task transitions to the Ready state, meaning it is able to execute once a Running task with an equal or higher priority is Completed. If a lower priority task is currently running, it is Preempted. Before being executed, several register and memory addresses are loaded in the Loading state. To model such state models of the embedded platform, we have
opted to use DEVS (Zeigler 1976, Zeigler et al. 2019), a well-suited formalism providing the necessary syntax and semantics to define discrete behavior.

3.2 Co-Simulation Interface

To facilitate the co-simulation between the embedded platform and the application software, EPSim integrates support for the Functional Mock-up Interface (FMI) standard (Blochwitz et al. 2011, MODELISAR Consortium and Modelica Association). This standard defines a standardized exchange of models between different tools using Functional Mock-up Units (FMU). Two different types of FMU exist: Model Exchange (ME) and Co-Simulation (CS). The difference between the two is the solver used to execute the FMU (i.e., the application). For ME FMUs, an external numerical solver is needed to execute the application, while in the case of CS FMUs, also called standalone FMUs, the required numerical solver is part of the FMU.

EPSim supports both types of FMUs so that the application software, developed using third-party tools such as MATLAB® Simulink®, can be co-simulated by means of Functional Mock-up Units (FMU). The embedded platform model serves as the master of the co-simulation algorithm and commands the application software to step forward once the task on which the application is deployed has finished its execution.

3.3 Simulation and Analysis Module

The heart and soul of EPSim is its Simulation and Analysis module. An embedded platform is characterized by hardware and middleware components that, to some extent, influence the behavior of the application deployed thereon. For example, a lower processor speed will lead to longer execution times of the application software and hence to longer sensor-to-actuator delays. By explicitly modeling the embedded platform (see Section 3.1), engineers can execute different types of simulations and analyses to assess how the control and embedded domain influence each other. EPSim currently supports four types of simulations and analyses:

- **Deployed application behavior:** Co-simulating the embedded platform with the application software facilitates virtual deployment of the application, providing engineers the means to evaluate the behavior of the application software as if it were deployed on the actual platform. According to the LET principle, EPSim delays the simulation of the application by taking into account the response times of the tasks that execute the different parts of the application. Users can indicate which outputs are of interest, of which EPSim collects behavioral traces. They can be evaluated in tools like MATLAB® Simulink®, which not only simplifies comparison but also ensures that control engineers are able to analyze the results in an environment they are familiar with.

- **Schedulability analysis:** EPSim allows embedded engineers to model the embedded platform, including the RTOS and its tasks. Engineers can configure task properties like priority, execution time, period, and deadline. The configuration of the different RTOS tasks will affect the schedulability of
a system, i.e., whether all tasks can be completed before their deadline. EPSim provides embedded engineers feedback whether task(s) fail to meet their deadline and how often this occurs.

- **Time analysis:** Following the previous, EPSim provides engineers with detailed timing insights. For each simulated task, the best-case, worst-case, and average response time is summarized. Similar timing information can be collected and summarized to the engineer for messages on a communication bus (e.g., CAN).

- **Performance analysis:** As we explicitly model certain hardware aspects, EPSim can provide insights into how these components perform during execution. Depending on the chosen abstraction level of the model, EPSim is able to estimate the load of a processor or core, the communication bus load, memory usage, etc.

### 3.4 Application Programming Interface

To model the embedded platform and to initiate a (co-)simulation and analysis, EPSim provides a Python-based API to the user. The different library components, modeled in DEVS, are abstracted so that only the configurable parameters are accessible. For the example of the task (Section 3.1), the API provides the following function for a periodic fixed priority task:

```python
task_Deb = FixedPrioTask(name='Debounce', execution_time=12.157e-6, deadline=13e-6, priority=10, offset=0.1e-6, executes_fmu=Control)
```

Once the system is defined using the API, EPSim transforms the given system configuration to a coupled DEVS model, which connects the atomic DEVS models of tasks, processor(s), scheduler, etc. (Vangheluwe 2001).

### 3.5 User interface

A Python-based API as described in the previous section feels familiar to embedded engineers who have experience with (different) programming languages. However, for engineers with less experience in lower-level languages, a Domain-Specific Language (DSL) is developed to evaluate the deployed behavior of their developed application (control) software. We have opted to define the DSL using JetBrains Meta Programming System (MPS) (JetBrains). It uses a projectional editor that interacts with an Abstract Syntax Tree (AST) (Voelter 2013) describing the meta-model of a system, guiding the engineers in configuring the virtual system. Similar to the Python-based API implementation, the model of the system is transformed by EPSim to a coupled DEVS model in addition to the configuration of the co-simulation algorithm.

### 4 CASE STUDY

In order to validate to what extent EPSim assists engineers in designing a CPS, we have applied EPSim in a system evolution design process for the development of a road sign recognition algorithm, extending an existing Adaptive Cruise Control (ACC) application. We opt for a system evolution design process to also demonstrate the transformation process of legacy software.

The existing system consists of two Lego® cars, as outlined in Figure 4. The first car, referred to as the lead car, follows a line with a preset speed profile with repeated emergency stops. The line following algorithm uses a color sensor to detect the line. The second car, referred to as the ACC car, follows the same line using a similar color sensor. An ACC algorithm aims to drive as close as possible to the lead car without colliding. The distance is measured by an additional ultrasonic sensor. Both cars are equipped with a
Vanherpen, De Meulenaere, Vanommeslaeghe, and Maes

Raspberry Pi 3 B+ board on which the algorithms are deployed on a single core. A Dexter daughterboard allows communication with the Lego® sensors and actuators. The Raspberry Pi communicates with this Dexter daughter board through SPI. During our experiment, the ACC algorithm, deployed on the second (ACC) car, is stressed by emergency stops of the lead car. Incorrect configuration of the embedded platform, and in particular its RTOS (e.g., task priorities), might lead to a collision.

![Figure 4: Overview of the Lego® case study.](image)

The evolved system extends the ACC algorithm on the second car with a road sign recognition feature. From a hardware design perspective, the ACC-car will be equipped with a second color sensor at the side of the car, allowing it to read barcodes printed on the floor. These barcodes represent road signs such as speed limitations and stop signs. From a software design perspective, an additional algorithm needs to be developed and deployed on the existing embedded platform.

The next section details the steps in the development process to reach the evolved system, related to EPSim.

5 VALIDATION OF AN EPSIM SUPPORTED DESIGN PROCESS

In the following subsections, we detail different steps in the development process required to develop the evolved system introduced in the previous section. Here, we focus specifically on the steps related to EPSim. In short, we start by characterizing the existing system by measuring the execution time and response time of the existing software components. This information is later used to correctly model the system using EPSim. Next, we design the additional software component for the evolved system, in this case in MATLAB® Simulink®. After this, we use EPSim to model the evolved system and to perform initial analyses, e.g., schedulability of the evolved system, before its deployment. To characterize the additional software component, we deploy it on the target platform and measure its execution time. This information is then added to the EPSim model. In the final step of our approach, we use EPSim to virtually characterize the evolved system, providing the expected execution times and response times for each software component. Additionally, we use EPSim to analyze the expected behaviour of the evolved system in simulation. In the current paper, we also validate this simulated behavior using a traditional Hardware-in-the-Loop test.

5.1 Characterization of the Existing System

To correctly estimate the impact of deploying the additional feature, the current system needs to be analyzed first. In the current case, a fixed-priority preemptive scheduler is used to schedule the different software components, each mapped on a single task of the RTOS. Each task is assigned a priority (a lower number indicates a higher priority), a period, and a (possible) offset. We focus here on timing-related aspects of the system, such as the system’s (worst-case) execution time (WCET) and (worst-case) response time (WCRT). To characterize these parameters, logged data or coverage techniques such as Modified Condition/Decision
Vanherpen, De Meulenaere, Vanommeslaeghe, and Maes

Coverage (MC/DC) are used to generate different input data so that different paths in the software are executed. Table 1 summarizes the analysis of the current system with its different software components.

<table>
<thead>
<tr>
<th>ID</th>
<th>SW component</th>
<th>Priority</th>
<th>Period (ms)</th>
<th>Offset (ms)</th>
<th>WCET (ms)</th>
<th>WCRT (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Read position of left encoder</td>
<td>5</td>
<td>10</td>
<td>0</td>
<td>0.206</td>
<td>0.206</td>
</tr>
<tr>
<td>2</td>
<td>Read position of right encoder</td>
<td>6</td>
<td>10</td>
<td>0</td>
<td>0.212</td>
<td>0.419</td>
</tr>
<tr>
<td>3</td>
<td>Read intensity from color sensor</td>
<td>7</td>
<td>10</td>
<td>0</td>
<td>0.177</td>
<td>0.597</td>
</tr>
<tr>
<td>4</td>
<td>Read speed from left encoder</td>
<td>8</td>
<td>10</td>
<td>0</td>
<td>0.287</td>
<td>0.885</td>
</tr>
<tr>
<td>5</td>
<td>Read speed from right encoder</td>
<td>9</td>
<td>10</td>
<td>0</td>
<td>0.287</td>
<td>1.173</td>
</tr>
<tr>
<td>6</td>
<td>Read battery voltage</td>
<td>10</td>
<td>10</td>
<td>0</td>
<td>0.180</td>
<td>1.354</td>
</tr>
<tr>
<td>7</td>
<td>Read distance from ultrasonic device</td>
<td>11</td>
<td>10</td>
<td>0</td>
<td>0.217</td>
<td>1.572</td>
</tr>
<tr>
<td>8</td>
<td>Steer power left motor</td>
<td>15</td>
<td>10</td>
<td>3</td>
<td>0.118</td>
<td>0.234</td>
</tr>
<tr>
<td>9</td>
<td>Steer power right motor</td>
<td>15</td>
<td>10</td>
<td>3</td>
<td>0.115</td>
<td>0.115</td>
</tr>
<tr>
<td>10</td>
<td>Execute “FromDS2DS” component</td>
<td>12</td>
<td>10</td>
<td>0</td>
<td>0.022</td>
<td>0.022</td>
</tr>
<tr>
<td>11</td>
<td>Execute “ACC controller” component</td>
<td>13</td>
<td>10</td>
<td>2.5</td>
<td>0.0045</td>
<td>0.068</td>
</tr>
<tr>
<td>12</td>
<td>Execute “line follower” component</td>
<td>14</td>
<td>10</td>
<td>2.5</td>
<td>0.0044</td>
<td>0.113</td>
</tr>
<tr>
<td>13</td>
<td>Execute “battery monitor” component</td>
<td>50</td>
<td>10</td>
<td>7.5</td>
<td>0.0048</td>
<td>0.048</td>
</tr>
</tbody>
</table>

5.2 Design of the Additional Component

The second part of our EPSim-driven method focuses on the design of the additional road sign recognition feature. The existing system was designed using a model-based approach. For the different software components in Table 1, a model in MATLAB® Simulink® exists. Also, a plant and environment model exists that simulates the acceleration, speed, and position of the cars. We opted to design the additional component in the same modeling environment, facilitating the evolution of the existing system. The additional component includes an algorithm based on the input of a color sensor, moving over the barcode representing the road sign. The component must be executed at a variable period depending on the speed of the vehicle, ensuring that the road sign recognition algorithm is called at least four times while reading a single barcode.

5.3 Sensitivity Analysis

In this phase of the method, the used platform (i.e., the Raspberry Pi of the ACC-car with its RTOS) is modeled in EPSim by means of its API or user interface as described in Section 3.4 and 3.5, respectively. Analyzed properties of the existing system (Table 1) are used to correctly model the existing system, while a domain expert estimates the properties of the newly created component(s). This allows engineers to evaluate its impact on the evolved system. For example, engineers can evaluate how the priority of the task representing the newly created software component affects the overall schedulability. Note that in this stage, parameters such as the WCET are estimated and need further refinement to correctly analyze the system’s behavior with EPSim. We refer to Section 5.5 for such a detailed analysis.
5.4 Characterization of the Evolved System

Similar to the analysis of the existing system, the newly created algorithm is deployed on its intended platform, i.e., the Raspberry Pi of the ACC-car, so that its (worst-case) execution time can be analyzed. Therefore, a set of inputs can be generated to trigger different execution paths in the algorithm (as in Section 5.1). Note that a certain execution path not only depends on the algorithm’s inputs but also on the internal state of the software component and/or the platform. In addition to an MC/DC analysis, we relied on techniques described in (Li et al. 2020) that force the processor into a certain state, alter the memory alignment, etc. This resulted in an observed WCET of 0.183 ms for the road sign recognition feature.

5.5 Application Response Time Analysis

Once the additional feature is developed and its execution time on the platform is analyzed, the final step of our methodology relies on EPSim. It refines the preliminary analysis described in Section 5.3. In particular, the WCET estimated in Section 5.3 is now precisely defined due to the characterization of the component in the previous step. For the periodicity it was decided that the task executing the additional component should operate at the highest priority with a minimal periodicity of 1.07 ms. This ensures that in the situation where the car drives at the maximum speed of 0.7 m/s, the color sensor is able to read the surface four times each 3 mm: (3 mm / 0.7 mm/ms ) / 4 = 1.07 ms.

Given this configuration, EPSim can be used to execute a schedulability analysis and provide the expected WCRT for each software component. Table 2 summarizes the configuration of the evolved system and the WCRT analysis results obtained with EPSim. When comparing the schedulability results in Table 2 with the ones in Table 1, one will notice that the additional higher priority task increases the response times for tasks with ID 1 through 13. This will affect the behavior of the application, which can be analyzed with EPSim using the virtual Hardware-in-the-Loop (vHiL) analysis.

To set up such a vHiL analysis, a co-simulation between the model of the embedded platform (EPSim) and the application (the Simulink® model) is required. Therefore, an FMU is created containing (i) the plant

<table>
<thead>
<tr>
<th>ID</th>
<th>SW component</th>
<th>Priority</th>
<th>Period (ms)</th>
<th>Offset (ms)</th>
<th>WCET (ms)</th>
<th>WCRT (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Read position of left encoder</td>
<td>5</td>
<td>10</td>
<td>0</td>
<td>0.206</td>
<td>0.392</td>
</tr>
<tr>
<td>2</td>
<td>Read position of right encoder</td>
<td>6</td>
<td>10</td>
<td>0</td>
<td>0.212</td>
<td>0.605</td>
</tr>
<tr>
<td>3</td>
<td>Read intensity from color sensor</td>
<td>7</td>
<td>10</td>
<td>0</td>
<td>0.177</td>
<td>0.783</td>
</tr>
<tr>
<td>4</td>
<td>Read speed from left encoder</td>
<td>8</td>
<td>10</td>
<td>0</td>
<td>0.287</td>
<td>1.255</td>
</tr>
<tr>
<td>5</td>
<td>Read speed from right encoder</td>
<td>9</td>
<td>10</td>
<td>0</td>
<td>0.287</td>
<td>1.544</td>
</tr>
<tr>
<td>6</td>
<td>Read battery voltage</td>
<td>10</td>
<td>10</td>
<td>0</td>
<td>0.180</td>
<td>1.725</td>
</tr>
<tr>
<td>7</td>
<td>Read distance from ultrasonic device</td>
<td>11</td>
<td>10</td>
<td>0</td>
<td>0.217</td>
<td>1.943</td>
</tr>
<tr>
<td>8</td>
<td>Steer power left motor</td>
<td>15</td>
<td>10</td>
<td>3</td>
<td>0.118</td>
<td>0.420</td>
</tr>
<tr>
<td>9</td>
<td>Steer power right motor</td>
<td>15</td>
<td>10</td>
<td>3</td>
<td>0.115</td>
<td>0.301</td>
</tr>
<tr>
<td>10</td>
<td>Execute “FromDS2DS” component</td>
<td>12</td>
<td>10</td>
<td>0</td>
<td>0.022</td>
<td>0.208</td>
</tr>
<tr>
<td>11</td>
<td>Execute “ACC controller” component</td>
<td>13</td>
<td>10</td>
<td>2.5</td>
<td>0.0045</td>
<td>0.254</td>
</tr>
<tr>
<td>12</td>
<td>Execute “line follower” component</td>
<td>14</td>
<td>10</td>
<td>2.5</td>
<td>0.0044</td>
<td>0.299</td>
</tr>
<tr>
<td>13</td>
<td>Execute “battery monitor” component</td>
<td>50</td>
<td>10</td>
<td>7.5</td>
<td>0.0048</td>
<td>0.234</td>
</tr>
<tr>
<td>14</td>
<td>Execute “read and process barcode” component</td>
<td>1</td>
<td>1.07</td>
<td>0</td>
<td>0.183</td>
<td>0.184</td>
</tr>
</tbody>
</table>
model simulating the movement of the two Lego® cars, (ii) the ACC algorithm, (iii) the line following algorithm, and (iv) the barcode processing algorithm. For the co-simulation configuration, the necessary information regarding the FMU is defined, such as its time step, input and output ports, and input data. This allows one to simulate the application, residing in the FMU, as if it were deployed on the actual platform.

To validate the correctness of EPSim, and more specifically, the control models that (virtually) run on the embedded platform, we compared the vHiL-results with those obtained from a HiL-simulation. For the HiL-setup, we deployed the different software components of the evolved system on the (real) Raspberry Pi of the ACC-car, configured its RTOS, and connected the Raspberry Pi with a Speedgoat Real-Time Target machine on which the plant model and the algorithms of the lead car is deployed. The results obtained are shown in Figure 5. The position of the lead car, and the desired and safety distance stem from the plant model. For the position of the ACC-car, both the EPSim (vHiL) and the HiL-results are shown. We observed that during normal operation, the results (bottom-right figure) of EPSim closely resemble those obtained from the HiL simulation. When the lead car executes an emergency stop, as detailed in the top-right figure, we observe that the ACC-car in the HiL setup switches to an emergency stop to avoid a collision. The sensitivity analysis results obtained from the co-simulation with EPSim also indicate the avoidance of a collision. However, we also observe that the behavior of the control algorithm differs. It attempts to maintain a certain distance by moving the car back and forth instead of executing an emergency stop as observed with HiL simulation. This can be explained by the fact that the models used in EPSim abstract certain details, which result in slight differences in, for example, communication delays. In this particular case study, the control algorithm receives an input a bit later so that the emergency stop mode is not reached.

6 CONCLUSIONS AND FUTURE WORK

In this paper, we have shown the embedded platform simulator EPSim that allows one to model an embedded platform using a library consisting of processors, tasks, I/O modules, and communication buses. The embedded components of the library are modeled using the DEVS formalism and can easily be configured and combined through an API or, at a higher abstraction level, using a domain-specific language. To allow engineers to assess the deployed behavior of application software on the embedded platform, EPSim facil-
Vanherpen, De Meulenaere, Vanommeslaeghe, and Maes

Vanherpen, De Meulenaere, Vanommeslaeghe, and Maes itates the co-simulation of the embedded platform model with application models contained in FMUs. As such, we enable virtual integration in early stages of the development process.

We demonstrated how EPSim can support a design process in which an existing adaptive cruise control system is extended with a road recognition feature. Running an application response time analysis with EPSim, we evaluated the (virtually) deployed application behavior and concluded that the results obtained with EPSim correspond to great extent to the ones obtained from a traditional Hardware-in-the-Loop simulation. However, depending on the property in which one is interested, there might be a need for more detailed models resulting in more accurate results (with respect to a certain property).

The virtual integration of the control and embedded domain addresses some challenges in the design of CPS. When used in current development processes, it may support engineers in selecting the most suitable microcontroller, in exploring the optimal deployment of application software, and in facilitating fault injection to analyze how robust an application behaves under various, often rare, conditions. On the longer term, EPSim could also be used to analyze the root cause of a fault which occurs in the field. This also suggests that EPSim could be used as a building block of a digital twin representing the embedded platform. Additionally, the virtual integration capabilities of EPSim allow faster iteration, potentially enabling more agile development processes. Further research is required to investigate the use of EPSim in these contexts.

ACKNOWLEDGMENTS

This research was partially supported by Flanders Make, the strategic research centre for the manufacturing industry.

REFERENCES


AUTHOR BIOGRAPHIES

KEN VANHERPEN is a post-doctoral researcher at the Faculty of Applied Engineering at the University of Antwerp and founder of DT Valley. His email address is ken.vanherpen@uantwerpen.be.

PAUL DE MEULENAERE is a professor at the Faculty of Applied Engineering at the University of Antwerp. His email address is paul.demeulenaere@uantwerpen.be.

YON VANOMMESLAEGHE is a doctoral student at the Faculty of Applied Engineering at the University of Antwerp. His email address is yon.vanommeslaeghe@uantwerpen.be.

DAVY MAES is senior project leader and team lead SmartX at Flanders Make. His email address is davy.maes@flandersmake.be.