A CO-SIMULATION APPROACH FOR THE EVALUATION OF MULTI-CORE EMBEDDED PLATFORMS IN CYBER-PHYSICAL SYSTEMS

Yon Vanommeslaeghe
Bert Van Acker
Ken Vanherpen
Paul De Meulenaere

Flanders Make - AnSyMo/CoSys Core Lab
Faculty of Applied Engineering
University of Antwerp
Antwerp, Belgium
{yon.vanommeslaeghe, bert.vanacker, ken.vanherpen, paul.demeulenaere}@uantwerpen.be

ABSTRACT

The increasing complexity of Cyber-Physical Systems (CPS) causes a shift from traditional single-core embedded platforms to complex multi-core or even heterogeneous platforms to meet the performance requirements. However, the deployment of control or monitoring algorithms on such platforms is not straightforward as the temporal behaviour of the embedded platform can affect the functional behaviour of these algorithms. In this paper we propose a co-simulation strategy for multi-core embedded platforms. This allows to evaluate the impact of the deployment on the behaviour of the algorithm early on in the design process, this way reducing the risk of costly re-engineering and allowing for a more agile design process. Our approach is validated using an example case in different deployment scenarios, after which its applicability is demonstrated using a real-world use case.

Keywords: Multi-core Embedded Platforms, Cyber-Physical Systems, Control Embedded Co-Design, Co-Simulation, Model-Based Systems Engineering

1 INTRODUCTION

The design and development of Cyber-Physical Systems (CPS) is inherently a co-design process between different engineering domains. Indeed, even in its simplest form a CPS consists of a control algorithm, designed by a control engineer, runs on an embedded platform, deployed by an embedded engineer, and controls or monitors a physical system, designed by a mechanical engineer. However, these CPS are becoming increasingly complex and so is the software that is meant to monitor and control them. For example, autonomous vehicles need to process large amounts of information and make complex decisions in real-time, wind turbines need complex real-time sensor-fusion to estimate wind forces, etc. These complex algorithms are generally computationally expensive, thus requiring powerful hardware to run them. This demand for more powerful hardware causes a shift from tradition single-core embedded platforms to multi-core or even heterogeneous platforms, even further increasing the complexity.

To cope with this increasing complexity, engineers often follow a model-based design process. By building a model of the CPS and its environment, the system can be tested in simulations to verify its functionality, assumptions about its interacting environment, and end-to-end behaviour. This process generally starts by

capturing functional requirements (i.e. specifications) and deducing a possible system architecture. The mechanical engineer then starts by reasoning about the physical behaviour of the system in its environment and derives a model-of-the-physics (also referred to as a *plant model*). The control engineer can then determine a control strategy for the system using this plant model. The correct operation of the developed control algorithm and its compliance with the defined requirements can then be verified in so-called Model-in-the-Loop (MiL) simulations. Here, the control algorithm is executed in a closed-loop simulation, together with the plant model and a model of the environment in which the system will operate. After the MiL stage, the algorithm is prepared for embedded deployment by the embedded engineer. Besides necessary transformations, e.g. fixed-pointing or discretization, the algorithm also needs to be divided and mapped onto independent executable functions or tasks. The Real-Time Operating System (RTOS) running on the embedded platform will execute these tasks using a predefined scheduling mechanism, enabling the proper execution of the control algorithm.

The verification of the functional and temporal behaviour of the control algorithm after deployment occurs in a so-called Processor-in-the-Loop (PiL) or Hardware-in-the-Loop (HiL) test. In these tests, a closed-loop setup with the deployed control algorithm, running on the embedded platform, and the simulated physical behaviour (plant), running on either a non real-time target (PiL) or a real-time target (HiL), is used to verify correct operation of the algorithm and its compliance to the requirements. During these steps, differences in behaviour will generally be observed compared to the MiL stage. This is partly due to the transformations during deployment and partly due to effects related to the embedded platform, e.g. delays due to processing time, additional measurement noise, etc. As such, these tests are indispensable to verify the performance and behaviour of the control algorithm. However, they have some drawbacks: (i) the control algorithm needs to be fixed and adapted for embedded deployment, (ii) the actual embedded hardware needs to be available and (iii) specialized hardware, such as a real-time target, is needed. Additionally, (iv) they can require a lot of time and effort to be performed.

In this paper we present a co-simulation strategy where we make use of Discrete Event System Specification (DEVS) (Kim and Zeigler 1987) models to model the temporal behaviour of multi-core embedded platforms. We make use of the Functional Mockup Interface (FMI) standard (Blochwitz, Otter, Akesson, Arnold, Clauss, Elmqvist, Friedrich, Junghanns, Mauss, Neumerkel, et al. 2012) to enable the co-simulation of these embedded platforms with functional (application) models and plant models contained in functional mock-up units (FMUs). This allows us to construct virtual prototypes, which can be evaluated in a virtual HiL (vHiL) setup to evaluate both the temporal and functional behaviour of the system (application and hardware) under development. As this effectively combines different aspects of the different engineering domains, we refer to this as *cross-domain evaluation*.

The paper is organized as follows. In Section 2 we present the considered embedded platform effects as well as our overall approach. Next, in Section 3 we compare simulation results obtained using the presented approach to actual measurements on an embedded platform in different situations. Additionally, we apply our approach to a real-world use case. In Section 4 we compare our approach to related work. Lastly, in Section 5 we present our conclusions and future work.

2 METHODOLOGY

In the development of the presented co-simulation approach we explicitly considered three main aspects related to the temporal behaviour of software running on multi-core embedded platforms. These are first discussed in the following paragraphs, after which we give an overview of our approach. These considerations also serve to compare our work to related work in Subsection 4.

The first aspect that needs to be taken into account is the temporal behaviour of an algorithm running on an embedded platform, which is typically different from the execution within a simulation environment.

Typically, simulation models are executed according to the Zero Execution Time (ZET) principle (Kirsch and Sengupta 2006). This means that (i) the inputs are sampled, (ii) (part of) the model is computed, and (iii) the outputs are updated in a single time step. However, this is not realistic as these actions do take a finite amount of time on the embedded platform. As such, the execution of an algorithm on (embedded) hardware is better approximated using the Bounded Execution Time principle (Kirsch and Sengupta 2006), which does consider the time it takes to perform these actions. *As such, this non-zero execution time needs to be taken into account to properly simulate the temporal behaviour of the embedded platform and its impact on system performance.*

Secondly, we need to take the characteristics of the execution on multi-core platforms into account. While multi-core platforms offer a lot of computational power that can be used to run advanced control and monitoring algorithms, fully utilizing the potential of these platforms is not straightforward. There are potential pitfalls specific to multi-core platforms that can negatively impact the performance of the algorithm if not taken into account. These problems mainly arise from shared resources. In this paper we explicitly take the effects of the memory architecture into account. In multi-core systems, tasks running on different cores may communicate using shared variables, which are stored in shared (global) memory. To ensure data consistency, we assume individual tasks follow a Read-Execute-Write (REW) semantic, also referred to as Acquisition-Execution-Restitution (AER) (Maia, Nogueira, Pinho, and Pérez 2016). Here, all reads from global memory are performed at the start of the task (acquisition), after this the tasks executes, and lastly all writes to global memory are performed at the end of the task (restitution). Keeping this in mind, we identify two main mechanisms that can affect the temporal behaviour: **memory contention** and **mutual exclusion**. **Memory contention** occurs when multiple cores are accessing the global (shared) memory at the same time. In this case, the memory transfer speed perceived by each core will be reduced due to low-level mechanisms to resolve resource contention. **Mutual exclusion** is used to prevent data corruption when multiple tasks need to access a shared variable. For example when tasks running on different cores are communicating using a shared variable. In this case, mutexes are generally used to "lock" the variable while it is being accessed by a certain task. Other tasks then need to wait before they can access this variable. Both of these mechanisms can introduce unexpected delays during the execution of tasks, which can negatively affect system-level performance. *As such, we need to be able to model and simulate these effects to properly evaluate their impact on system-level performance early on in the design process.*

Last but not least, we need to consider the problem of what we refer to as *perceived time* in this paper. In a simulation environment, the current simulated time is always known precisely. Similarly, every time the model of the control algorithm is solved, the solver takes into account the exact time step since the previous evaluation to ensure a correct simulation. However, to prepare the algorithm for embedded deployment, the model is discretized with a fixed time step dt, corresponding to the expected period p at which the algorithm needs to run after deployment. After this, C-code, which can run on an embedded platform after compiling and linking, is generated from the model of the control algorithm. At this step, the time step dt is hardcoded in the code. It is then the responsibility of the embedded engineer to ensure the algorithm is executed at this exact time step. As a result, each time the algorithm is executed on the embedded platform it will assume that exactly dt amount of time has passed, even if this is not the case. As such, each task on the embedded platform can have its own notion of the passing of time, which may differ from reality. This is what we refer to as *perceived time*. Discrepancies between perceived and actual time can be caused by unexpected delays on the start of a task, for example due to memory contention or mutual exclusion, but also when a lower-priority tasks needs to wait for a high-priority task to be finished. These discrepancies can have a significant impact on both the temporal and functional behaviour of the algorithm as it has direct impact on every operation that requires this time to be known, such as integrals or derivatives. In a good design, steps are taken to avoid these effects, e.g. by ensuring that external inputs are sampled at fixed periods, or by introducing additional tasks that take care of data transfers at predetermined intervals. However, this doesn't mean these effects can be ignored. On the contrary, we need to be able to simulate the potential

impact of these effects to verify the correctness of the design. *As such, to correctly evaluate the impact of the embedded platform on the behaviour of the application, simulators need to consider this perceived time explicitly to ensure a representative simulation. Otherwise, the simulation may be too optimistic.*

2.1 Architectural Model

To enable the verification of the functional and temporal behaviour at an early stage of the design process, we need to glue the model of the application to the virtual execution of the embedded platform. In Figure 1, the conceptual overview of the simulator is shown.

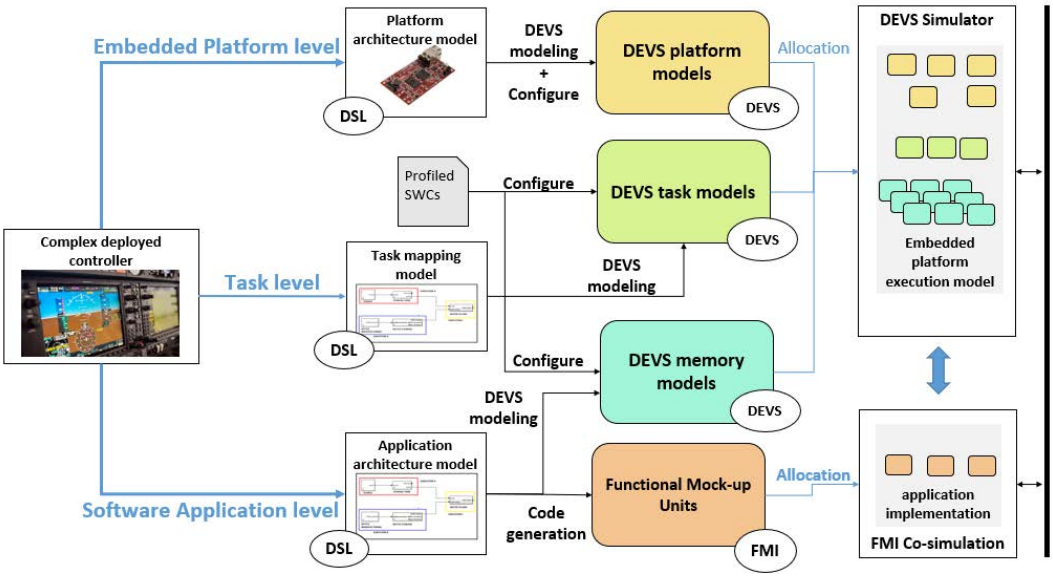

Figure 1: Conceptual overview cross-domain simulator.

We rely on a **Discrete Event Simulator (DEVS)** as backbone of the proposed simulator to enable correct and generic simulations of the aforementioned cross-domain simulations. DEVS is a formalism for modelling discrete-event systems in a hierarchical and modular way, rooted in systems theory. DEVS allows us to glue the application model to the embedded platform model by means of a *coupled model*. The coupled DEVS model depends on (i) the application model, (ii) the embedded platform model and (iii) the execution on the embedded target by means of RTOS tasks. The coupled model for the cross-domain simulation can be implemented using the following models:

- **Platform architecture:** This model is used to (i) compose the partial coupled DEVS model comprising the *processor core*, *scheduler* and *memory architecture* and (ii) configure the atomic DEVS models to match the real embedded platform.
- **Task mapping:** This model is used to (i) add the identified RTOS tasks to the coupled DEVS model of the embedded platform and (ii) configure these tasks to match the RTOS task configuration and estimated execution time on the corresponding embedded platform.
- **Application architecture:** This model is used to (i) add the application variables to the coupled DEVS model of task model and (ii) link the application model to the coupled DEVS model.

To enable heterogeneity within the application models, we propose the use of the **Functional Mock-up Interface (FMI) standard**. This enables to *co-simulate* different models with their own solver in a co-simulation environment. This way, all components can be modeled using the most appropriate formalism at

the correct level of abstraction (Mosterman and Vangheluwe 2004). The use of this FMI standard within the simulator makes it tool-independent. A key challenge is merging the FMI co-simulation environment with the DEVS base simulator. In subsequent subsections, each model and comprising modeling elements are discussed in detail.

2.2 Platform architecture

A first model which serves as input for the cross-domain simulations is the *platform architecture model*. The modeling of an embedded platform can be very complex as it can be modeled at different abstraction levels, depending on the goal of the simulation, ranging from very high-level end-to-end latency models to low-level digital circuit models. The selection of the modeling abstraction is a trade-off between accuracy and modeling effort. We choose to model the embedded platform using 3 main building blocks, the **core**, the **scheduler** and the **memory architecture**.

A **core** is a computational unit which is able to execute a program, e.g. an RTOS task. Within the DEVS model, the core not only executes the RTOS tasks, but also triggers memory accesses to perform *read/write operations*. The **scheduler** is the mechanism responsible for deciding which task should be executing at a particular task on an available core. In the current paper, we consider a priority-based non-preemptive scheduler. Lastly, the **memory** is the storage mechanism supporting the computational units of the embedded platform. Typically in embedded platforms, an architecture of different memory types is present, ranging from high-speed on-chip caches to lower-speed RAM, enabling inter-core communication. In our setup, we make an abstraction of this complex memory architecture and used a simplified memory architecture which focuses on the temporal effects of the (shared) memory access of multiple cores.

2.3 Application architecture

A second model used to define the co-simulation setup is the *application architecture* of the complex controller connected to its (controlling) environment. This application model not only defines the *FMI co-simulation* and the necessary *FMI-DEVS interfaces*, it also alters the coupled DEVS model of the cross-domain simulation. Inspired by Component-Based Software Engineering approaches (CBSE) (Cai, Lyu, Wong, and Ko 2000), we defined the application architecture using **components**, **interfaces** and **variables**.

A **component** represents an element of the application architecture, containing a program to be executed and which passes data through defined in- and/or output ports. In this work we focus on the use of FMUs as application components. **Interfaces** define the interconnection between application components, enabling communication between them. These interface definitions enable us to extend the FMI co-simulation setup by adding FMU communication interfaces. This way, the application FMUs can interchange data via the provided FMU functions e.g. *fmiGetReal()*, *fmiSetReal()*, *fmiGetBoolean()*, etc. Lastly, **variables** represent software variables used to pass data through an interface. These variables may be shared between cores, and are therefore important for realistic simulations as they enable us to take *memory contention* and *mutual exclusion* into account

2.4 Task mapping

The last essential model to enable the cross-domain simulations is the *task mapping*, providing the glue between the application model and the platform model. Within the task model, the application components can be *mapped* onto RTOS tasks as they would be deployed on the real embedded platform. These tasks can be configured to enable correct or expected execution behavior on the modeled multi-core platform. Correct

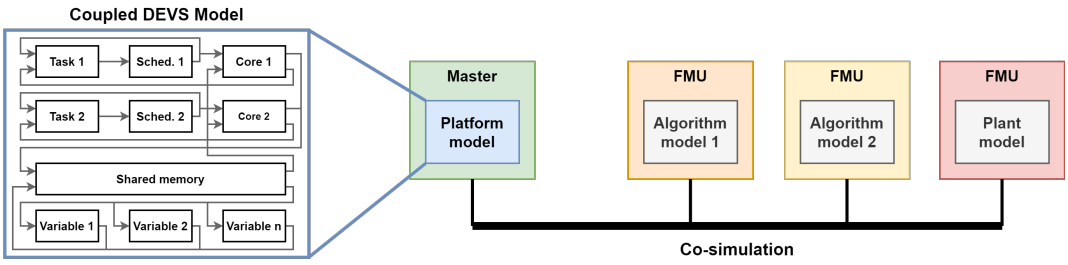

Figure 2: Conceptual overview of the co-simulation strategy.

task mapping and task configuration such as assigning *task priority*, is essential to obtain the expected functional behavior of the application. The two main building blocks to define and configure this task mapping are the **task** and the **task trigger**. A **task** is a program or function under control of the (real-time) operating system and is used to execute the aforementioned application components, whereas a **task trigger** is the triggering mechanism used to activate a task.

2.5 Co-Simulation Strategy

The previous sections mainly discuss the different models used to define the platform and application architecture. These models are used to configure atomic DEVS models representing the **task(s)**, **scheduler(s)** and **core(s)**, but also the different **variables(s)** and the **shared memory**. Additionally, these models are used to construct a coupled DEVS model representing the system under design. This coupled DEVS model can be used by itself to evaluate the *temporal behaviour* of the application on the embedded platform. As such, it can already be used to verify certain requirements, or to identify possible problems with the architecture, such as data inconsistencies, deadline misses, etc. However, to enable the verification of the *temporal behaviour* together with the *functional behaviour*, we need to be able to co-simulate the architectural DEVS model, together with a functional model of the application.

To enable the co-simulation of the architectural DEVS model, together with a functional application model, we make use of the FMI-standard. A conceptual overview of our co-simulation setup is shown in Figure 2. Here, application components corresponding to different tasks, as well as the plant model, i.e. a model of the physical system the application is meant to monitor or control, are contained in FMUs. In an FMI co-simulation environment, an FMI-master is responsible for orchestrating the co-simulation involving multiple FMUs. In our approach, we include the coupled DEVS model as part of the master. This DEVS model is used by the master to determine when to step the FMUs associated with different tasks, when and how to perform reads/writes, etc. This allows the master to take into account the *temporal behaviour* of the application on the embedded platform. Overall, this results in a hierarchical approach. A high-level master orchestrates the execution of the DEVS model and the plant model to allow for closed loop simulations. The DEVS model itself is then used as a low-level master to orchestrate the execution of the task FMUs. This is discussed in more detail below.

2.5.1 Low-level master

As previously mentioned, the DEVS model is used as a low-level master. It orchestrates the execution of the task FMUs, as well as the communication between different FMUs using read/write-operations, while preserving the *temporal behaviour* of the application on the embedded platform. The (coupled) DEVS model comprises multiple different atomic models. The atomic models representing the **task(s)** and **scheduler(s)** are standard models mimicking the functionality of a real-time operating system (RTOS). The **processor**

model used here is an extension of the common (single-core) processor model, with explicitly modeled states for memory accesses. This allows us to follow the REW-semantic and explicitly take into account the execution time of the read, execute, and write steps. The **variable** model is a simple model with two states that represent if the variable is currently free to be accessed or is currently occupied. This corresponds closely to how mutexes are used to manage access to shared resources on an embedded platform. As such, this allows us to take into account the effects of mutual exclusion on the *temporal behaviour*. Lastly, the **shared memory** model keeps track of memory access by different cores and corresponding transfer speeds. This allows us to take into account the effect of memory contention on the *temporal behaviour*.

State transitions in specific atomic models are used to trigger e.g. the execution of an FMU, communication between FMUs, etc. When a **task** enters the *running* state, associated read operations are performed first. When a variable has been read, the low-level master sets the associated input of the task FMU to the value stored in the **shared variable** model using the appropriate FMI function, e.g. *fmiSetReal*. This value can originate from a previous write operation or directly from the plant model. Initial values are defined in the **application architecture** model. This process is repeated for each variable read by the task, after which the **processor** enters the *running* state. This state corresponds to the execution of the **task**. After the execution time of the **task** has elapsed, the execution is complete and the *fmiDoStep* function of the associated task FMU, which contains a model implementation of the task, is called. It is here that we explicitly take the perceived time into account by always stepping the FMU with the associated task period instead of the precise elapsed (simulated) time. To allow this, we ensure the communication step size of each task FMU is equal to the task period. After the **task** has finished executing, the required write operations are performed. Write operations are performed almost identically to read operations. The difference is in the linking with FMI. When a write operation is finished, this signals the low-level master to get the current value of the associated output of the task FMU using the appropriate FMI function, e.g. *fmiSetReal*, and to store it in the **variable** model. This value can then later be read during other read operations, for example by other tasks. As such, the communication between different FMUs is explicitly done via the modelled **interfaces** and associated **variables**. This preserves the *temporal behaviour* of read/write operations on the embedded platform and the effect of potential data inconsistencies. Additionally, by separating the read, execute and write operations in this way, the execution of a task is effectively transformed to the BET paradigm.

2.5.2 High-level master

While the low-level master takes care of the co-simulation of the embedded platform and application models, to properly evaluate the behaviour of the system, it often needs to be evaluated in a closed-loop simulation with a plant model, which represents the physical part of the system. As such, we propose to use a high-level master to oversee the simulation of the plant model, and the DEVS model contained in the low-level master. This high-level master alternatingly executes the plant and the DEVS model, meaning it first steps the plant model and then runs the DEVS model up to that point in (simulated) time. As such, the step size of the simulation is dictated by the plant model (assuming a fix-step plant model). A different approach is also possible if a variable-step plant model is available. In this case, the step size could be determined by the time until the next DEVS transition.

3 RESULTS AND DISCUSSION

To validate the presented co-simulation strategy and to demonstrate the potential impact of the presented embedded platform effects on application performance, we first make use on an example case. In Subsection 3.1 we present this example case, after which we use our approach to analyze and predict the performance of the application in different deployment situations. After this, in Subsection 3.2, we apply our approach to a real-world use case and show simulation results.

The embedded platform considered in both cases is the MicroZed, which features a Xilinx Zynq-7000 system on chip (SoC) with two ARM Cortex-A9 cores running at 667MHz. The FPGA part of this system is not used. The considered memory read/write speed is characterized as $350 Mbit/s$ for **single-access** and $227 Mbit/s$ for **multi-access**. These figures were determined by running benchmarks on the actual platform.

3.1 Application on an example case

Consider a signal which represents the position x of a certain object over time. However, information about the current velocity v of this object is required. In this case the velocity may be estimated by periodically sampling the position value and calculating the discrete derivative as follows: $v_t = (x_t - x_{t-1})/\Delta t$. As an example case, we consider the deployment of this calculation in the context of a larger (dummy) task, e.g. an advanced state estimation or sensor fusion algorithm. We refer to this task as the estimator task. This task is executed with a period p. As such, during code generation the value of Δt in the discrete derivative calculation is fixed at $\Delta t = p$. However, if the task samples the position at the start, and there is jitter on the start time of the task, the actual time between samples will not match p exactly. A such, there will be a discrepancy between the *perceived* and *actual time*, which impacts the accuracy of the velocity estimate. Although in realistic cases one takes care to sample the inputs at a fixed periodicity, the goal of this example is to demonstrate the effect of the embedded platform on the estimated velocity. As such, the accuracy of the estimated velocity is used to evaluate the presented simulation strategy in different scenarios.

In the following paragraphs we first consider a reference implementation where the estimator task experiences no influence from other tasks. Next, we consider situations where the estimator task experiences delays due to multi-access and mutual exclusion, caused by a task running on a different core. In each situation, we predict the accuracy of the estimated velocity using the presented co-simulation strategy and compare this to the measured accuracy on the real-world counterpart of the embedded platform in a Hardware-in-the-Loop setup.

Reference The first situation is an ideal situation, in which the estimator task experiences no influences from other tasks. The estimator task is defined as follows: **Preferred core:** Core 1, **Period:** $250\mu s$, **Reading variables (size):** Variable1 (float[240]), Position (float), **Worst Case Execution Time:** $75\mu s$, **Writing variables (size):** Variable2 (float[240]), Estimated velocity (float).

This configuration is simulated using the presented co-simulation strategy to predict the accuracy of the velocity estimate. This prediction is then compared to the measured accuracy on the real-world counterpart of the modelled multi-core embedded platform in a Hardware-in-the-Loop setup. Results are shown in Figure 3, which shows a section of the simulation and measured traces on the left. Error histograms of the estimated velocity are shown for the simulation and embedded deployment are shown on the right. When comparing the error histograms, we see that the standard deviation for the embedded platform ($\sigma = 8.139$) is much larger than in the simulation ($\sigma = 0.459$). This may be explained by the presence of noise on the analog inputs used to measure the position signal.

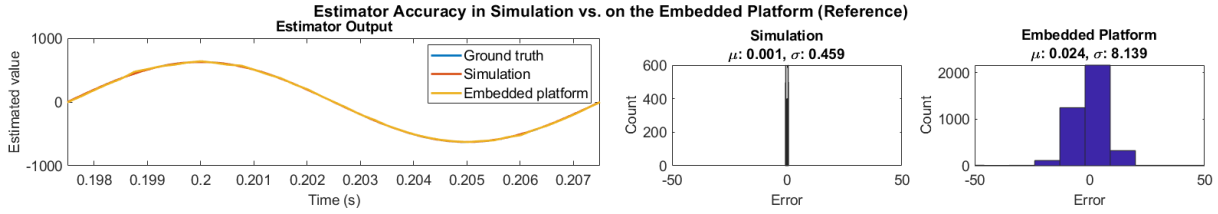

Figure 3: Reference accuracy in simulation and on the embedded platform.

Multi-access In the second scenario, we consider the combination of the estimator task as defined in the reference situation with a second, interfering task. This interfering task is defined as follows: **Preferred core:** Core 2, **Period:** $500\mu s$, **Reading variables (size):** Variable3 (float[240]), **Worst Case Execution Time:** $125\mu s$, **Writing variables (size):** Variable4 (float[240]).

In this situation, both tasks will read their input variables at the same time (multi-access) every $500\mu s$. As such they experience a reduced read rate, which delays the execution of the tasks. Results are shown in Figure 4. From the traces and the error histograms we can see that there is a larger error on the velocity estimate when compared to the reference situation. Additionally, we see that for this situation the predicted precision ($\sigma = 23.046$) is much closer to the actual precision on the embedded platform ($\sigma = 27.562$).

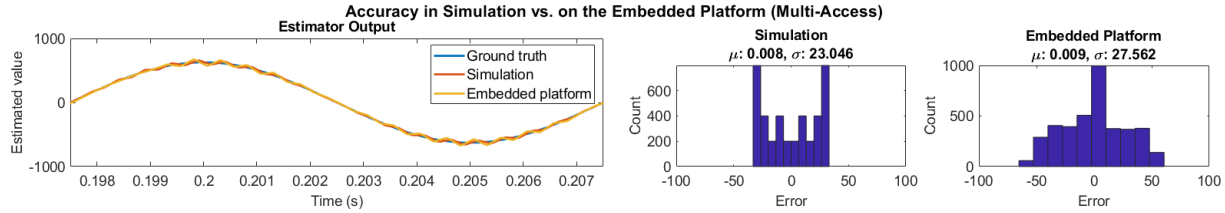

Figure 4: Accuracy in simulation and on the embedded platform, with multi-access.

Mutual exclusion In the third situation, we modify the definition of the interfering task so instead of Variable3, it reads Variable1 (the same as the estimator task). Additionally, we add a $1\mu s$ offset to the estimator task to ensure the interfering task starts reading Variable1 first. As such, Variable1 will be locked using the associated mutex, forcing the estimator task to wait until it is unlocked. This happens every other execution of the estimator task. Results are shown in Figure 5. Here we again see that the error on the velocity estimate is larger compared to the previous situations. Additionally, the predicted precision ($\sigma = 38.979$) is again closer to the actual precision on the embedded platform ($\sigma = 39.262$).

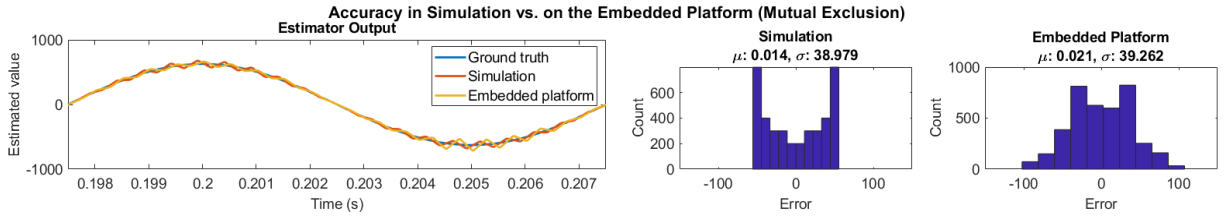

Figure 5: Accuracy in simulation and on the embedded platform, with mutual exclusion.

3.2 Application on a real-world use case

Lastly, we apply the presented approach to a real-world use case. Here, we consider the embedded deployment of an advanced, energy efficient load angle control algorithm for brushless DC motors (De Viaene, Verbelen, Derammelaere, and Stockman 2018). In short, this algorithm consists of two main parts: a load angle estimator and a load angle controller, which are both mapped to different tasks. Additionally, the estimator task contains a watchdog which monitors the estimated load angle and will overrule the controller when it gets too high. This is done to prevent the motor from losing synchronism, for example when the load suddenly increases. The time it takes for the watchdog to respond is an important factor to evaluate system performance. We make use of the presented approach to simulate and evaluate the performance of this algorithm after deployment on a multi-core embedded platform. This predicted performance is then compared to the original Simulink model, made by the control engineer, which does not take into account the embedded platform effects.

Figure 6 shows a section of simulation traces obtained from both a Simulink simulation and a simulated deployment of the energy efficient load angle control algorithm on a dual-core platform, using the presented approach. The plot show the load angle over time during a test sequence. At $t = 3s$ (vertical black dotted line) the load increases suddenly. When the load angle reaches a threshold of 100 degrees (horizontal red dotted line), the watchdog needs to respond. The time at which the watchdog responds is shown for both the Simulink simulation (vertical blue dotted line) and the simulated multi-core deployment (vertical red dotted line). Here, we see that the watchdog responds slightly later (~5ms) in the multi-core simulation (orange) than in the Simulink simulation (blue). However, this has a big impact on the overshoot of the load angle above the threshold (5.4 vs. 1.7 degrees). As such, these simulation results provide valuable feedback to the control engineer to re-evaluate and possible reconfigure the algorithm to minimize this overshoot if necessary, while taking into account the effects of the deployment.

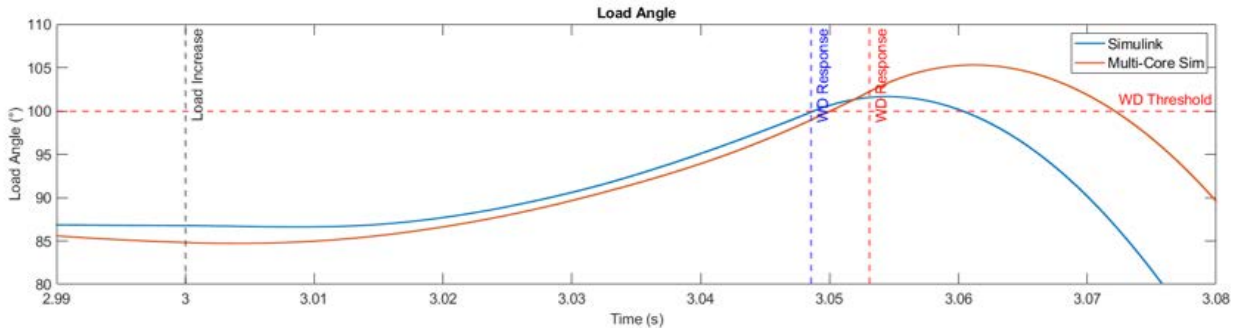

Figure 6: Section of simulation traces for the load angle control case.

4 RELATED WORK

Common hardware/software co-design frameworks generally follow the basic principle of the platform-based design (Sangiovanni-Vincentelli and Martin 2001) methodology: the "orthogonolization of concerns", i.e. separation of different aspects of the design. These frameworks generally only consider "embedded" parameters, i.e. the *temporal behaviour* instead of system level performance, i.e. the *temporal behaviour* of the embedded platform together with the *functional behaviour* of the application. For example, Oueslati et al. (Oueslati, Cuenot, Deantoni, and Moreno 2019) present both an analytical and simulation-based approach to determine bus interference experienced by applications running on a multi-core SoC. However, they do not investigate the impact of this bus interference on the actual *functional behaviour* of the application. However, to support more agile design processes, the evaluation of the *temporal* together with the *functional behaviour* should be made possible earlier in the design process, perferably at the model level.

Numerous tools already exist that allow engineers to evaluate the deployed behaviour of a control algorithm at a higher level of abstraction, i.e. the model level. TrueTime (Henriksson, Cervin, and Årzén 2003) is a Simulink toolbox which includes a kernel block and networking blocks. The kernel block enables the transformation from a ZET to a BET model by executing the model as it would be scheduled by an RTOS. To allow this, TrueTime requires code to be generated from the model, either Matlab scripts or C++ code, which allows the *perceived time* to be preserved by hardcoding the timestep. However, while TrueTime might be used to simulate distributed systems, it is not possible to simulate multi-core systems with shared resources. As such, effects from memory accesses cannot be taken into account.

T-Res (Cremona, Morelli, and Di Natale 2015) extends on the usability of the TrueTime approach by using triggered subsystems instead of generated code. While this does facilitate the integration with existing Simulink models, the main drawback to this approach is that it does not preserve the *perceived time* by default. Indeed, Simulink requires triggered subsystems to inherit the sample time, which means that every

time a subsystem is triggered the exact time step is used instead of the expected time step. As such, this may hide some of the effects of the embedded platform on the behaviour of the algorithm by essentially making the simulation too correct. While it is possible to preserve the perceived time, even when using triggered subsystems, this would require the model to be manually discretized at the expected time step or other workarounds. Additionally, T-Res relies on external simulators for the scheduling simulation.

Mertens et al. (Mertens, Vanherpen, Denil, and De Meulenaere 2019) improve on this by proposing the use of a Simulink library of components that represent different parts of the embedded platform. They enable a tight integration with Simulink and remove the need for external simulators by relying on SimEvents (Clune, Mosterman, and Cassandras 2006), a built-in discrete event simulator in Simulink. However, they also make use of triggered subsystems. As such, this method has the same drawback regarding the preservation of *perceived time*. They also focus on single-core systems and do not explicitly consider memory accesses.

Li et al. (Li, Mani, Mosterman, and Hübscher-Younger 2016) and Brandberg and Di Natale (Brandberg and Di Natale 2018) demonstrate that SimEvents can also be used to simulate multi-core platforms, by simulating the scheduling and memory access delays. Additionally, Brandberg and Di Natale provide support for multiple abstract memory access patterns. They also compare their method to TrueTime and T-Res, showing that using SimEvents allows for better simulation performance. However, as with the work of Mertens et al., they also rely on triggered subsystems to link the embedded platform model (SimEvents) to the application model (Simulink), meaning that workarounds are needed to preserve the effect of *perceived time*.

5 CONCLUSIONS AND FUTURE WORK

We present a co-simulation strategy for multi-core embedded platforms. This enables the evaluation of the *functional* and *temporal behaviour* of control and monitoring algorithms after deployment earlier in the design-process. This allows for a more agile design process by allowing design decisions to be evaluated quickly in simulation. This reduces the risk of costly re-engineering should the system not meet requirements after deployment. We achieve this by explicitly modeling (i) the application architecture, (ii) the embedded platform architecture, taking into account the effects of shared resources on multi-core platforms, and (iii) the task mapping. We make use of the FMI standard to enable the co-simulation of the embedded platform model, modelled using DEVS, and the application model, contained in FMUs. This not only ensures tool-independence, meaning that all components can be modeled using the most appropriate formalism at the correct level of abstraction, but also allows taking into account time and data synchronization.

In the future, we plan to extend this work in different ways. First of all, we currently only support one memory access pattern, whereas multiple different patterns may be found in practice. As such, we plan to include support for multiple different patterns. Secondly, we plan to extend the configurable DEVS models allowing the modelling of heterogeneous embedded platforms, i.e. a combination of a CPU with a GPU or FPGA, as these platforms are becoming more common. Lastly, we plan to further verify the correctness and efficiency of our simulator by comparing it to existing solutions and by applying our approach to more complex case studies.

ACKNOWLEDGMENTS

This research was supported by Flanders Make, the strategic research centre for the manufacturing industry in Belgium, with the Model-based Force Measurements (MoForM) project.

REFERENCES

Blochwitz, T., M. Otter, J. Akesson, M. Arnold, C. Clauss, H. Elmqvist, M. Friedrich, A. Junghanns, J. Mauss, D. Neumerkel et al. 2012. "Functional mockup interface 2.0: The standard for tool independent exchange of simulation models". In *Proceedings of the 9th International MODELICA Conference; September 3-5; 2012; Munich; Germany*, Number 076, pp. 173–184. Linköping University Electronic Press.

Brandberg, C., and M. Di Natale. 2018. "A simevents model for the analysis of scheduling and memory access delays in multicores". In *2018 IEEE 13th International Symposium on Industrial Embedded Systems (SIES)*, pp. 1–10. IEEE.

Cai, X., M. R. Lyu, K.-F. Wong, and R. Ko. 2000. "Component-based software engineering: technologies, development frameworks, and quality assurance schemes". In *Proceedings Seventh Asia-Pacific Software Engeering Conference. APSEC 2000*, pp. 372–379. IEEE.

Clune, M. I., P. J. Mosterman, and C. G. Cassandras. 2006. "Discrete event and hybrid system simulation with simevents". In *Proceedings of the 8th international workshop on discrete event systems*, pp. 386–387.

Cremona, F., M. Morelli, and M. Di Natale. 2015. "TRES: a modular representation of schedulers, tasks, and messages to control simulations in simulink". In *Proceedings of the 30th Annual ACM Symposium on Applied Computing*, pp. 1940–1947.

De Viaene, J., F. Verbelen, S. Derammelaere, and K. Stockman. 2018. "Energy-efficient sensorless load angle control of a BLDC motor using sinusoidal currents". *IET Electric Power Applications* vol. 12 (9), pp. 1378–1389.

Henriksson, D., A. Cervin, and K.-E. Årzén. 2003. "TrueTime: Real-time control system simulation with MATLAB/Simulink". In *Proceedings of the Nordic MATLAB Conference*. Copenhagen, Denmark.

Kim, T. G., and B. P. Zeigler. 1987. "The DEVS formalism: hierarchical, modular systems specification in an object oriented framework". Technical report, Institute of Electrical and Electronics Engineers (IEEE).

Kirsch, C. M., and R. Sengupta. 2006. "The evolution of real-time programming". *Handbook of Real-Time and Embedded Systems*, pp. 11–1.

Li, W., R. Mani, P. J. Mosterman, and T. Hübscher-Younger. 2016. "Simulating a multicore scheduler of real-time control systems in simulink.". In *SummerSim*, pp. 11.

Maia, C., L. Nogueira, L. M. Pinho, and D. G. Pérez. 2016. "A closer look into the aer model". In *2016 IEEE 21st International Conference on Emerging Technologies and Factory Automation (ETFA)*, pp. 1–8. IEEE.

Mertens, J., K. Vanherpen, J. Denil, and P. De Meulenaere. 2019. "A library of embedded platform components for the simulation of real-time embedded systems". In *2019 Spring Simulation Conference (SpringSim)*, pp. 1–12. IEEE.

Mosterman, P. J., and H. Vangheluwe. 2004. "Computer automated multi-paradigm modeling: An introduction". *Simulation* vol. 80 (9), pp. 433–450.

Oueslati, A., P. Cuenot, J. Deantoni, and C. Moreno. 2019. "System Based Interference Analysis in Capella".

Sangiovanni-Vincentelli, A., and G. Martin. 2001. "Platform-based design and software design methodology for embedded systems". *IEEE Design & Test of Computers* vol. 18 (6), pp. 23–33.