INTEGRATING DEVS AND FMI 3.0 FOR THE SIMULATED DEPLOYMENT OF EMBEDDED APPLICATIONS

Yon Vanommeslaeghe[a b], Bert Van Acker[a b], Joachim Denil[a b], and Paul De Meulenaere[a b]

[a]Cosys-Lab (FTI), University of Antwerp, Belgium
[b]AnSyMo/Cosys, Flanders Make, Belgium
{yon.vanommeslaeghe, bert.vanacker, joachim.denil, paul.demeulenaere}@uantwerpen.be

ABSTRACT

The demand for ever more performant, intelligent, and safer cyber-physical systems (CPS) drives the development of increasingly complex control and monitoring algorithms. Embedded deployment of these complex algorithms is not straightforward, as the temporal behavior of the embedded platform can affect their functional behavior. It is desirable to evaluate this effect as early as possible in the development process to avoid costly redesigns. In this paper, we present a co-simulation approach to evaluate the expected performance of embedded applications after deployment. To achieve this, we first show how Discrete EVent System specification (DEVS) models can be included in Functional Mock-up Units (FMUs) to capture the temporal behavior of the embedded platform. For this, we leverage features introduced in version 3.0 of the Functional Mock-up Interface (FMI) standard. We validate our approach using two example cases, one where we compare to an existing state-of-the-art approach, and one using real-world performance measurements.

Keywords: discrete event system specification, functional mock-up interface, co-simulation, embedded deployment, cyber-physical systems

1 INTRODUCTION

The term Cyber-Physical System (CPS) is generally used to describe systems which integrate software (cyber) and physical components, allowing this software to monitor and interact with the physical world through sensors and actuators. Examples of such systems range from intelligent thermostats to (semi-)autonomous vehicles, airplanes, etc. The complexity of these systems is ever-increasing, driven by a demand for ever more performant, intelligent, and safer systems. In this regard, a lot of recent advances have been achieved through software, by using more and more complex control and monitoring algorithms. In a typical development process, these algorithms are often developed and tested in a simulation environment, assuming, e.g., infinite computational resources, ideal interfaces, etc. However, during the development process, these algorithms will generally be deployed on an embedded platform, which does have finite computational resources, non-ideal interfaces, etc. As such, this act of embedded deployment can impact the behavior of the algorithm. Indeed, when deploying these algorithms, issues can be expected related to real-time performance, memory consumption, numerical precision, properties of the physical interfaces, etc. For this reason, Karsai et al. [1] argue that the design of CPS can only be accomplished in a co-design fashion, and that modeling the embedded platform needs to take part in it.

One major factor that can influence the behavior of an application after deployment is the temporal behavior of the algorithm running on the embedded system, which is generally different from its execution within a

Proc. of the 2024 Annual Simulation Conference (ANNSIM'24), May 20-23, 2024, American University, DC, USA
C. Ruiz-Martin, B. Oakes and R. Cárdenas, eds.

simulation environment. In simulation, models are often executed according to the Zero-Execution Time (ZET) principle [2], where (i) inputs are sampled, (ii) the model is computed, and (iii) its outputs are updated in a single time-step. However, these actions do take a finite amount of time when executing on an embedded platform. This execution time can become non-negligible for complex algorithms, leading to unexpected sensor-to-actuator delays, which can affect the behavior of control algorithms. Additionally, these applications generally have to share computational resources with other tasks running on the same embedded platform. This can cause further unexpected delays, e.g., if one task has to wait for a higher priority task to finish executing, further impacting performance. This effect was demonstrated by Morelli and Di Natale [3], who showed a difference in the behavior of a control system depending on the priorities of its individual tasks. This impact of the temporal behavior on the functional behavior of an application generally only becomes visible quite late in the development process, e.g., during Hardware-in-the-Loop (HiL) testing, leading to costly re-engineering if problems are found. As such, if we can model and simulate these effects, their impact can be evaluated earlier on in the development process. This would allow engineers to discover potential problems earlier on, reducing the required re-engineering effort.

In this paper, we present a co-simulation approach to evaluate system-level performance, i.e., the temporal together with the functional behavior, of control and monitoring applications in the context of cyber-physical systems. More specifically, we show how we can use the functionality introduced in version 3.0 of the Functional Mock-up Interface (FMI) standard to include Discrete EVent System specification (DEVS) models in Functional Mock-up Units (FMUs). We go on to show how this enables the evaluation of system-level performance, where we use DEVS models contained in FMUs to capture the temporal behavior of the embedded platform and its application. We link these models to other FMUs containing the actual implementation of the algorithm under design, i.e., which capture their functional behavior, to arrive at a co-simulation setup. We demonstrate this for two example cases. First, we compare our approach to another approach from the state-of-the-art for an example control system. Then, to further validate our approach, we compare simulation results obtained using our approach against the measured real-world performance of an example monitoring algorithm.

2 RELATED WORK

A number of tools already exist that allow engineers to predict the deployed behavior of an application in simulation. One such tool is TrueTime [4], which is a Simulink toolbox that allows engineers to include execution delays into Simulink models by executing parts of the model as it would be scheduled by a Real-Time Operating System (RTOS). A drawback of the TrueTime approach is that it requires code to be generated from the parts of the model representing the application, either MATLAB scripts or C++ code, which then need to be re-integrated in the Simulink model. This limits its usability. T-Res [5] extends on the usability of the TrueTime approach by using triggered subsystems instead of generated code to represent different parts of the application. While this does facilitate its integration with existing Simulink models, T-Res does rely on external simulators, limiting the portability of the models.

Mertens et al. [6] improve on this by removing the need for external simulators. Instead, they rely on SimEvents [7], a built-in discrete event simulator within Simulink, to model the different parts of the embedded system. Like T-Res, they rely on triggered subsystems to link the temporal behavior to the functional behavior of the application. While Mertens et al. focus on single-core embedded platforms, Li et al. [8] and Brandberg and Di Natale [9] demonstrate that SimEvents can also be used to simulate multicore embedded platforms, by simulating, among others, memory access delays. Additionally, Brandberg and Di Natale add support for multiple abstract memory access patterns.

The previously mentioned approaches rely on MATLAB/Simulink as a simulation environment, which limits their integration with other modeling tools. As the design of cyber-physical systems generally encompasses multiple engineering domains, each often with specialized modeling and simulation tools, this is a major

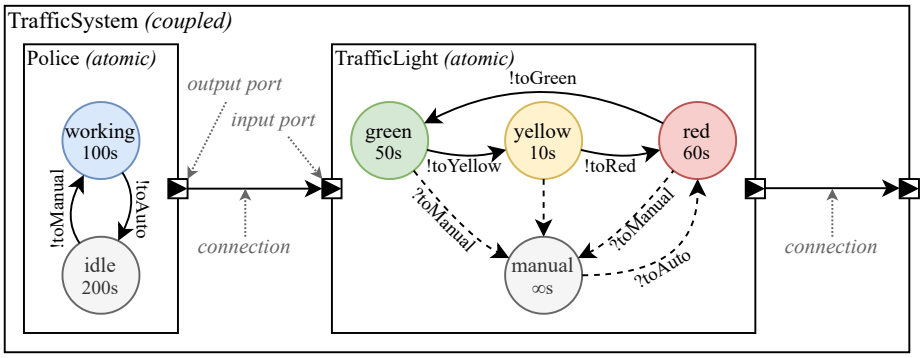


Figure 1: Illustration of a traffic system model, modeled using DEVS.

drawback of these approaches. To address this problem, in previous work [10], we made use of the FMI standard to enable interoperability with different modeling tools. Like Denil et al. [11], we made use of DEVS models to model the temporal behavior of the embedded platform. However, to get around the limitations of FMI 2.0, which did not explicitly support discrete event simulation, we opted to integrate these DEVS models in our importer. As such, our previous work relied on a custom importer, limiting the general applicability of that approach.

More recently, version 3.0 of the FMI standard added support for discrete event simulation. Ravi et al. [12] make use of this new discrete event support to enable timing-aware Software-in-the-Loop (SiL) simulation, specifically focussing on automotive applications. By using the newly introduced clocks to communicate timing events, they are able to separate the timing simulation and simulation orchestration. This allows them to use a generic importer to run the co-simulation, making for a more generic approach. However, in their case study, they make use of pre-computed timing traces to provide the temporal behavior. They do not show how the embedded platform simulation itself can be integrated in their setup.

3 BACKGROUND INFORMATION

This section provides an overview of the relevant background information regarding the Discrete EVent System specification (DEVS) and the relevant features introduced in version 3.0 of the Functional Mock-up Interface (FMI) standard. This mainly serves to provide reference information for the remainder of the paper.

3.1 The Discrete Event System Specification (DEVS)

This subsection provides a brief overview of the relevant aspects of the discrete event system specification. A more extensive discussion regarding the DEVS concepts can be found in [13].

DEVS was introduced by Zeigler [14, 15] as a basis for discrete-event modelling and simulation. It allows for the modeling of discrete-event systems at two main levels, using (i) *atomic* DEVS models and (ii) *coupled* DEVS models. These two types of models are discussed in the following subsections.

3.1.1 Atomic DEVS Models

At the lowest level, *atomic* DEVS models describe the behavior of a discrete-event system as a sequence of deterministic transitions between sequential states, including how it reacts to external input (events) and how it generates output (events) [13]. Figure 1 shows an illustration of two such atomic DEVS models, one of a police officer (“Police”), and one of a traffic light (“TrafficLight”).

An atomic DEVS model can be defined as follows:

$$atomicDEVS \equiv \langle S, ta, \delta_{int}, X, \delta_{ext}, Y, \lambda \rangle$$

With:

- S the *set of admissible sequential states*, e.g., for "TrafficLight": $S = \{green, yellow, red, manual\}$
- ta the *time advance function*, which models the time the system remains in a certain state, before transitioning to the next sequential state, e.g., $ta(green) = 50s$; $ta(yellow) = 10s$; ...
  Note: (i) instantaneous transitions can be modelled using $ta(s) = 0$, and (ii) if the system should remain in a state forever, this can be modelled using $ta(s) = +\infty$.
- δ_{int} the *internal transition function*, which models the transition from one state to the next sequential state, e.g., $\delta_{int(green)} = yellow$, $\delta_{int}(yellow) = red$, ...
- X the *set of admissible inputs*, e.g., for "TrafficLight": $X = \{toManual, toAuto\}$
- δ_{ext} the *external transition function*, which describes how the system responds to *external events* $x \in X$, e.g., $\delta_{ext}((green, e), toManual) = manual$, with e the *elapsed time* since transitioning to the current state s
  Note: the *time left* $\sigma = ta(s) - e$ is often used
- Y the *set of admissible outputs*, e.g., for "TrafficLight": $Y = \{toGreen, toYellow, toRed\}$
- λ the *output function*, which determines which output events are generated at the time of an *internal transition*, e.g., $\lambda(red) = toGreen$, $\lambda(green) = toYellow$, ...
  Note: (i) the state *before* the transition is used as input to λ, and (ii) output events are *only* generated on *internal* transitions and *not* on *external* ones

3.1.2 Coupled DEVS Models

At the higher level, *coupled* DEVS models describe a system as a network of coupled components, with connections between components denoting how they influence each other. More specifically, through a connection, output events of one component can become input events for another. These components can be atomic DEVS models, but also other coupled DEVS models. As such, DEVS allows for a hierarchical modelling approach [13]. Additionally, components can be connected to input/output ports of the encompassing coupled DEVS model.

Figure 1 shows an illustration of such a coupled DEVS model ("TrafficSystem"), consisting of two atomic DEVS model of a police officer and a traffic light. In *autonomous* mode, the traffic light automatically transitions between the states "green", "yellow", and "red". When the police officer starts working (transitions to the "working" state), an output event "toManual" is generated, which causes an external transition in the "TrafficLight" model ("?toManual"), transitioning it to a *manual* mode ("manual"). Similarly, when the police officer stops working, the traffic light is switched back to *autonomous* mode, starting in the "red" state.

3.2 The Functional Mock-Up Interface (FMI) Standard

This subsection provides a brief overview of the relevant aspects of the functional mock-up interface standard, in particular version 3.0. A more extensive discussion regarding these concepts can be found in [16], and the FMI standard itself [17].

The design of cyber-physical systems generally involved multiple different engineering domains, each with often their own specialized modeling and simulation tools. With this in mind, the FMI standard aims to increase the interoperability between different tools. It defines a standardized wrapper for models (with

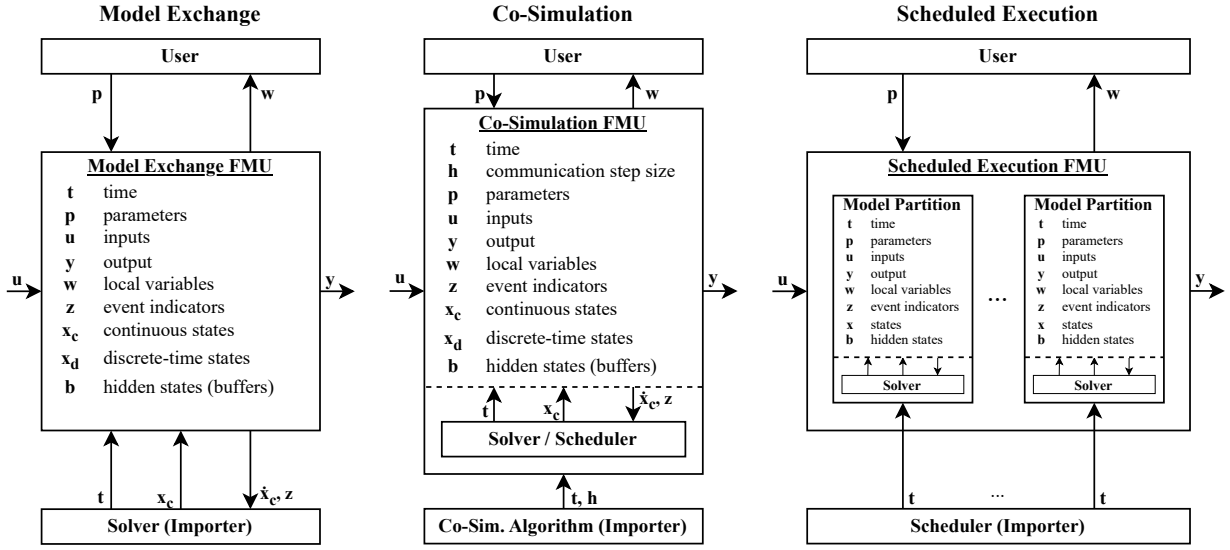


Figure 2: The different types of FMUs, including the new scheduled execution type [17].

their solvers), the Functional Mock-up Unit (FMU), with a standardized interface. This allows external tools, referred to as *importers*, to interact with these models in a standardized, tool-agnostic way. In this regard, version 2.0 of the FMI standard mainly supported the simulation of continuous systems. However, version 3.0 of the standard (FMI 3.0) adds a number of features to support discrete event simulation. It is some of these features that we rely on to integrate DEVS models in FMUs. These are described below.

First, FMI 3.0 introduced a new *event mode*, next to the previously supported *step mode*, which the importer can switch between when needed (if supported by the FMU). Event mode allows the explicit handling of discrete events, such as input events, time events, state changes, etc., without incrementing the internal time, as would normally happen in stepped mode. In addition to this, it adds the ability for FMUs to signal their early return from a $doStep$ call. When an FMU is in step mode, the importer can request the FMU to advance its internal time by a certain amount by calling the $fmi3DoStep$ function. However, the FMU can now use the return value *earlyReturn* to signal early return, i.e., that it could not, or did not, advance the internal time by the requested amount. Additionally, it can use $eventHandlingNeeded$ to signal that it has returned early because an event has occurred, and $lastSuccessfulTime$ to inform the importer of the internal time when the event occurred.

Second, it introduces the concept of input and output *clocks* to signal events. In this regard, the standard defines two main types of clocks, *time-based*, and *triggered* clocks. Time-based clocks are associated with an interval, which indicates the time between the last and next tick at any moment in simulation time. Such intervals can be queried or set by the *importer*. In contrast, *triggered clocks* have no a priori known interval, and it is up to the *importer* or the FMUs themselves to set/get the activation state of these clocks when needed [16]. Regarding time-based clocks, the standard discerns different types. However, in our approach, we specifically use *aperiodic countdown clocks*. For these types of clocks, the interval is set by the FMU after the occurrence of an event. This interval can then be queried by the importer to determine when this clock should tick. This is mainly intended to be used to handle time-delayed actions.

Last, FMI 3.0 introduced a new type of FMU, the *scheduled execution (SE) FMU*. These types of FMUs can expose individual model partitions, and allow the importer to trigger the execution of these different partitions using associated *clocks*. This essentially exposes the scheduling of the execution of the model (partitions) contained in the FMU. The types of FMUs defined in the standard are illustrated in Figure 2.

4 APPROACH

To enable the integration of DEVS models in functional mock-up units, we need to adapt the semantics of DEVS to those of FMI. More specifically, there are two main aspects we need to consider. First, we need to make sure that the FMI importer can be made aware when an internal transition is about to occur in the DEVS model, such that it can move the relevant FMUs into event mode. Second, we need to be able to communicate input/output events between DEVS models, or with other FMUs, using the standard FMI interface. In the following subsections, we describe how we address these two aspects, as well as provide some considerations and our motivation for our chosen approach. After this, in Section 4.3, we illustrate our approach using the traffic system example previously shown in Figure 1. After this, in Section 4.4, we describe how we expand on this approach to enable the simulated embedded deployment of an application (contained in one or more FMUs), using a DEVS model of the embedded platform.

4.1 Signaling Imminent Internal Transitions

It is up to the importer to move the relevant FMUs into event mode when needed to handle the occurrence of events. As such, the importer needs to be made aware of the (imminent) occurrence of events. For our DEVS models, this would mean the (imminent) occurrence of internal transitions.

One mechanism provided in version 3.0 of the FMI standard that could be used to signal imminent internal transitions to the importer, is the ability for an FMU to signal the early return from a $fmi3DoStep$ call, as described in Section 3.2. When the importer calls the $fmi3DoStep$ function, we could check if the requested step size is greater than the remaining time until the next internal transition in the DEVS model. If this is the case, we could advance the internal time up to the point of the internal transition, and signal the early return and need for event handling to the importer using the relevant return values. However, there are some drawbacks to this approach. First, the importer needs to explicitly support early return. As such, relying on this mechanism would likely limit in which tools these DEVS FMUs could be used. Second, in the case the importer does support early return, an FMU signaling such an early return might mean that the importer needs to roll back the state of other FMUs which did fully advance their internal time, such that they may be advanced instead to the $lastSuccessfulTime$ of the signaling FMU. Similarly, this is a feature that needs to be explicitly supported by the FMUs. In our experience, most FMUs do not support this feature. As such, this would likely severely limit the co-simulation setups in which these DEVS FMUs could be used.

To avoid the mentioned drawbacks of relying on the early return mechanism, we opt instead to use another feature introduced in FMI3.0: the *countdown aperiodic clock*. More concretely, for our DEVS FMUs, we define a countdown aperiodic (input) clock, whose period is equal to the time until the next internal transition (σ). As such, a call to $fmi3GetInterval$ for this clock allows the importer to query the DEVS FMU for the time until the next internal transition. Then, when in event mode, when the importer instructs this clock to tick using a call to $fmi3SetClock$, we instruct the DEVS solver to perform the internal transition. As it is the responsibility of the importer (i) to keep track of when input clocks *should* tick, and (ii) to move the FMUs into event mode and to instruct the relevant clocks *to* tick at the right point(s) in time, we believe that relying on this mechanism for the DEVS FMUs makes it more likely for them to be useable in different (co-)simulation tools and scenarios.

4.2 Communicating Input/Output Events

As mentioned in Section 3.1, DEVS models can generate output events on internal transitions, and can transition between states in response to input events (external transitions). To enable the co-simulation of DEVS FMUs with other FMUs, we need to be able to communicate the occurrence of these events using some interface defined in the FMI standard. In our approach, we make use of triggered input/output clocks to

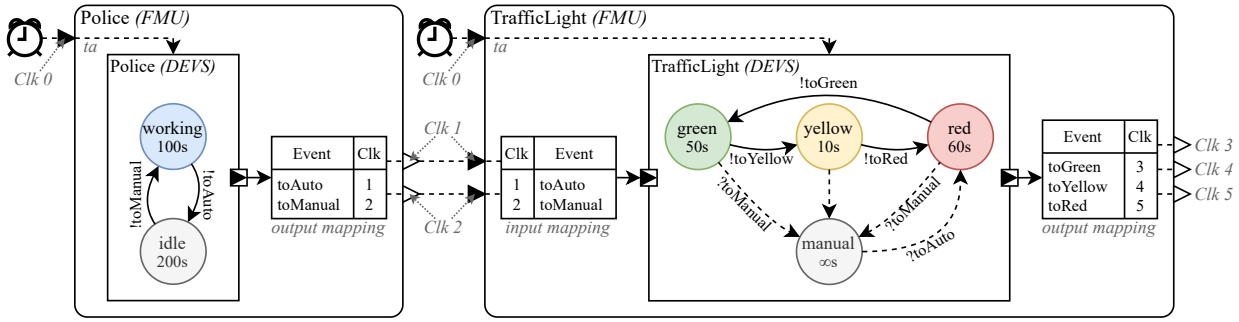


Figure 3: Modified traffic system example showing a co-simulation scenario with DEVS models contained in FMUs, illustrating the required adaptations.

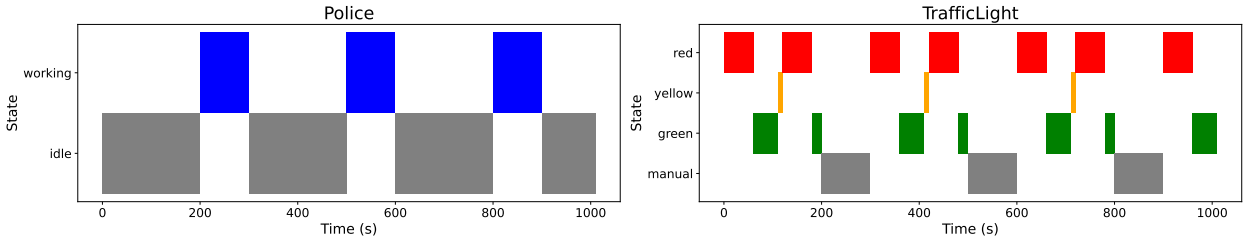


Figure 4: Results of the co-simulation, showing the states of the two DEVS models over time.

do this. More explicitly, we define a triggered input clock for each admissible input event in X and a triggered output clock for each admissible output event in Y. Then, we define a mapping, which associates each input clock (using its value reference) to a specific event in X. This mapping allows us to generate the correct input events on calls to $fmi3SetClock$. Similarly, we define a mapping between output events in Y and associated output clocks (by value reference). On internal transitions, we keep track of the generated output events. Then, this mapping allows us to report the correct clock activation state on a call to $fmi3GetClock$, with an output clock being active if the associated event has occurred and inactive otherwise.

4.3 Example Co-Simulation of DEVS Models Using FMI

We illustrate our approach using the traffic system example previously introduced in Section 3.1. Previously, this system was modeled using a coupled DEVS model, containing two atomic DEVS models, one for the police officer (“Police”), and one for the traffic light (“TrafficLight”) (Figure 1). Now, we instead wrap these two DEVS models in two FMUs, with the mentioned adaptations to adapt them to the FMI interface. This is illustrated in Figure 3. This figure shows the two FMUs, each containing one of the DEVS models. Each FMU has an input clock associated with the time advance (labeled *ta*, value reference 0), as described in Section 4.1. Additionally, each FMU has input and output clocks associated with the admissible input and output events. For these clocks, we also show the mapping between clocks (by value reference) and events (by unique identifier), as described in Section 4.2, as a table. The coupling between the two models in then achieved by connecting the correct input/output clocks of the FMUs. To implement this co-simulation setup, an FMU wrapper for the DEVS models, which implements the described adaptations, was implemented in Python. We then made use of UniFMU [18], which provides C-compatible binaries that implement the FMI interface, to turn them into “real” FMUs. This allowed us to import these DEVS FMUs and interact with them like any other FMU using FMPy [19]. Note, at the time of writing, UniFMU does not fully support FMI 3.0. As such, we extended the latest version with the FMI 3.0 functions required for our approach. Using this setup, we were able to co-simulate the two DEVS models using only the functions defined in the

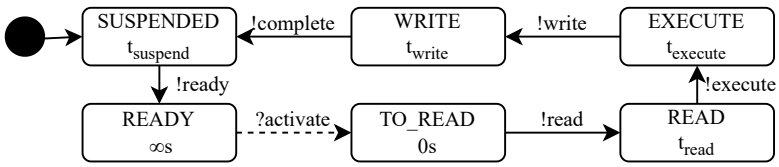


Figure 5: Illustration of the (atomic) DEVS model of a task.

FMI standard. For this, we did use our own, manually implemented importer and not a generic one. To the best of our knowledge, generic importers which fully support FMI 3.0 are not freely available at the time of writing. Results of this co-simulation are shown in Figure 4, which shows the state of the two DEVS models over time.

4.4 Simulated Embedded Deployment

As previously mentioned, our main goal is to evaluate the behavior of embedded applications in simulation as if it would be executing on the embedded platform, thus evaluating the effects of both the temporal and functional behavior. As such, we first need to capture these two aspects.

To model the temporal behavior of the application on the embedded platform, we make use of DEVS. In this paper, the example applications will be deployed on a single-core embedded platform, using a fixed-priority non-preemptive scheduler. To model this, two DEVS models are created, a *task* model, and a *scheduler* model. First, the *task* model captures the basic states of a task running on an RTOS, i.e., suspended (waiting), ready, and running. However, we assume tasks follow a Read-Execute-Write (REW) semantic, also known as Acquisition-Execution-Restitution (AER) [20]. As such, the running state is split into three sequential states, namely read, execute, and write, as shown in Figure 5. Making this distinction gives us the ability to explicitly consider access to peripherals, e.g., to get sensor values or to command actuators, and to link this to models of the functional behavior of these peripherals. Additionally, this task model contains information regarding its period, priority, etc., but also of the execution time of the read, execute, and write actions (respectively t_{read}, $t_{execute}$, and t_{write}). Currently, we obtain this execution time information by profiling parts of the application in isolation on the target embedded platform. As such, this does require some up-front deployment effort. However, this should remain minimal as this profiling can generally be done using a limited deployment, e.g., using a Processor-in-the Loop (PiL) setup. Even with this up-front effort, the presented approach provides an advantage as different parts of the application (e.g., tasks) could be profiled in isolation, after which different deployment strategies (e.g., task mapping, priorities, etc.) could be explored in simulation.

Second, the *scheduler* model captures the basic behavior of a fixed-priority non-preemptive scheduler. It keeps track of which tasks are in the ready state, and if one is currently running. If no task is running, it will signal the task with the highest priority to start running. Using these two models, we can construct a model of an embedded platform running any number of different tasks. This is illustrated in Figure 6, which shows an embedded platform model, consisting of a scheduler with two tasks. This embedded platform model is wrapped and turned into an FMU as described in the previous sections, making it compatible with the FMI standard. The task models generate output events when it transitions between certain states. More specifically, it signals reading, execution, and writing. These events are mapped to output clocks of the FMU as shown in the figure.

To capture the functional behavior of the application, we make use of *scheduled execution* FMUs, which contain the application code, i.e., the control or monitoring algorithm. Using the scheduled execution FMUs allows us to trigger the execution of the application code when specific clocks are active. By linking the execution of these FMUs to the activation of, e.g., “t1_execute” as shown in Figure 6, the application code

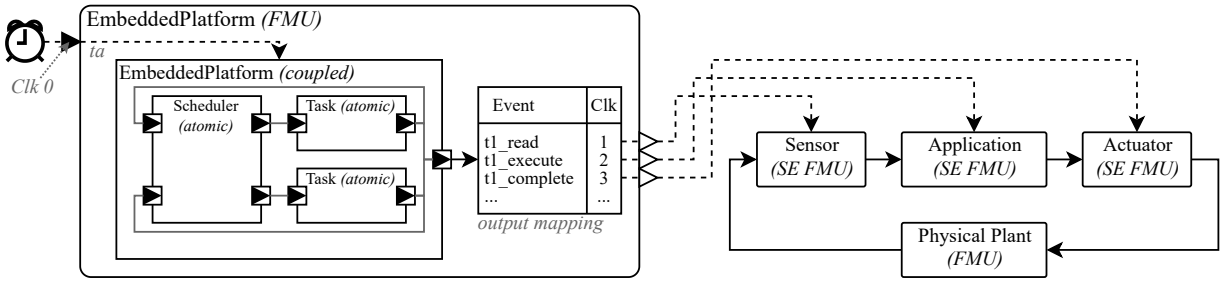


Figure 6: Illustration of a simulated embedded deployment using an embedded platform modelled in DEVS together with application models contained in scheduled execution FMUs.

is executed at the same point in time as it would if it were running on the actual embedded platform, thus linking the temporal to the functional behavior. We also use a similar mechanism with models of the interfaces, linked to clocks signalling relevant reads or writes, as also illustrated in Figure 6. Lastly, the model of the physical part of the system, the plant model, is included using a "regular" co-simulation FMU. As with the DEVS-only example, we use our own importer to orchestrate the co-simulation of these different types of FMUs.

5 RESULTS AND DISCUSSION

To validate our method for simulated embedded deployment of control and monitoring algorithms, we follow a two-step approach. First, in simulation, where we compare results obtained using our approach to those obtained using the approach proposed by Brandberg and Di Natale [9]. Second, we further validate our approach by using it to predict the performance of a simple monitoring algorithm in simulation, and then comparing this prediction to its actual performance after deployment on a real embedded system.

5.1 Validation in Simulation

As mentioned in the Section 2, Brandberg and Di Natale [9] present an approach to simulate the impact of multicore specific embedded platform effects, such as memory access delays, on the functional behavior of an application. Their approach makes use of SimEvents to model the temporal behavior of the embedded platform, while Simulink models contained in triggered subsystems provide the corresponding functional behavior of the application. While they focus on these multicore specific effects, their approach can also be used to simulate single-core embedded platforms. Indeed, their models are freely available and include a simple example case using a single-core system [21]. It is this simple example case that we use to validate our approach. In short, this case consists of three servo loops, running on a shared, single-core embedded platform, as three different tasks with different priorities and periods.

To validate our approach, we generate code from the three servo controller models corresponding to the three different tasks. We then use this code to build three scheduled execution FMUs. Additionally, we generate a regular co-simulation FMU of their plant model. We use these FMUs to reproduce their setup using our approach as described in Section 4.4. Note, this example case does not include explicit models of the interfaces. As such, we use latches to represent ideal sensors and actuators in our co-simulation model. From this, we obtain simulation traces which describe both the temporal and functional behavior of this example application. These traces are then compared to reference traces obtained using the models of Brandberg and Di Natale. An excerpt of the temporal trace, which matches between the two approaches, is shown in Figure 7.

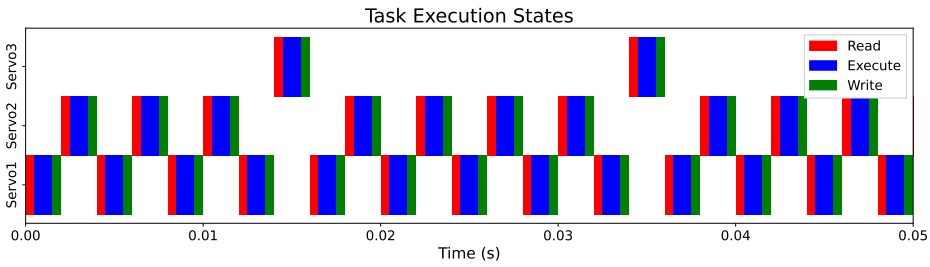
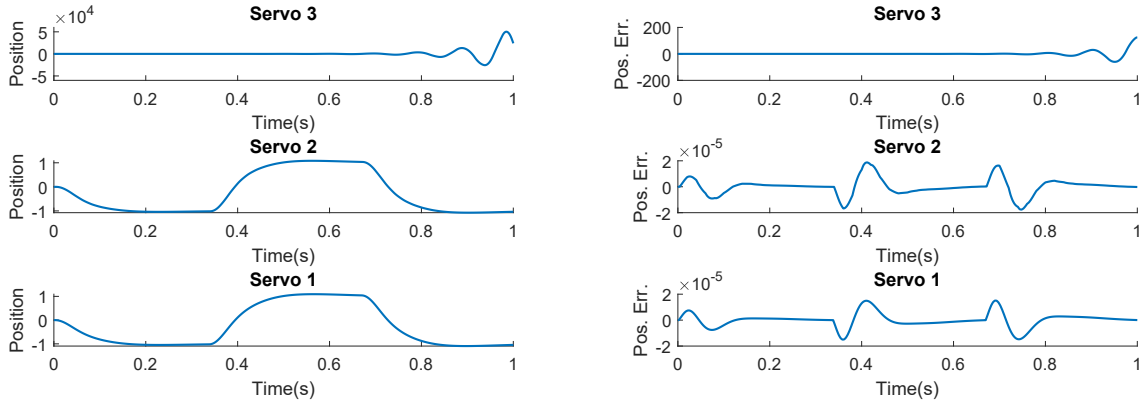


Figure 7: Temporal behavior of the three servo loops running on the simulated embedded platform.



(a) Servo position.

(b) Servo position error.

Figure 8: Functional behavior of the three servo loops running on the simulated embedded platform. Task "Servo3" experiences deadline misses, resulting in unstable behavior.

The functional trace obtained using our approach is shown in Figure 8a. In this example, task "Servo 3" experiences deadline misses, the impact of which is clearly visible in the functional trace. Additionally, Figure 8b shows the error between these traces, and the reference traces. This shows that while these traces do not match exactly to the reference, they do match very closely, with the absolute error remaining below 2×10^{-5} for tasks "Servo 1" and "Servo 2". While the error is much larger for task "Servo 3" (within 200), its unstable behavior due to the deadline misses also causes the position to assume large values. Relative to these values, the observed error is also minimal.

5.2 Real-World Validation

One of the main goals of the presented approach is to avoid costly re-engineering by allowing us to evaluate the impact of the temporal behavior of the embedded platform on the functional behavior of applications earlier in the development process. As such, the presented approach needs to accurately predict the performance of an application in simulation compared to its real-world performance after deployment. As such, to further validate our approach, we make use of a second example case to validate exactly this aspect. As an example case, we consider a simple monitoring algorithm which estimates the velocity of an object based on its position. It simply does this by periodically sampling the position, using an Analog-to-Digital Converter (ADC), and calculating the discrete derivative of the position. This control algorithm is initially deployed as a single task running on an embedded system with period $250\mu s$. For the position input, we consider a sinusoidal signal with amplitude 1 and frequency 100Hz. Then, we introduce a second, interfering task,

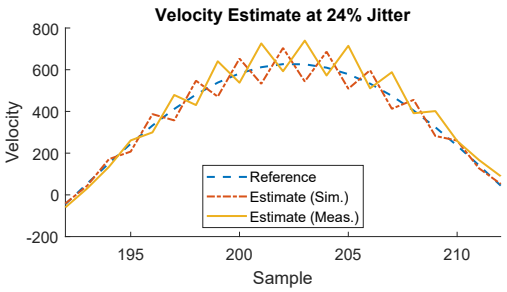


Figure 9: Actual versus estimated velocity at 24% jitter in simulation and on the real system.

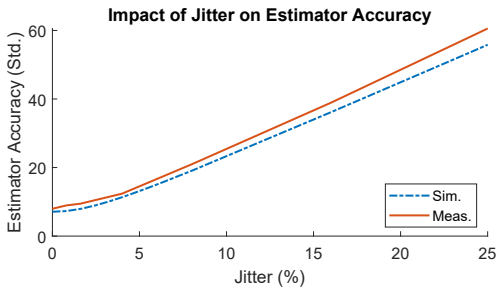


Figure 10: Effect of activation jitter on estimator accuracy.

with period $500\mu s$, and with a higher priority than the monitoring task. As such, every other activation, the monitoring task will need to wait for the interfering task, causing jitter, and causing the monitoring task to sample the position signal at the wrong point in time. This will degrade the accuracy of the estimated velocity, as shown in Figure 9, which we can observe.

We deploy this application first virtually, using our co-simulation approach as described in Section 4.4, in this case using an actual model of the ADC for the sensor model. We use this co-simulation setup to predict the accuracy of the velocity estimate for different amounts of jitter by varying the execution time of the interfering task. This experiment is then repeated on a real-world test setup, where the monitoring algorithm is deployed on a real embedded platform, together with an interfering task, of which we can vary the execution time. In this setup, the position signal is provided using a function generator. Results for this second use case are shown in Figure 10, which shows the accuracy of the estimated velocity for varying amounts of jitter, both predicted in simulation (Sim.) and measured on the real test setup (Meas.). While we see that the real performance is overall slightly worse than predicted, these results do look promising. Indeed, even with the simplified embedded platform models, the predicted curve matches the real curve rather well. Additionally, while we have tried to provide a representative model of the physical interface, i.e., the model of the ADC, which includes, among others, quantization noise, it's possible that additive measurement noise on the real system further affects the accuracy of this simple monitoring algorithm. This could provide an explanation for the overall observed difference in accuracy. However, further investigation is needed to validate this claim.

6 CONCLUSIONS AND FUTURE WORK

In this paper, we showed a co-simulation approach to simulate the embedded deployment of control and monitoring algorithms. This allowed us to evaluate and predict their functional behavior in simulation, taking into account the temporal behavior of the embedded platform. To achieve this, we first showed how Discrete EVent System specification (DEVS) models can be included in Functional Mock-up Units (FMUs), by leveraging features introduced in version 3.0 of the Functional Mock-up Interface (FMI) standard. These new features allowed us to adapt the semantics of DEVS to FMI, allowing us to include DEVS models of an embedded platform in a co-simulation environment. These DEVS FMUs capture the temporal behavior, while scheduled execution FMUs (also introduced in FMI 3.0) were used to include the functional behavior of the embedded application. Using this co-simulation setup, we were able to execute the embedded application in simulation as it would on an actual embedded platform. To validate this approach, we compared it both to another existing approach in simulation, and to real-world measurements with promising results.

We plan to extend this approach in future work. Mainly, this work showed a proof-of-concept of how FMI 3.0 can be used to include DEVS models of the embedded platform in co-simulation. As such, the embedded platform models used in the current paper were rather simplified, yet detailed enough to demonstrate the

approach. In future work, we plan to extend the approach with more detailed embedded platform models, including multicore or even heterogeneous platforms. Additionally, in the examples described in this paper, the embedded platform model was always included as a single FMU. However, in future work, we intend to extend this by including (DEVS) models of different parts of the embedded platform as different FMUs. This would allow for a more flexible and modular approach. Indeed, the embedded platform model could be built using different FMUs, instead of generating an FMU for a specific platform configuration. In this regard, instead of characterizing reads and writes using their expected execution time, the sensor and actuator models described in Section 4.4, could be replaced with corresponding DEVS models which capture this temporal behavior and signal the end of a read/write to the task model. This would also allow for easier reconfiguring of the embedded platform model based on changing design decisions. For example, replacing a model of a controller area network (CAN) bus, with one of a FlexRay bus. However, further work is needed to prove that coupling DEVS models through FMI in this way does not violate the DEVS formalism.

ACKNOWLEDGMENTS

This research was supported by Flanders Make, the strategic research center for the manufacturing industry, within the Flexible Multi-Domain Design for Mechatronic Systems (FlexMoSys) project.

REFERENCES

[1] G. Karsai and J. Sztipanovits, "Model-integrated development of cyber-physical systems," in *Software Technologies for Embedded and Ubiquitous Systems: 6th IFIP WG 10.2 International Workshop, SEUS 2008, Anacarpi, Capri Island, Italy, October 1-3, 2008 Proceedings 6*. Springer, 2008, pp. 46–54.

[2] C. M. Kirsch and R. Sengupta, "The evolution of real-time programming," in *Handbook of Real-Time and Embedded Systems*. Chapman and Hall/CRC, 2007, pp. 193–216.

[3] M. Morelli and M. Di Natale, "Control and scheduling co-design for a simulated quadcopter robot: A model-driven approach," in *International Conference on Simulation, Modeling, and Programming for Autonomous Robots*. Springer, 2014, pp. 49–61.

[4] D. Henriksson, A. Cervin, and K.-E. Årzén, "Truetime: Real-time control system simulation with matlab/simulink," in *Proceedings of the Nordic MATLAB Conference*. Copenhagen, Denmark, 2003.

[5] F. Cremona, M. Morelli, and M. Di Natale, "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*, 2015, pp. 1940–1947.

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

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

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

[9] C. Brandberg and M. Di Natale, "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)*. IEEE, 2018, pp. 1–10.

[10] Y. Vanommeslaeghe, B. Van Acker, K. Vanherpen, and P. De Meulenaere, "A co-simulation approach for the evaluation of multi-core embedded platforms in cyber-physical systems," in *Proceedings of the 2020 Summer Simulation Conference*, 2020, pp. 1–12.

[11] J. Denil, P. De Meulenaere, S. Demeyer, and H. Vangheluwe, "Devs for autosar-based system deployment modeling and simulation," *Simulation*, vol. 93, no. 6, pp. 489–513, 2017.

[12] S. Ravi, L. Beermann, O. Kotte, P. Pazzaglia, M. Vinnakota, D. Ziegenbein, and A. Hamann, "Timing-aware software-in-the-loop simulation of automotive applications with fmi 3.0," in *2023 ACM/IEEE 26th International Conference on Model Driven Engineering Languages and Systems (MODELS)*. IEEE, 2023, pp. 62–72.

[13] H. Vangheluwe, "The discrete event system specification (devs) formalism," *Course Notes, Course: Modeling and Simulation (COMP522A), McGill University, Montreal Canada*, vol. 13, 2001.

[14] B. P. Zeigler, *Multifacetted modelling and discrete event simulation*. Academic Press Professional, Inc., 1984.

[15] B. P. Zeigler and T. I. Oren, "Theory of modelling and simulation," *IEEE Transactions on Systems, Man, and Cybernetics*, vol. 9, no. 1, pp. 69–69, 1979.

[16] S. T. Hansen, C. Â. G. Gomes, M. Najafi, T. Sommer, M. Blesken, I. Zacharias, O. Kotte, P. R. Mai, K. Schuch, K. Wernersson *et al.*, "The fmi 3.0 standard interface for clocked and scheduled simulations," *Electronics*, vol. 11, no. 21, p. 3635, 2022.

[17] The Modelica Association. (2022) Functional mock-up interface specification. [Online]. Available: https://fmi-standard.org/docs/3.0/

[18] C. M. Legaard, D. Tola, T. Schranz, H. D. Macedo, and P. G. Larsen, "A universal mechanism for implementing functional mock-up units," in *11th International Conference on Simulation and Modeling Methodologies, Technologies and Applications*, ser. SIMULTECH 2021, Virtual Event, 2021.

[19] FMPy. [Online]. Available: https://fmpy.readthedocs.io/en/latest/

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

[21] C. Brandberg. (2018) Analysis of scheduling and memory access delays in multicores. [Online]. Available: https://www.mathworks.com/matlabcentral/fileexchange/66173-analysis-of-scheduling-and-memory-access-delays-in-multicores

AUTHOR BIOGRAPHIES

YON VANOMMESLAEGHE is a postdoctoral researcher at the University of Antwerp, Faculty of Applied Engineering in the Electronics and ICT department, Cosys-Lab, and is a co-worker at Flanders Make. His research interests include modeling and simulation for optimal (embedded) deployment in the context of cyber-physical systems. His email address is yon.vanommeslaeghe@uantwerpen.be.

BERT VAN ACKER is a postdoctoral researcher at the University of Antwerp, Faculty of Applied Engineering in the Electronics and ICT department, Cosys-Lab, and is a co-worker at Flanders Make. His research interests include modeling and simulation for optimal deployment in the robotics and automation domain. His email address is bert.vanacker@uantwerpen.be.

JOACHIM DENIL is an Associate Professor at the University of Antwerp, Faculty of Applied Engineering in the Electronics and ICT department, Cosys-Lab, and is associated with Flanders Make. His research interest include the design, verification and evolution of cyber-physical systems. His email address is joachim.denil@uantwerpen.be.

PAUL DE MEULENAERE is a Professor at the University of Antwerp, Faculty of Applied Engineering in the Electronics and ICT department, Cosys-Lab, and is associated with Flanders Make. His research interests include co-design and software deployment for cyber-physical systems. His email address is paul.demeulenaere@uantwerpen.be.