Explicit Semantic Adaptation of Hybrid Formalisms for FMI Co-Simulation

Joachim Denil, Bart Meyers
University of Antwerp
{Joachim.Denil|Bart.Meyers}
@uantwerpen.be

Paul De Meulenaere
University of Antwerp
Paul.Demeulenaere
@uantwerpen.be

Hans Vangheluwe
University of Antwerp
McGill University
hv@cs.mcgill.ca

ABSTRACT

With the advent of Software-Intensive and Cyber-Physical Systems, hybrid formalisms can be used to intuitively model the interactions of different models in different formalisms. Hybrid formalisms combine discrete (time/event) model constructs with continuous-time model constructs. These hybrid formalisms usually require a dedicated simulator. In this work we explicitly model the interfaces involved in the semantic adaptation of different formalisms and implement the execution using the Functional Mock-up Interface standard for co-simulation. The interfaces and co-simulation units are automatically generated using transformations. On the one hand, this allows tool builders to reuse the existing simulation tools without the need to create a new simulation kernel for the hybrid formalism. On the other hand, our approach supports the generation of different bus architectures to address different constraints, such as the use of hardware in the loop, the API of the legacy simulator, bus or processor load performance, and real-time constraints. We apply our approach to the modelling and (co-)simulation of an automotive power window.

Author Keywords

Functional Mock-up Interface, Co-simulation, MDE, Semantic Adaptation, Heterogeneous Modelling

ACM Classification Keywords

I.6.1 SIMULATION AND MODELING (e.g. Model Development). : Miscellaneous

INTRODUCTION

Model-based design (MBD) has become the de facto standard for the development of software-intensive and cyber-physical systems. MBD enables investigating a system at higher levels of abstraction using executable models. When designing complex engineered systems, different models have to be studied in concert [9]. Combining different modelling paradigms is primarily done for appropriateness [8]: certain behaviours, for example the laws of physics, are intuitively represented in a continuous-time formalism, while responding to events are more naturally described using a discrete-event formalism. Hybrid formalisms combine continuous-time and discrete time/event model constructs in a single model. Hybrid formalisms can thus be used to intuitively model these interactions.

Simulation of a hybrid formalism is often done using a dedicated simulator. However, when the simulators of the individual formalisms are readily available, co-simulation can be used. Co-simulation simulates a coupled problem by orchestrating different simulators with data-flow between them. The Functional Mock-up Interface (FMI) [1] is one such technique to orchestrate the different simulators. FMI is a standard that defines the interface for the simulation by coupled simulators (a simulator is a model combined with its simulation kernel), by model exchange or by co-simulation, using so-called Functional Mock-up Units (FMUs).

A challenge in using co-simulation with these heterogeneous formalisms is how to meaningfully combine them. Semantic aspects of the heterogeneous languages have to be adapted to talk to each other using the predetermined FMI API. A key point of this approach is that we should be able to compose heterogeneous models without modifying the underlying simulators. In this paper we combine discrete-event and continuous-time models using FMI co-simulation. We generate the appropriate orchestration and adaptation mechanism using model transformations. This allows tool builders to reuse existing simulation tools without the need to create a dedicated simulation kernel for each hybrid formalism. We introduce a number of composition architectures, from which the tool builder can choose from to achieve optimal results in terms of extra-functional requirements.

The rest of this paper is structured as follows: Section "The Functional Mock-up Interface" introduces FMI co-simulation. Section "Motivating Example" introduces our motivating example, the power window model. Section "Semantic Adaptation using FMI" discusses how heterogeneous models can be semantic adapted in the context of co-simulation. Section "Approach", generates the semantic adaptation for co-simulation using model transformations. In Section "Discussion" we discuss the generalities of this approach. Related work is described in Section "Related Work" and we conclude in Section "Conclusions".

THE FUNCTIONAL MOCK-UP INTERFACE

SpringSim'15, April 13-16, 2015, Alexandria, VA.

DOI string from ACM form confirmation

The functional mockup interface[1] is a standard for co-simulation and model exchange. The initial standard was developed within the context of the MODELISAR project. We introduce the co-simulation standard where a model and its simulator are exported together.

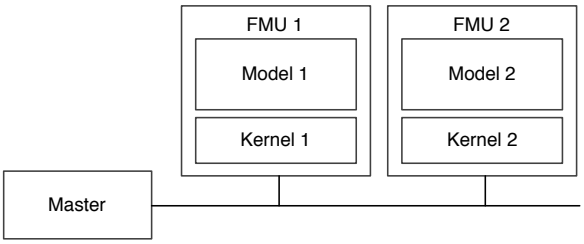

Figure 1. The FMI master-slave architecture.

The standard introduces an API to which the components, the functional mock-up units (FMU), in a FMI co-simulation must comply. A co-simulation model is a collection of connected FMUs, as shown in Figure 1. The FMUs are called slaves and consist of C-code that calls the solver of the contained simulator. Each slave has a description XML that describes the input and output variables of the slave. The XML file also contains constraints on the available functionality of the slave as well as the dependency relations between the output and the input ports of a slave FMU. We introduce some of the essential functions of the co-simulation API that allows communication between a master and FMU slaves during step-wise simulation:

- *fmi2DoStep*: The DoStep function advances the time in the co-simulation slave. The call has three arguments: (a) *currentCommunicationPoint*, is the current clock time of the master, (b) *communicationStepSize*, is the step size the solver has to compute, and (c) *noSetFMUStatePriorToCurrentPoint*, is a Boolean value that states whether the master will never call *fmi2SetFMUState* for a communication point prior than the *currentCommunicationPoint*. When the call returns *fmi2OK*, the step was computed successfully. If the slave returns *FMI2Discard*, the slave computed only part of the communication step successfully. If possible, the master algorithm should retry the communication step with a smaller size (This is done using the *fmi2SetFMUState*). When the slave is unable to compute the step, *fmi2Error* is returned.
- *fmi2GetStatus* and *fmi2GetXXXStatus*: This function allows the master algorithm to query the status of the slave. The 'XXX' is replaced by the data type of the status value. The information to retrieve is specified using the *statuskind* argument. For example, the *fmi2LastSuccessfulTime* information returns the last successful time of a slave algorithm after a *fmi2DoStep* with the return value *fmi2Discard*. The master algorithm can use this information to call the solver with a smaller time step.
- *fmi2GetXXX* and *fmi2SetXXX*: These functions allow the master algorithm to set and get variable values in the slave. The 'XXX' is replaced by the data type of the variable (for example, fmi2GetBoolean, allows the master to retrieve a Boolean value).
- *fmi2GetFMUState* and *fmi2SetFMUState*: These functions allow the master to save and restore the full state of a slave. It allows the master to rollback a communication step. The implementation of these functions are not mandatory in the slave. The rationale for the optional rollback implementation is to address legacy systems. An FMU can wrap a legacy simulator that does not allow any save and restore operations.

The master algorithm orchestrates the co-simulation. The FMI standard does not impose any restrictions on the master algorithm.

[1]Functional Mockup Interface: `http://www.fmi-standard.org`

MOTIVATING EXAMPLE

We use a power window case study to illustrate our work. The power window system is an automobile window that can be opened and closed by pressing a button or switch, as opposed to using a hand-turned crank handle. An increasing set of functions is being added to increase the comfort, safety and security of vehicle passengers. The power window system has all the essential complexity of a hybrid system [9, 11]. The power window system consists of an electromechanical part, the physical window, gears and motors, and a controller that receives commands from the user to raise or lower the window. When an object is present while closing the window, the motor should reverse the direction of the window and stop after one second.

Figure 2 shows a hybrid model of the power window system. The electromechanical components of the window are expressed using causal-block diagrams (CBD). CBDs are a general-purpose formalism used for modelling of causal, continuous-time systems. CBDs are commonly used in tools such as MathWorks Simulink®. CBDs use two basic entities: blocks and links. Blocks represent (signal) transfer functions, such as arithmetic operators, integrators or relational operators, or simply in- and outputs. Blocks can also have a notion of memory, so that e.g., an integrator operator is available. Links are used to represent the time-varying signals shared between connected blocks. The simulation of CBDs on digital computers requires a discrete-time approximation.

In Figure 2 three input signals are shown on the left and one output signal on the right. A power window controller block converts this input to a useful signal for a motor, and its physical acceleration. The window position is obtained by integrating the acceleration of the motor twice. However, when moving the window, friction can arise that depends on the velocity of the window.

The power window controller is modelled using a Statecharts variant. Statecharts represents modal behaviour and consists of states (circles), with one initial state, and transitions with events (conditions for the transition to fire, shown before the slash), and actions (actions that are executed after firing the transition, shown after the slash). This variant is hybrid; it does not purely contain events. Rather, it is possible to specify conditions and actions using simple arithmetic operators and external variables.

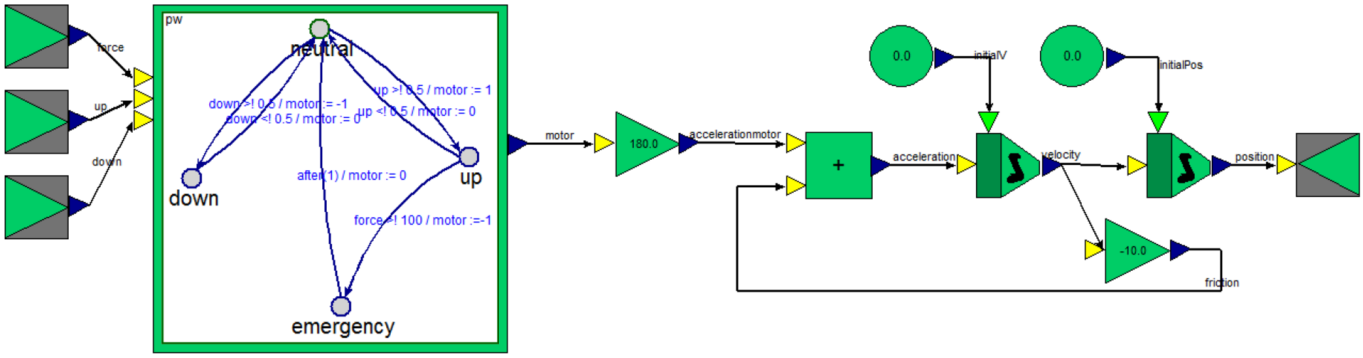

Figure 2. Hybrid model of the power window in AToM^3.

Four states are present in the model: (a) going up, (b) going down, (c) neutral and, (d) emergency. The specified events in the controller model are described over the continuous values of the input ports of the embedded controller model. For example, if the up button is pressed, the controller reacts by going from the neutral to the *going up* state. If the force acting on the window becomes bigger than 100 Newton, the controller changes to the emergency state. The event on this transition is specified using $force >! 100$, where $>!$ means crossing the threshold value of 100 from below. The controller also contains a timed transition to change from the emergency to the neutral state: $after(1)$, meaning that this transition is fired after staying for 1 second in the emergency state. The actions specify the signal values of the output port.

To co-simulate our hybrid model, we have access to two dedicated, legacy simulation tools. Our first simulation tool is able to simulate Causal-Block Diagrams. This tool is also FMI compliant, meaning that our tool is capable of generation an FMU from a CBD model [12]. Our second tool, SCC, is a Statechart compiler capable of generating stand-alone Python code from a Statechart specification [6]. The SCC tool is not FMI compliant. The Statechart compiler accepts a Statechart specification in the DES format. DES is a textual format to describe states, transitions and events.

SEMANTIC ADAPTATION USING FMI

The power window case study is co-simulated with two different tools: (a) a CBD simulator and, (b) a Statechart simulator. Because our Statechart tool only accepts pure events, we need to translate the incoming data from the CBD to the Statechart and vice versa. This adaptation between two different heterogeneous formalisms is an essential part of what is referred to as semantic adaptation.

Semantic adaptation focuses on adapting time, control and data between different heterogeneous models. Time is adapted because the clocks of different models have to be synchronised as clocks might run at different clock speeds, or like in our case, a timed formalism (CBD) has to be co-simulated with an untimed formalism (Statecharts). Control adaptation takes care of the scheduling of a computation step in the simulator, i.e., when is the *doStep*-function of a simulator called. Data adaptation is used to transform data values between different heterogeneous models, e.g., signal values to events.

More in detail, to simulate the power window case study using FMI co-simulation, the following adaptations are required:

- Our Statechart simulation kernel can only simulate 'pure' Statechart models. The hybrid control model needs to be transformed into a version containing only events instead of expressions over the input ports of the hybrid model.
- The expressions are extracted that are used as events in the controller. These expressions relate to the continuous-time behaviour of the CBD model which influences the behaviour of the discrete-event controller. Consequently, the *State Event Locator*, part of the semantic adaptation, has to take care of the adaptation of data and control by using state event detection and location. Firstly, this requires that the semantic adaptation *detects* when a threshold of one of the event expressions is exceeded (crosses the threshold from below, above or both). Secondly in certain cases, the time at which the threshold is crossed has to be *located* within a certain precision bound. The literature describes different numerical algorithms to detect the crossing of a threshold, for example the bisect method or regula falsi method [9]. Common for these algorithms is that the integration step of the proceeding model should be repeated multiple times with a smaller time-step. This requires, in our case, that the CBD FMU must allow for the state to be saved and restored using the *fmi2GetFMUState* and *fmi2SetFMUState* functions and are able to handle different time steps.
- When the control model has executed, pure events are outputted. Similar to the state-event locator, a *Transducer* transforms the events coming from the 'pure' control model to a signal value that can be understood by the CBD simulator.
- The hybrid control model also contains transitions that rely on the advancement of time, for example the $after(1)$ transition. Because the Statechart formalism does not have a notion of time, semantic adaptation should convert clock-based event transitions to 'pure' event-based control model.
- Finally, the separate simulators should also adhere to the API of the FMI standard. This is done using a wrapper that calls the equivalent function(s) of the simulator when a call to the wrapper is done.

The semantic adaptation between different heterogeneous models can result in different architectures, of which four are

shown in Figure 3. The optimal architecture has to be chosen depending on (but not limited to) the following constraints of the co-simulation:

Legacy API: The legacy API of a simulation tool has a large influence on the possibilities of a co-simulation. The legacy tool complies to the parts of the FMI standard for which its API is defined. For example, if a simulation tool does not have any roll-back facilities (i.e., *getState* and *setState* counterparts), a simple wrapper around this legacy tool will not support roll-back mechanisms.

Real-world components: In a normal co-simulation setup the master is responsible for the global virtual time of the simulation. However, introducing real-world components in our co-simulation setup might restrict simulation strategies. For example, if a mechanical component is introduced, the global clock has to abide to the wall clock time (real time).

Performance requirements: Often, extra-functional requirements are present that depend on the architecture. These might be limiting the number of communications between the master and slaves, or an opposing requirement, minimising the use of memory.

Re-usability: When the FMI architecture will be used in different implementation phases, it is often desirable that the architecture is highly modular, so that different simulators can be easily plugged in.

The four architectures presented in Figure 3 each address different architectural constraints, which are discussed in the remainder of this section. In each architecture, master is shown along with the FMUs, for which the ports are shown. The port shape is triangular for signal values, and circular for event values. The logical data flow is shown as arrows between FMU ports. We call this the "logical" data flow, because the actual data flow is only between the FMU and the master.

Adaptation in master (Figure 3 (a)): The master algorithm implements the state-event detection and location algorithms as well as the data adaptations. This has the advantage that when an event, with not enough precision is located, the master can discard the current time-step, restore the state of the CBD FMU and use a smaller time-step. Because the master is responsible for control of the slaves, the master decides, based on the outcome of the adaptation algorithm, whether control is given to the Statechart slave. The master is also responsible for the global time. The master can keep a global clock and post a new event on the clock so that the Statechart FMU is called when the $after(1)$ timer has expired. The other advantage is that the wrappers for both the simulation kernels are mostly straightforward. The disadvantage of this approach is that the master algorithm is not reusable and complicated, illustrated by the communication arrows between FMUs that go from signal ports to event ports and vice versa.

Wrapping the embedded model (Figure 3 (b)): The embedded model is wrapped completely by the semantic adaptation. The master algorithm only uses signals and is unaware that there is a need for semantic adaptation of the continuous-time signals to events. The master calls the FMU of the embedded model for each new step in the CBD model. If an event is detected, but without enough precision, the wrapper returns the *fmi2Discard* status. When the master queries the slave for the last successful time step, the event location mechanism returns the new time step. The master reacts to this by restarting the computation of the CBD model. This procedure iterates until the time step is accepted by the embedded model. The wrapper also needs to keep track of time and request a computation from the master algorithm when a timer expires. The procedure is similar to the event location. The proposed approach results in a lot of communication overhead between the master and the different slaves. From a re-usability point of view, the wrapper of the embedded model is not generic and needs to be regenerated. However, the master algorithm is straightforward and reusable and mechanical components can be plugged in instead of the plant or environment.

Adaptation in separate FMUs (Figure 3 (c)): Two additional FMUs, adhering to the FMI standard are created. The first FMU, the State Event Locator (SEL), detects and locates the events. Similar communication as in architecture (b) occurs between the master and the slaves for locating events. The SEL is also responsible for keeping the time of the embedded model. Because of this, the SEL has to execute two times per simulation step. One time to detect and locate the events, and a second time, after executing the embedded model, to activate and deactivate time-based events that are enabled in the Statechart. The Transducer (TD) is responsible for translating events to signal values. The wrappers for both the Statechart simulation kernel and CBD simulation kernel are reusable in all situations. The master algorithm is straightforward and reusable, and each FMU can be replaced with a different FMU or a real-world component.

Adaptation on the encapsulating model (Figure 3 (d)): The encapsulating model is adapted with the State Event Location and Transducer. When an event is located without enough precision, the step is rejected by the slave. However, the slave FMU can use the location mechanism internally to compute the correct time step for the master and provide it when queried for the last successful time. This results in less communication on the bus. It is however less straightforward to plug in different FMU's or real-world components.

APPROACH

In this section we show how the different components of the power window co-simulation are generated using model transformations. We show the implementation of the modular architecture, as shown in Figure 3 (c). The generation of the other architectures are similar but use slightly modified model-to-text transformations. The domain-specific language presented in this section is created using the MetaDepth tool [3]. The model-to-text transformations are created with the Epsilon Generation Language [13]. All models and transformations are available for download[2].

Overview

Figure 5 shows an overview of the process of creating an FMI co-simulation from a hybrid formalism. The process model

[2] `http://msdl.cs.mcgill.ca/people/joachim/fmi_hybrid`

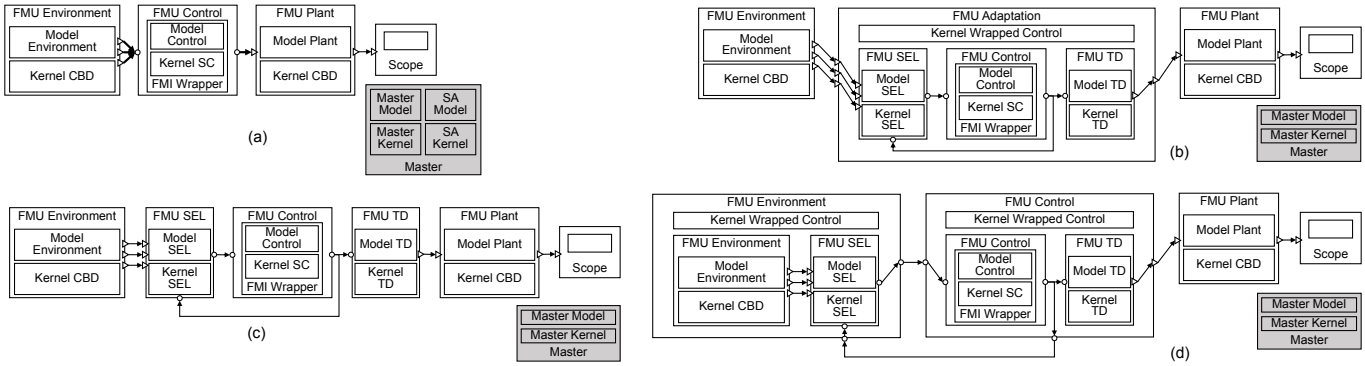

Figure 3. Four architectures for implementing the heterogeneous model.

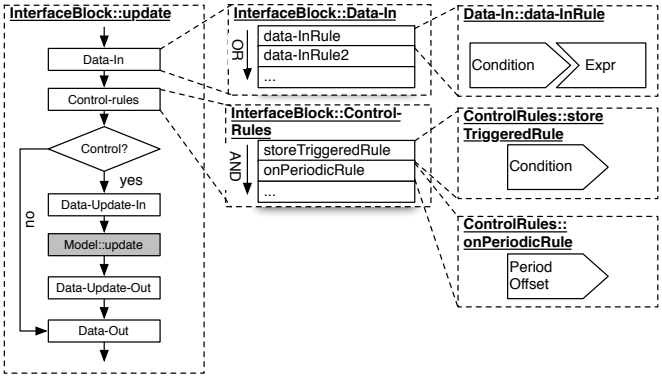

Figure 4. Execution of semantic adaptation models, from [7].

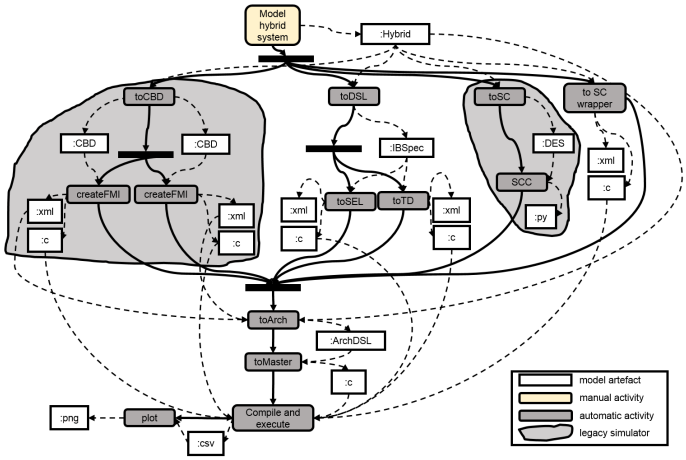

Figure 5. Activity Diagram model for generating the FMI code according to option (c) of Figure 3.

is expressed using the UML 2.0 Activities formalism [14]. Four distinct phases are present in the process model: (1) the generation of the SEL and TD FMUs, (2) the generation of the Statechart FMU, (3) the generation of the CBD FMUs and, (4) the generation of Master algorithm. In the following subsections we discuss the different phases.

We analysed the concept of semantic adaptation between different heterogeneous models to factor out the essentials of the adaptation of time, control and data in a domain specific language for modelling so-called *interface models* [7]. We extended the DSL in [7] so that adaptations between actual models, rather than domains, can be modelled in the DSL. The DSL models adaptations between heterogeneous modelling domains using a set of rules and executes these rules in five different steps, as shown in Figure 4. The DSL also contains relations between the different clocks in the heterogeneous models for the adaptation of time. The design choice for the interface model is that it is an hierarchical composition: an embedded model is wrapped by an outer model. This way, both directions of semantic adaptation are defined. However, all rules are optional, so it does not restrict the user to model non-hierarchical hybrid models. The five different steps as shown in Figure 4 are as follows:

- Data-in: When an interface model receives control, data-in rules are evaluated to update the buffers of the interface model. Operation can be defined to transform the data.
- Control: Control rules define if the embedded model should be executed at the current time. The control rules can schedule an event on the global clock to receive control at a later instant.
- Data-update-in: If control is given to the embedded model, data-update-in rules are executed to supply correct values to the embedded model.
- Data-update-out: Once the embedded model has completed its update procedure, the output buffers of the interface model are updated by the data-update-out rules.
- Data-out: Finally, the data-out rules are responsible for producing the output of the interface model from the data available in the output buffers, even if control was not given to the embedded model.

Generation of SEL and TD FMUs

The DSL model for the power window in Figure 2 that is generated by the *toDSL* transformation is shown in Listing 1. In the hybrid formalism, the crossing events, after events and actions on transitions represent the interface between the heterogeneous formalisms. For each transition in the hybrid formalism, we create a data-in rule to detect the event. In case of multiple orthogonal components in the hybrid Statechart, the orthogonal components are first flattened by generating all possible state combinations. For the hybrid model in Figure 2 six data-in rules are created from the original hybrid model, plus an additional one that produces a NULL-event if none of the data-in rules was applicable. The rule order in the DSL is important and defines which event is detected based on the input ports. These rules consists of a condition (depending on a signal value or on a timer value), and an action under which the condition is applied (producing some event and adding it to the interface model's buffer).

Listing 1. Semantic Adaptation model

```
IB {
  IMPLEMENTS "CBD_SC_InterfaceBlock_parsed"
  TIMED CLOCK ibClock
  EXTERNAL MODEL cbd ("CBDSim" WITH TIMED
            CLOCK cbdClock)
  INTERNAL MODEL sc ("SCSim" WITH UNTIMED
            CLOCK scClock)
  PINS:
    IN ["up" "down" "force"]
        OUT ["motor"]
  RULES:
    IN
      "force" >! ("100") -> TOSTORE("in_event0");
      "up" >! ("0.5") -> TOSTORE("in_event1");
      "up" <! ("0.5") -> TOSTORE("in_event2");
      "down" >! ("0.5") -> TOSTORE("in_event3");
      "down" <! ("0.5") -> TOSTORE("in_event4");
      AFTER ("1") -> TOSTORE("in_event5");
      ALWAYS -> TOSTORE(null);
    CONTROL
      STORETRIGGERED FROMSTORE != null
    UPDATEIN
      ALWAYS -> FORWARD FROMSTORE TO SUCCESSORS;
    UPDATEOUT
      ON DATA: "out_event0" -> TOSTORE(("1"));
      ON DATA: "out_event1" -> TOSTORE(("-1"));
      ON DATA: "out_event2" -> TOSTORE(("0"));
    OUT
      ALWAYS -> FORWARD FROMSTORE TO SUCCESSORS;
  TAG RELATION cbdClock = ibClock
}
```

Listing 2. Data rules in the SEL

```
...
if (fi->b[_em_has_run] == fmi2False){
 // SET THE NEW TIME OF SEL
 fi->currentTime = commStepSize + currentCommPoint;
 //RULE1:
 if (__crosses_below(fi->r[_prev_force],
         fi->r[_force], 100))
 fi->s[_out] = _events_out[_in_event0];
 //RULE2:
 ...
```

The *DATA-IN* rule section of the interface model in Listing 1 represents the State Event Locator and is transformed to an *if-else* ladder structure in C-code, as shown in Listing 2. As already stated, in this architecture, the State Event Locator has to be executed two times every time step because of the after-events. The first execution detects and locates the events, and the second execution starts and stops the timers needed to trigger time-based events in the Statechart. Therefore, a Boolean value *_em_has_run* has to be set by the master after the execution of the embedded model and reset after the execution of the global time step.

Similar code for the Transducer is generated by a model-to-text transformation generates based on the *DATA-UPDATE-OUT* rules. Because there are no further rules in the *DATA-OUT* section, a zero-order hold semantics is created for the output of the Transducer.

Generation of the Statechart

The transformation *toSC* generates a DES model from the hybrid model containing pure events. A DES model is a textual representation of a Statechart. The transformation is similar to the generation of the interface model. The DES file format can be read by the SCC compiler to create a standalone Python simulator from this specification.

To execute this Python simulator in a FMU context, a wrapper is required. This wrapper is different depending on the hybrid model, as the defined inputs and outputs of the FMU are taken into account. Once written for this Python simulator, the wrapper can be generated, as all necessary information can be extracted from the hybrid model. This wrapper is also shown in the architectures presented in Figure 3.

Generation of the CBD FMUs

To generate both FMUs for the CBD models, we export the CBDs to the format of the CBD legacy tool and generate the FMUs using the FMI-compliant tool [12]. This legacy CBD tool does not support rolling back a time step, as the *setState* and *getState* FMI functions are not available.

Generation of Master algorithm

The master algorithm is responsible for orchestrating the co-simulation by communicating with the slaves in the correct order. The model transformation creates all necessary code to instantiate, initialise, run, advance time, get and set values of the different FMUs and finalise the co-simulation. The template code we used to create the model-to-text transformation is based on the master algorithm of the FMU SDK created by QTronic[3].

Based on the ports between the different models, data exchange variables and reference values need to be created. The data exchange variables are arrays of pointers to pass data values of one FMU to another FMU via the master. The reference values are used to locate the correct buffer for the variable in the FMU. Because our legacy tool creates the two CBD FMUs, we have no control over the placement of the variables in the memory of the FMU. Therefore, we use the created *modeldescription* XML files to create the correct value reference variables in the master algorithm.

The master is also responsible for the adaptation of control in this architecture. The master can hand out control by calling the FMU with the *doStep* method. As already described in Section Semantic Adaptation using FMI, the return status from the *doStep* function is used to accept or reject a step. Finally, in the power window interface specification control has to be adapted so that the Statechart is only activated when the State Event Locator located an event: the return value of the State Event Locator is not NULL. A part of the generated code for the control adaptation is shown in Listing 3.

Results

Finally, the created FMUs and master algorithm are simulated together. Our experiment simulates the hybrid model for 10 virtual seconds. The environment model simulates a user that presses the up button for six seconds, releases the button for one second and then pushes the button again. However, at four seconds, an object is detected between the window and the frame. Figure 6 shows the the results of our co-simulation. The upper three plots show the values of the environmental model. The lowest plot shows the position of the window (i.e., the result of the simulation).

[3] QTronic website: **http://www.qtronic.de/en/fmusdk.html**

Listing 3. Master algorithm excerpt

```
...
/* CONTROL */
if (val_out_SEL_TO_SC[0] != NULL){
 fmu_sc.setString(c_sc, vr_in_SEL_TO_SC, 1,
         val_out_SEL_TO_SC);
 flag = fmu_sc.doStep(c_sc, currentTime, STEP_SIZE,
         fmi2True);
 if (flag != fmi2OK){
     /* Exception Handling */
 }
 fmu_sc.getString(c_sc, vr_out_SC_TO_SEL, 1,
         val_out_SC_TO_SEL);
 fmu_sc.getString(c_sc, vr_out_SC_TO_TD, 1,
         val_out_SC_TO_TD);
...
```

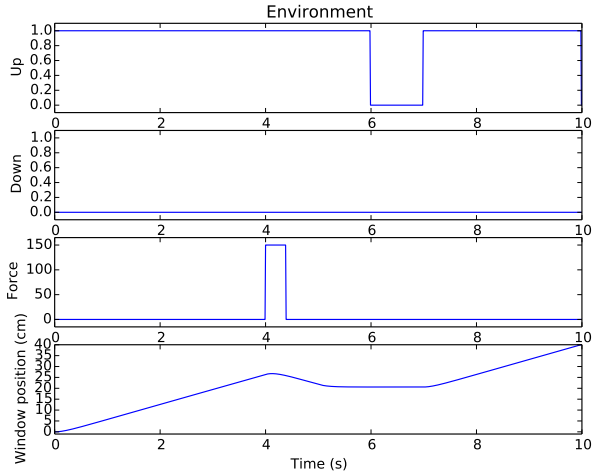

Figure 6. Results of the FMI co-simulation of the powerwindow

As expected the window starts going up immediately when the simulation starts. At four seconds the object is detected and the window goes down for one second. The window stops completely until the up button is re-pressed at seven seconds.

DISCUSSION

In our hybrid model, there are no feedback loops between the CBD model before the Statechart and after the Statechart. However, in reality, the force on the window is not only dependent on the object between the window and the frame but also on the position of the window. This creates a feedback loop from the plant model (after the Statechart) to the environment model (before the Statechart). When a feedback loop is present, we split the CBD model in two separate models at a block that does not have direct feed-through semantics. Valid locations for splitting up the model are integrator, derivative and delay blocks: they only depend on the computation of the previous time step. However, this implies that the output of the plant CBD model must be available before running the environment model. The master orchestrates this situation by saving the state of the plant model, extracting the delayed value and rolling back the state of the plant FMU before starting the time step. More information can be found in [16].

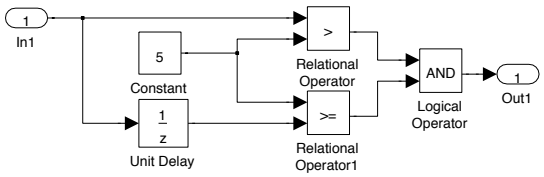

Figure 7. State-event location within the CBD formalism

Semantic adaptation is not strictly enforceable by creating interface blocks in the FMI co-simulation (FMU or master). In certain formalisms it is possible to use the language of the formalism to implement the required adaptations. For example in the CBD language, state event location can be done by adding a pattern of blocks to the signal where event detection has to occur. Figure 7 shows such a CBD pattern to detect a crossing from below of the threshold value 5. Furthermore, when the execution semantics are explicitly modelled, as proposed in [10], the event location and roll-back mechanisms can be implemented in the model as well. This could make semantic adaptation, in the case of event location with precision, trivial. However, more research on this topic is required, and this does not address the reuse of legacy simulators.

In certain cases, the legacy API has a mismatch between certain elements. For example, Feldman et al. report such a problem where the simulator time is expressed using an integer value in milliseconds while the FMI standard uses a floating point value for this purpose [5]. The user needs to decide on such semantic adaptations.

In this paper we focus on a single instance of semantic adaptation between a causal-block diagram and a Statechart. However, we believe that the presented technique is more general and can be applied to a broad spectrum of hybrid formalisms. By explicitly modelling the required semantic adaptation we can create model-to-text transformations to operationalise other hybrid formalisms. Our language to model semantic adaptation is rule-based and can be extended to include features needed for other adaptations.

RELATED WORK

Many hybrid simulation approaches already exist in academia and industry. For example, the simulation of Stateflow® and Simulink® by the MathWorks®. In academia other heterogeneous modelling and simulation approaches are available. Modhel'X[2] and Ptolemy[4] are component-oriented multi-formalism approaches. They consider that the semantics of a modelling language is given by its Model of Computation (MoC). A MoC is a set of rules defining the relations between the elements of a model, the operational semantics. The meta-model is similar for all languages but the semantics are given by a corresponding MoC, thus defining different behaviours. The heterogeneous models have a hierarchical organisation, each with their own MoC. At the boundaries between the different MoCs combinations can occur. In the Ptolemy approach this boundary is fixed and coded statically in the kernel of the tool. ModHel'X on the other hand allows the explicit specification of the boundary. The problems of combining these different heterogeneous MoCs is similar to the semantic adaptation we presented in this work, although we address the adaptation at the formalism and model level rather than only at MoC level.

Feldman et al. present a technique to generate FMUs from Rhapsody Statecharts [5]. Similar to our case study, they provide guidelines how to integrate these heterogeneous components in an FMI co-simulation setup. By explicitly modelling these interactions, our work is able to generate the needed semantic adaptation for different circumstances.

Tripakis and Broman discuss how to create different FMUs from heterogeneous modelling formalisms [15]. They formally define how to encode an FMU and which adaptations are required for certain classes of formalisms, for example timed and untimed formalisms. This work is seen as complimentary to our own. We provide a method to explicitly model the interactions between heterogeneous modelling languages and to automatically create FMU from these models. In this work we do not enforce the wrapping of the semantic adaptation to the FMU but rather we allow different architectural choices based on the problem at hand.

CONCLUSIONS

Meaningfully combining different heterogeneous modelling languages is a huge challenge. Hybrid formalisms combine modelling elements from different heterogeneous languages to intuitively model certain systems. However, these new languages typically require their own simulation kernel. In this paper, we presented an approach to operationalise hybrid modelling languages by reusing the available simulators of the reused formalisms. Our approach uses the FMI standard for co-simulation to orchestrate the different simulators. The semantic adaptation between the heterogeneous formalisms is explicitly modelled. Model-to-text transformations are responsible for the generation of the different artefacts needed in the co-simulation. To achieve optimal results in terms of extra-functional requirements, a choice of architectures are available. We believe that our presented technique is useable for a broad class of heterogeneous and hybrid modelling environments.

In the future we want to integrate more languages into our hybrid co-simulation tool to show that our approach is generic. In this paper we hinted at the relation between different architectures and the different extra-functional requirements of a co-simulation. In the future, we want to investigate this relation and provide the users with guidelines to choose the most appropriate architecture for the problem at hand.

REFERENCES

1. Blochwitz, T., Otter, M., Åkesson, J., Arnold, M., Clauss, C., Elmqvist, H., Friedrich, M., Junghanns, A., Mauss, J., Neumerkel, D., et al. Functional mockup interface 2.0: The standard for tool independent exchange of simulation models. In *9th International Modelica Conference* (2012).
2. Boulanger, F., and Hardebolle, C. Simulation of Multi-Formalism Models with ModHel'X. *2008 International Conference on Software Testing, Verification, and Validation* (Apr. 2008), 318–327.
3. de Lara, J., and Guerra, E. Deep meta-modelling with metadepth. In *Objects, Models, Components, Patterns*, J. Vitek, Ed., vol. 6141 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 2010, 1–20.
4. Eker, J., Janneck, J., Lee, E., Ludvig, J., Neuendorffer, S., and Sachs, S. Taming heterogeneity - the Ptolemy approach. *Proceedings of the IEEE 91*, 1 (Jan. 2003), 127–144.
5. Feldman, Y. A., Greenberg, L., and Palachi, E. Simulating Rhapsody SysML Blocks in Hybrid Models with FMI. In *Proceedings of the 10th International Modelica Conference* (Mar. 2014), 43–52.
6. Feng, T. H. An extended semantics for a Statechart Virtual Machine. In *Summer Computer Simulation Conference (SCSC 2003)* (2003).
7. Meyers, B., Denil, J., Boulanger, F., Hardebolle, C., Jacquet, C., and Vangheluwe, H. A dsl for explicit semantic adaptation. In *Proceedings of the 7th workshop on Multi-Paradigm Modelling* (2013), 47–56.
8. Mosterman, P., and Vangheluwe, H. Computer automated multi-paradigm modeling: An introduction. *Simulation 80*, 9 (Sept. 2004), 433.
9. Mosterman, P. J. Hybrid Dynamic Systems: Modeling and Execution. In *Handbook of Dynamic Systems Modeling*. 2002.
10. Mosterman, P. J., and Zander, J. Advancing Model-Based Design by Modeling Approximations of Computational Semantics. In *4th International Workshop on Equation-Based Object-Oriented Modeling Languages and Tools* (2011), 3–7.
11. Prabhu, S., and Mosterman, P. Model-Based Design of a Power Window System: Modeling, Simulation and Validation, 2004.
12. Pussig, B., Denil, J., De Meulenaere, P., and Vangheluwe, H. Generation of functional mock-up units for co-simulation from simulink®, using explicit computational semantics: Work in progress paper. In *Proceedings of the Symposium on Theory of Modeling & Simulation - DEVS Integrative*, DEVS '14, Society for Computer Simulation International (San Diego, CA, USA, 2014), 38:1–38:6.
13. Rose, L. M., Paige, R. F., Kolovos, D. S., and Polack, F. A. The epsilon generation language. In *Model Driven Architecture–Foundations and Applications*, Springer (2008), 1–16.
14. Rumbaugh, J., Jacobson, I., and Booch, G. *Unified Modeling Language Reference Manual, The*. Pearson Higher Education, 2004.
15. Tripakis, S., and Broman, D. Bridging the Semantic Gap Between Heterogeneous Modeling Formalisms and FMI (UCB/EECS-2014-30). Tech. rep., University of California at Berkeley, 2014.
16. Van Acker, B., Denil, J., De Meulenaere, P., and Vangheluwe, H. Generation of an optimised master algorithm for fmi co-simulation. Tech. rep., Universiteit Antwerpen, 2014.