Co-Simulation of Hybrid Systems with SpaceEx and Uppaal

Sergiy Bogomolov[1] Marius Greitschus[2] Peter G. Jensen[3] Kim G. Larsen[3]
Marius Mikučionis[3] Andreas Podelski[2] Thomas Strump[2] Stavros Tripakis[4]

[1]IST Austria, Austria
[2]University of Freiburg, Germany
[3]Aalborg University, Denmark
[4]Aalto University, Finland, and University of California, Berkeley, USA

Abstract

The Functional Mock-up Interface (FMI) is an industry standard which enables co-simulation of complex heterogeneous systems using multiple simulation engines. In this paper, we show how to use FMI in order to co-simulate hybrid systems modeled in the model checkers SPACEEX and UPPAAL. We show how FMI components can be automatically generated from SPACEEX and UPPAAL models. We also validate the co-simulation approach by comparing the simulations of a room heating benchmark in two cases: first, when a single model is simulated in SPACEEX; and second, when the model is split in two submodels, and co-simulated using SPACEEX and UPPAAL. Finally, we perform a measurement experiment on a composite model to show a potential for statistical model checking using stochastic co-simulations.
Keywords: FMI, hybrid system, timed automaton

1 Introduction

Despite advances in model checking and other formal verification techniques, simulation remains the workhorse for system analysis. A plethora of simulation tools are available today, from the academia as well as from the industry. These tools support a large variety of modeling languages, targeted at different types of systems from various disciplines (e.g., mechanical, electrical, digital, continuous or discrete, or mixes thereof). Unfortunately, these tools can rarely interoperate. This is a problem, because modern cyber-physical systems are highly complex and multidisciplinary, requiring specialized modeling languages and tools from several domains.

The *Functional Mock-up Interface*[1] (FMI) is a standard developed to address this problem. components and a C API that these components must implement. The components are called *functional mock-up units*, or FMUs. An FMU is typically generated automatically *(exported)* from some simulation tool, and corresponds to a (sub-)model designed in that tool. The submodels/FMUs are then *imported* into a *host* simulator. The host commands the simulation by calling the API methods of the FMUs, thus effectively achieving integration of the original simulation environments. FMI supports two integration modes: (a) *model exchange*, where the host simulator is handling the numerical integration; and (b) *co-simulation*, where each FMU implements its own numerical integration mechanism (or any other internal mechanism to advance its state in time). Because each mode imposes its own requirements on the FMUs (for instance, in model exchange, the FMUs must provide to the host information such as state derivatives, which are not necessary for co-simulation) the FMI APIs for the two modes are different.

In this work we use FMI in order to connect two state-of-the-art modeling and verification tools for cyber-physical systems: SPACEEX [15] and UPPAAL [16]. SPACEEX is a tool for modeling and verifying *hybrid systems* [3]. UPPAAL is primarily a model-checker for *timed automata* [2], however, it also supports statistical model-checking of hybrid systems [10].

Our goal is to integrate these two tools for co-simulation. That is, we want to be able to: (a) build a sub-model of the system (e.g., the model of the *plant* under control) in SPACEEX; (b) build another sub-model (e.g., the *controller*) in UPPAAL; (c) automatically generate an FMU for each sub-model; (d) import the FMUs, connect and co-simulate them in a host environment.

The motivations for connecting SPACEEX and UPPAAL in this manner are numerous. First, although both SPACEEX and UPPAAL support simulation of hybrid systems, each tool offers its own modeling language, which is not compatible with that of the other tool. Translating from one language to the other is limited to common features supported by the tools. For example, even though frameworks CIF [1, 6] and HSIF [18] solve the complexity problem of one format translation to another by performing at most two translations, the approach still suffers from the fact that UPPAAL features

[1]See `https://www.fmi-standard.org/` for more details.

like committed locations and C-like function code are not supported in SPACEEX and UPPAAL has limited support for ODEs. Moreover, by using co-simulation, we are able to take advantage not just of the specific strengths of languages of each tool, but also of their native simulation engines, since each FMU is internally running essentially a "copy" of the simulation algorithm of the original tool.

As host environment we use the tool Ptolemy[2]. Ptolemy is a modeling and simulation environment for heterogenous systems [12]. Recently, support has been implemented in Ptolemy for using it as a host environment for co-simulation based on FMI. FMUs (developed by other tools) can be imported into Ptolemy, connected using Ptolemy's graphical user interface, and co-simulated using an implementation of the co-simulation algorithm described in [8]. This algorithm has desirable properties, such as *determinacy*, namely, the fact that the results of the simulation are independent of arbitrary factors such as names of the FMUs, order of creation, or order of evaluation in the diagram.

The contributions of this paper are the following:

1. We show how FMUs can be generated automatically from models of hybrid and timed automata built in SPACEEX and UPPAAL. There are several subtleties involved in this, as hybrid and timed automata are models designed primarily with verification in mind, whereas FMI is designed for simulation and therefore imposes certain properties on FMUs, such as determinism.

2. We report on the implementation and case studies. In particular, we apply our co-simulation framework to a room heating benchmark [13].

3. We validate the co-simulation algorithm proposed in [8] by comparing the results of the case study in two settings: (a) when the case study is modeled and simulated in a single tool, and (b) when the various components of the case study are modeled in two tools and co-simulated using our framework. We show that our co-simulation framework computes the same simulation trajectories as the setting (b) provided that the maximum simulation step size of co-simulation is sufficiently small.

4. We demonstrate how stochastic simulations can be included into the composite model with hybrid systems and applied a simple statistical measurement to show the potential for statistical model checking using FMI co-simulations.

The rest of the paper is organized as follows. In Sec. 2, we introduce the necessary background on FMI for this work. Afterwards, we present our translation of SPACEEX and UPPAAL models into FMUs in Sec. 3. This is followed by the case study in Sec. 4. We discuss related work in Sec. 5. Finally, we conclude the paper in Sec. 6.

[2]See `http://ptolemy.eecs.berkeley.edu/`.

2 Background on FMI

Conceptually an FMU can be seen as a (timed) state machine. This machine has a set of input variables (or *ports*), a set of output variables, and a set of internal states. The machine interacts with its environment only by means of a clearly defined set of *interface methods*. These methods are specified in the FMI standard. For the purposes of this paper, and following the formalization presented in [8], the key interface methods of FMI (for co-simulation) are:

- A method `init` to initialize the state of the FMU. If S is the set of states of the FMU, then $\texttt{init} \in S$.

- A method `set` to set a given input variable to a certain value. The signature of `set` is $\texttt{set} : S \times U \times \mathbb{V} \to S$, where U is the set of input variables of the FMU, and $\mathbb{V}$ is the set of all possible values (for simplicity we ignore typing and use a single universe $\mathbb{V}$ of values for all variables). Given state s, input variable $u \in U$, and value $v \in \mathbb{V}$, $\texttt{set}(s,u,v)$ returns the new state obtained after setting u to v.

- A method `get` which returns the value of a given output variable. Its signature is $\texttt{get} : S \times Y \to \mathbb{V}$, where Y is the set of output variables of the FMU. Given state s and output variable $y \in Y$, $\texttt{get}(s,y)$ returns the value of y in s.

- A method `doStep` which advances the state of the machine in time. Its signature is $\texttt{doStep} : S \times \mathbb{R}_{\geq 0} \to S \times \mathbb{R}_{\geq 0}$, where $\mathbb{R}_{\geq 0}$ is the set of non-negative real numbers. The behavior of `doStep` is explained below.

As said above, an FMU is essentially a state machine (of type Moore or Mealy): the `get` method corresponds to the output function of the machine, while the `doStep` method corresponds to the transition function. The difference is that `doStep` takes as input a *time step* $h \in \mathbb{R}_{\geq 0}$: in that sense, an FMU is a timed state machine.

The behavior of `doStep` is as follows. Given state $s \in S$, and time step $h \in \mathbb{R}_{\geq 0}$, a call to $\texttt{doStep}(s,h)$ is interpreted as the co-simulation algorithm "asking" the FMU to perform a simulation step of length h. For a number of reasons, including numerical integration issues, the FMU may "accept" or "reject" this request. If it rejects, it means that it was not able to advance time by h (but may have been able to advance time by a smaller delay $h' < h$). Formally, $\texttt{doStep}(s,h)$ returns a pair (s',h') where $s' \in S$ is a state and $h' \in \mathbb{R}_{\geq 0}$ is a time step, such that:

- either $h' = h$, which is interpreted as F having *accepted* h, and having moved to a new state s';
- or $0 \leq h' < h$, which is interpreted as F having *rejected* h, but having made partial progress up to h', and having reached a new state s'.

It is worth noting that FMUs are *deterministic* machines, in the sense that for a given sequence of inputs (i.e., a sequence of input values and time steps), the sequence of states and outputs that the machine produces is unique. This is because there is a unique initial state $\texttt{init} \in S$, and $\texttt{set}, \texttt{get}, \texttt{doStep}$ are all *total functions*. Moreover, the fact that these functions are total implies that the machine is able to accept any input at any time, therefore, it is implicitly *input-enabled*.

In addition to the above, each FMU comes with a set of *input-output dependencies*, $D \subseteq U \times Y$. D specifies for each output variable which input variables it depends upon (if any): $(u,y) \in D$ means that output variable y depends on input variable u. This information is used to ensure that a network of FMUs has no cyclic dependencies, and also to determine the order in which all network values are computed during a simulation step [8].

FMI specifies the methods that every FMU must implement, but it does *not* specify the co-simulation algorithm (also called a *master algorithm*). In fact, devising such an algorithm with good properties is not a trivial problem, and has been the topic of previous work [8]. In that work, two co-simulation algorithms have been proposed and proved to have desirable properties, such as termination of a simulation step, and *determinacy*. The determinacy property says that the results of a simulation do not depend on the order in which the algorithm chooses to call doStep on a set of FMUs. This allows to ensure that the simulation results are well-defined and are not influenced by arbitrary factors such as FMU names, order of creation, geometrical position in the diagram of a graphical model, etc., as is often the case with simulation tools.

In a nutshell, the co-simulation method proposed in [8] relies on the following principle. First, the co-simulation algorithm chooses a default time step, $h_{\max}$, called the *maximum step size*. Second, the algorithm saves the state of each FMU in the model (FMI specifies methods for an FMU to export and import its state, although these are optional). Assuming there are n FMUs, $F_1, ..., F_n$, the algorithm maintains n states, $s_1, ..., s_n$. Third, the algorithm calls $F_i.\texttt{doStep}(s_i, h_{\max})$ on each FMU F_i, and collects the returned time steps $h'_1, ..., h'_n$. There are two cases: either all FMUs accepted the proposed time step, i.e., $h'_1 = h'_2 = \cdots = h'_n = h_{\max}$, in which case this simulation step is over, and the algorithm proceeds to the next; or at least one FMU F_i rejected $h_{\max}$, i.e., $h'_i < h_{\max}$ for some i. In the latter case, the algorithm computes the minimum of $h'_1, ..., h'_n$, $h_{\min} = \min\{h'_1, ..., h'_n\}$, restores the saved state of each FMU, and tries again with new step size $h_{\min}$.

Assuming that the FMUs satisfy the reasonable "monotonicity" property that if they were able to advance time by h'_i then they are also able to advance time by any smaller step, and by the fact that $h_{\min}$ is smaller than all h'_i, the second attempt is guaranteed to succeed. That is, $h_{\min}$ will be accepted by all FMUs. As a result, at most after two attempts, a co-simulation step is successful, and the algorithm proceeds with the next step, repeating the same procedure as above.

The FMI standard sets out a framework where FMUs share the notion of time and exchange the variable values via input-output ports: outputs from one FMU are mapped as inputs to other FMU(s) and so on. The output port values are said to be owned and controlled by the emitting FMU, wheres the inputs are computed and provided by another (outputting) FMU. The framework foresees that before producing an output an FMU may first need some input values and thus input-output dependency information is introduced. Overall the I/O port connectivity graph derived from the model of interconnected FMUs, together with the local I/O dependencies of each individual FMU, result in a global I/O dependency graph for the entire model [8].

Time and I/O values are synchronized by the co-simulation algorithm: the time is agreed by repeatedly consulting each FMU and the I/O values are propagated according to the dependencies. The co-simulation algorithm assumes that each FMU provides a static dependency list of its ports before simulation starts, and that the resulting global I/O dependency graph is acyclic, and therefore there exists a schedule for computing the value of every input port before the value of a dependent output port is requested [8].

3 Translating Models into FMUs

The behavior of individual FMUs is provided by model-checker's simulation engines based on the guidelines described in [20]. In particular the report distinguishes models with continuous dynamics and discrete dynamics. The continuous behavior is modeled by differential equations over continuous variables whose values can be shared among FMUs by the means of port connections. The output ports can be mapped to the owned/controlled variables which are read and written to, whereas the input ports are mapped to variables which only participate in constraints and their values are read-only by that FMU.

The discrete behavior is modeled by discrete transitions in the automata control flow structure. The discrete transitions are designed to be executed with micro-steps of zero delay. The transitions can also be decorated with event labels and each tool supports its own kind(s) of synchronizing compositions internally and therefore the discrete transition synchronization is also handled individually within the tools. Report [20] provides means of discrete transition synchronization by allocating two special port variables: one for incoming (input) synchronization and one for outgoing (output) synchronization. The domain of discrete input (output) port coincides with the set of input (output resp.) labels plus special value *absent* which denotes no synchronization or internal discrete transition.

3.1 Uppaal

UPPAAL uses timed automata models [2] extended with discrete variables over structured types to describe a behavior of a timed system. In timed automata the continuous dynamics is controlled by real-valued clock variables (with derivatives set to one) and discrete state complemented with integer variables – both of whom are candidates for exchange via FMU input-output ports. Statistical model checking (SMC) extensions [10, 11] allow a finer control of the clock derivatives by means of ordinary differential equations, moreover the discrete transitions are stochastic where the execution is determinized by probability distributions over time and over branching edges. The stochastic semantics of a parallel composition is similar to FMI co-simulation algorithm [8] in a way the minimum delay is negotiated and thus the timed composition with FMI framework is straightforward, and the tasks is to find a systematic way of handling discrete synchronizations.

UPPAAL supports the notion of discrete I/O synchronization natively by means of input and output channel labels, thus its discrete input and output transitions can be mapped directly to input/output port variables of FMU dedicated to transfer the synchronization label name. Nonetheless, we distinguish the following kinds of transitions: internal (transitions without I/O channel synchronization or internally synchronized transitions for which channels are not marked as input nor output), input transitions (labeled by an input synchronization where the channel name is marked as FMU input) and output transition (labeled by an output synchronization where channel is marked as FMU output). The marked outputs are controlled by UPPAAL simulation and are executed asynchronously irrespectively of whether the receiving FMU is ready to synchronize. Meanwhile the input transitions are executed only when there is a corresponding input label set on discrete input port. Only one (internal, input or output) transition at most is possible at a time, hence a fine-grained simulation control can be achieved by the co-simulation algorithm.

UPPAAL FMUs do not introduce I/O dependencies between continuous variables because the models do not use algebraic expressions to compute variable values. Instead of algebraic expressions the automata models use discrete transitions to update the variable values, hence the variable output ports depend on synchronizing label input, moreover only one discrete transition is allowed hence output synchronization also depends on synchronization label input.

3.2 SpaceEx

SPACEEX [15] uses hybrid automata to describe the system behavior where the continuous variable derivatives are constrained with differential equations. The continuous variables are candidates for input and output exchange via FMU ports. The discrete transitions of hybrid automata can be decorated with labels, and synchronization may involve multiple participating processes, but there is no notion of input and output – all processes are equal contributors, therefore the simulator needs to implement the input/output semantics required by FMI. We use a special label naming notation to mark input and output labels (see Fig. 6). The transitions with input labels are to be executed only when the discrete input variable of FMU is set to the corresponding label name. Meanwhile the transitions with an output label is controlled by SPACEEX simulation, is executed asynchronously by setting the discrete output variable with the label name irrespectively of whether the receiving FMU can synchronize with it. We ensure the SPACEEX FMU determinism by enforcing the must-semantics of discrete transitions in a hybrid automaton. In other words, a discrete transition is taken as soon as its guard is enabled. Finally, we resolve the non-determinism between input, output and internal transitions in the following way: Input transitions have priority over output transitions and output transitions are preferred over the internal ones.

3.3 Discussion on Co-Simulation Semantics

In this section we discuss the co-simulation semantics and contrast it to those typically used by a model-checking tool. In particular, we demonstrate by example how the FMI co-simulation algorithm resolves input/output dependencies and contrast it with execution analysed in a model checker. Our goal is to offer insights in the differences of the two semantics.

Consider a system model shown in Fig. 1 which consists of four timed automata composed in parallel. Labels of the form $a!$ denote sending output a, whereas $a?$ denotes receiving input a. The variable x is a *clock* measuring time. The constraint $x = 1$ is a *guard* which allows the corresponding transition of the automaton to occur only if the guard is satisfied, i.e., in this case only when x equals 1. Clocks are initially zero. The automata synchronize in a chain: the first automaton can output a to the second one, the second one can output b to the third one and so on.

In principle, the system can be loaded into FMI model in any combination: individually (one automaton per FMU) or collectively (multiple automata per FMU), but before an FMU can be loaded into FMI model, it must declare its input/output dependencies. According to [8] each automaton should expose an input/output variable which will contain a label value that is being synchro-

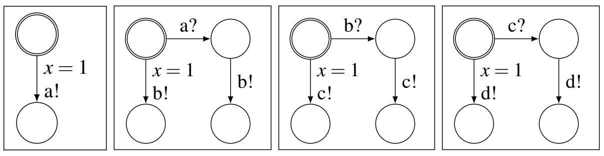


Figure 1. An example timed automata chain.

nized. The first automaton in the example above will have only an output variable, which may have values {*a, absent*}. The second automaton will have an input variable ranging over {*a, absent*} and an output variable ranging over {*b, absent*}, and so on. The value *absent* denotes that currently there is no synchronization and a specific label value denotes a synchronization using that label. Timed automata must declare a dependency between its input and output label variable in order to avoid simultaneous input and output synchronization.

In addition, it is assumed that each FMU is *input-enabled*, meaning that it can handle (i.e., it is able to receive) any declared input at any time. If a component is not input-enabled and an input synchronization is triggered then simulation is aborted. An easy way of achieving input-enableness is to use broadcast channels, which do not block the sender process and receiver may simply ignore the synchronization.

Suppose the automata from Fig. 1 are loaded within separate FMUs and connected according to synchronization labels. That is, the output of the first (leftmost) automaton is connected to the input of the second, the output of the second automaton is connected to the input of the third, and so on. The co-simulation algorithm would detect that it has to fulfill inputs values for the fourth, third, and second automata in order to proceed, therefore the input/output value propagation will have to start with the first automaton and then proceed to the second etc. Once the values of all input and output variables are updated, the algorithm proceeds with advancing each FMU in time. It is this dynamic behavior in time which interests us in this example.

In particular, observe that the second, third, and fourth automata in the example are non-deterministic in the sense that, according to UPPAAL semantics, at time $x = 1$ an automaton can either delay, or take an outputting transition, or synchronize on inputs. For instance, at time $x = 1$, the second automaton can either emit b, or receive a (which will be available in this case, because it is sent by the first automaton at exactly that time), or let time pass. In UPPAAL/verification semantics, all these options are possible at the individual component level. Moreover, not only individual components can be non-deterministic, but their composition is non-deterministic as well, based on so-called *interleaving semantics*. This means that when multiple automata are enabled at a given time, the choice of which one to execute is arbitrary. Non-deterministic behavior is standard (and useful) in verification and model-checking applications. The same is true when these tools are used for simulation. That is, different simulations in UPPAAL can yield different results.

In FMI, the situation is very different, as both FMUs are deterministic components, and their composition, ensured by the co-simulation algorithm, is guaranteed to yield deterministic results as well. Interestingly, in this example, if all automata decide to output at time $x = 1$, some of them will succeed outputting in parallel, while others will be preempted by incoming inputs. In particular, the master algorithm will request the first automaton to produce its output, and thus the second will be busy handling an input and will not be able to produce output at that time. Since the second is not sending anything, then the third automaton will be free to produce an output and preempt the fourth one. On one hand, such FMI system will not be able to simulate all execution orders that would be considered in model-checking (i.e. if automata decide to output at the same time, then only one order can be simulated due to dependencies). On the other hand, simulations may contain parallel synchronizations (e.g. between first and second, and between third and fourth) which are possible only in several steps in model-checkers.

4 Case Study

We have implemented the FMI standard in the UPPAAL [16] and SPACEEX [15] model checkers by providing model export to FMU. In this section, we present and evaluate the performance of the resulting FMI framework on a case study inspired by the well-known room heating benchmark originally proposed by Fehnker et al. [13]. Our model consists of a room with a heater (Fig. 2) and a controller (Fig. 3) which regulates the heater behavior. We model the room and the controller as a SPACEEX and UPPAAL FMUs, respectively (see

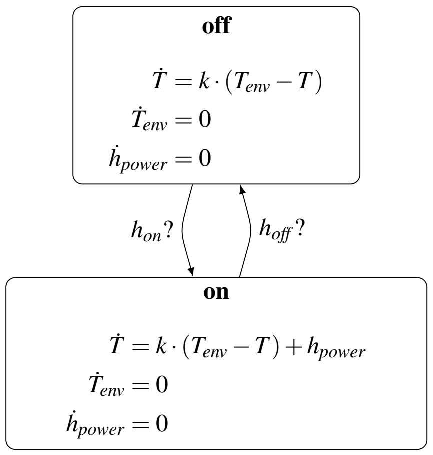


Figure 2. Room component modelled in SPACEEX. The component switches between "on" and "off" modes. The temperature variable T is exported as output and synchronizations labels h_{on} and h_{off} as inputs.

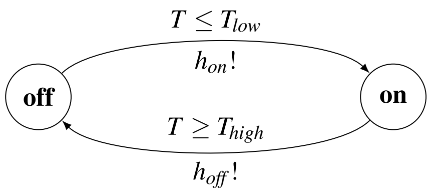


Figure 3. Bang-bang controller component modelled in UPPAAL uses urgent channels to ensure as-soon-as-possible transition trigger. Temperature variable T is an input and synchronization labels h_{on} and h_{off} as outputs.

Fig. 4). We consider a bang-bang controller which turns the heater on and off as soon as some temperature thresholds T_{low} and T_{high} have been reached. The as-soon-as-possible behavior is enforced by using urgent channels which effectively make the controller deterministic. The room temperature T evolves according to the following differential equation:

$$\begin{aligned} \dot{T} &= k \cdot (T_{env} - t) + h_{power} \\ \dot{T}_{env} &= 0 \\ \dot{h}_{power} &= 0 \end{aligned}$$

In other words, the room temperature depends linearly on the difference between the current room temperature T and outside temperature T_{env}. We assume the outside temperature T_{env} and heater power h_{power} to be constant. The constant k defines the heat exchange rate between the room and outside environment. If the heater is off, the heater power is set to zero.

4.1 Evaluation

We evaluate our FMU framework by comparing simulation trajectories of the FMUs with the ones produced by a SPACEEX model consisting of both the controller and room components. We consider three different simulation step values: 1 (see Fig. 5a), 0.1 (see Fig. 5b) and 0.01 (see Fig. 5c). Considering the simulations, we observe that the FMU trajectories *overshoot* the controller constraints in the sense that the controller exhibits a delayed reaction when the room temperature crosses the temperature thresholds. The behavior is justified by the fact that the method call `doStep` for every FMU relies only on the *local* information about the state evolution when making decisions, e.g., the controller FMU does not have any information about the room temperature evolution beyond the value which can be provided when the method `doStep` is called. Therefore, the controller FMU detects that the guard is enabled only a *simulation iteration later* after this event has already happened. We observe that the impact of the overshooting can be made arbitrary small by choosing a small enough simulation step (see Fig. 5c vs. Fig. 5a and Fig. 5b). We note that the overshooting problem is *inherent* to the considered master algorithm and can be circumvented by incorporating additional cross-component knowledge into

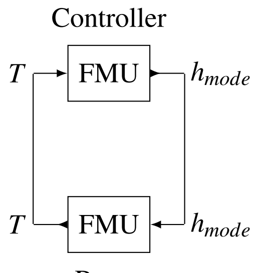
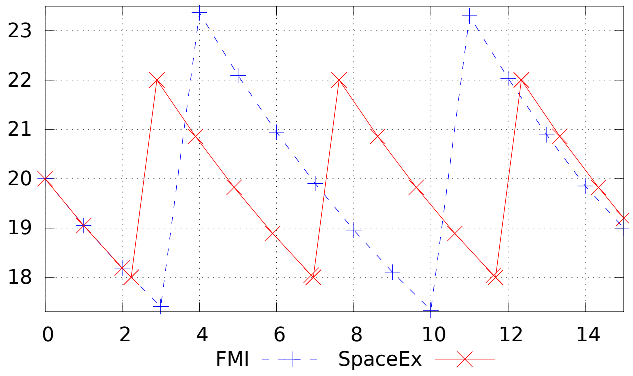
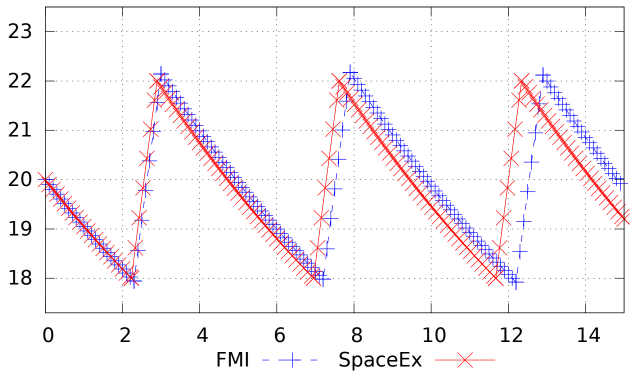


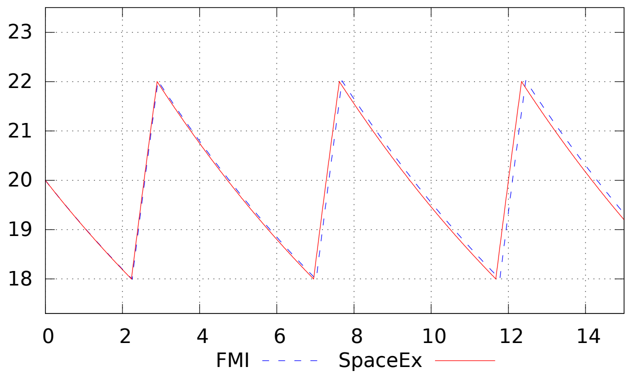
Figure 4. SPACEEX and UPPAAL FMUs connected using variables T and h_{mode} – the room temperature and heater mode, respectively.



(a) Maximum step size 1.



(b) Maximum step size 0.1.



(c) Maximum step size of 0.01.

Figure 5. Simulation trajectories: each red × is a data point reported by SPACEEX, and blue + reported by the co-simulation.

the master algorithm. Overall, our experiments validate that our co-simulation framework based on SpaceEx and Uppaal provides equivalent simulation results compared to the setting where all components are modelled in one tool.

4.2 Supervisory Control Example

In this section we show how supervisory control systems similar to [13] benchmark can be setup within FMI. Suppose we would like to model a building heating setup with two rooms sharing a common wall and a heater. The room temperature now is influenced not just by outside temperature but also heat transfer between the rooms. Figure 6 shows a hybrid automaton from SpaceEx modeling the room temperature dynamics. The difference from a previous example here is an extra term $(Tother - t) * 0.2$ denoting a contribution from another room. Another room is modeled analogously except that it responds to *heater2_on* and *heater2_off* signals instead of *heater1_on* and *heater1_off*.

Our controller consists of two parts: local bang-bang controller and a supervisor shown in Fig. 7. In order to emulate the move of heaters we create instances of controllers which can be disabled and enabled by supervising controller, hence the local controller has an extra mode besides *On* and *Off*. The supervising controller has two kinds of stochastic behavior: it can pick any pair of rooms (one recipient and another donor) to transfer the heater, and it can choose the timing of transfer. When a pair of rooms is selected (by choosing concrete room identifiers for *rec* and *donor* variables) the donor is disabled by moving from location *decide* to location *move* and the recipient is enabled by going from *move* to *idle*. The supervisor may stay in location *idle* arbitrary long, but the exact duration is decided by an exponential probability distribution of rate 1 (which means 1/1 time units on average). Similarly the supervisor may stay in *decide*

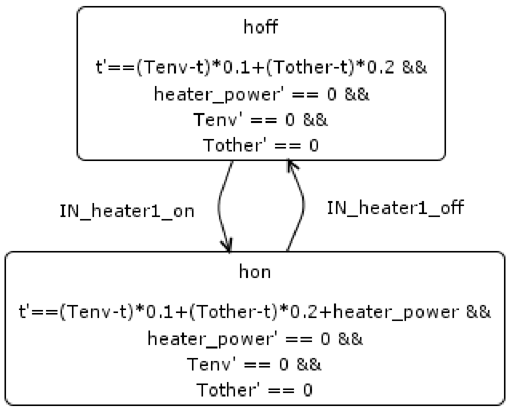
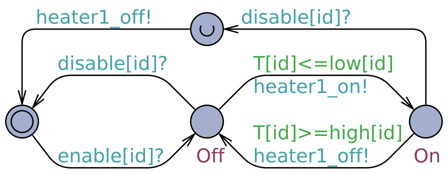
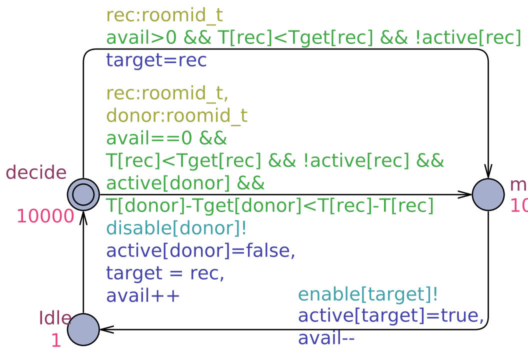


Figure 6. Hybrid automaton for a heated room connected to another room. Inputs are temperatures *Tenv*, *Tother* and labels *IN_heater1_on* and *IN_heater1_off*, while output is temperature t. We use the prefix *IN* to mark input labels.



(a) Local bang-bang controller which can be moved (disabled). The inscribed U means urgent location where time delay is not allowed. The inputs are temperature variable *T[id]* and labels *enable[id]* and *disable[id]*, while outputs are labels *heater1_on* and *heater1_off*.



(b) Supervising controller moves the heaters between rooms by reading inputs on *T[i]* and sending outputs on labels *enable[i]* and *disable[i]* where *i* is the room index.

Figure 7. Two layers of Uppaal controllers.

and *move* but the duration will be 1/10000 on average, i.e. denoting that the heater can be transferred from one room to another rather quickly.

Figure 8 shows the overall component connectivity diagram where the supervisor is reading temperatures from each room and controls the local movable heater controllers. The movable heaters then may either turn on the heat in their room or let them cool off giving the heat to outside. The individual heated rooms are then connected to the outside temperature and to each other denoting the heat exchange. The splitter FMUs are repeaters needed to connect multiple components to the same signal.

Once connected, the composed model is then simulated using FMI master algorithm. Figure 9 shows the temperature dynamics in each room. In particular, the plot shows that in the beginning the temperature drops until the supervisor detects a room temperate below $Tget = 17°$, then around 6 time units a heater raises the temperature in room 1. The local controller keeps rising the temperature until it goes over 22° bound at around 7.5 time units. Notice that the temperature in room 2 also rises due to heat exchange between the rooms. Around 10 time units the supervisor decides to hand over the heater to room 2. At 14 time units the heater is switched back to room 1 and so on. We can conclude that even though the temperature drops well below 18° overall it seems that the controllers manage to sustain the temperature at the similar level without loosing control (without dropping to outside temperature level).

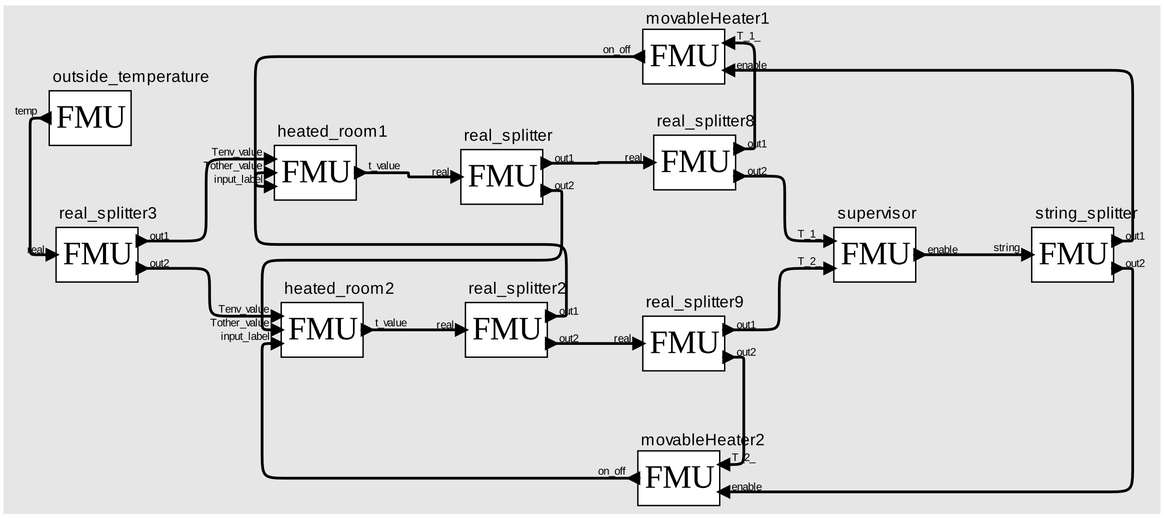


Figure 8. Ptolemy diagram for supervisory control of two heated rooms.

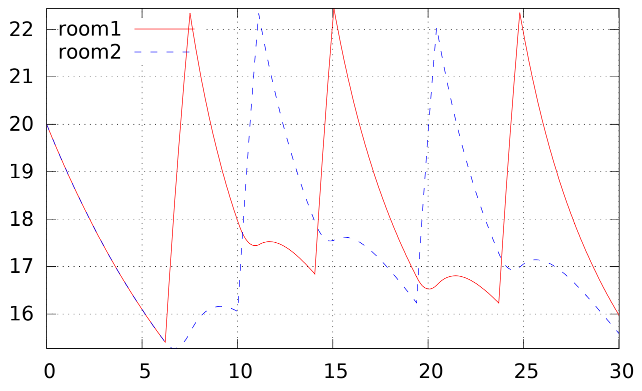
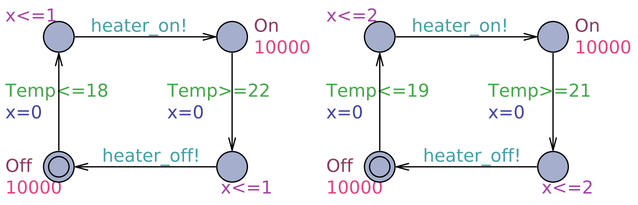


Figure 9. Temperature trajectories for each of the rooms composed with stochastic supervising controller.



(a) Wide range and fast. **(b)** Narrow range and slow.

Figure 10. Stochastic controllers use regular (non-urgent) channels, therefore timings are stochastic: delays are distributed uniformly (when clock invariant is used) and exponentially (in locations *On* and *Off*).

4.3 Stochastic Simulations and SMC

The following is a demonstration of statistical model checking (SMC) using the FMI framework. We show how the performance of two stochastic controllers simulated by UPPAAL can be compared using SMC approach together with the heated room simulation provided by SPACEEX. Figure 10 shows two controllers: (a) reacting within 1 time unit to 18.0° and 22.0° temperature bounds and (b) reacting within 2 time units to 19.0° and 21.0° temperature bounds. The channels used in these controllers are not urgent and therefore the delay between temperature detection and heater activation is decided stochastically based on uniform distribution over the allowed delay by invariants, i.e. the concrete delay will be chosen from $[0,1]$ for the first controller and from $[0,2]$ for the second one. The *On* and *Off* locations do not have any invariant and therefore in principle the process may stay there forever. In such cases UPPAAL uses an exponential (Poisson) probability distribution to decide a particular time delay and hence asks to provide a rate of the exponential. The higher the exponential rate, the shorter the delays, hence we can provide a high rate to ensure that the detecting transition is fired arbitrary quickly.

In our setup, we would like to know which controller is better at keeping the room temperature within 18.0° and 22.0° bounds. In order to answer this question we setup two FMI models for each controller with an equal room, run 100 simulations with 100 time units in length and 0.05 granularity, compute the amount of time spent outside the temperature range for each simulation and then compute the confidence intervals for both models. Table 1 shows a summary of amounts of time during which the temperature was either below or above the range. The estimated time duration use confidence interval (CI) notation which means that if we repeat the measurement experiment then the real mean (which is unknown) will fall into the interval with a probability of 95%. The results show that the second controller was more successful at maintaining the lower bound of the temperature, but was more overshooting beyond the upper bound. In total, the first controller kept the temperature in good range longer by 8.57 time units on average, which is much larger than confidence interval, hence the first controller is better.

Table 1. Time with temperature outside the range (95% CI).

Controller	Time below	Time above	Total
Wide and fast	7.56 ± 0.20	32.69 ± 3.36	40.26 ± 0.59
Narrow and slow	2.40 ± 0.19	46.43 ± 0.82	48.83 ± 0.79

5 Related Work

The FMI standard and corresponding documentation are constantly evolving, as new versions of the standard are developed. The web site[3] also contains a list of tools supporting FMI. Descriptions of FMI can also be found in the academic literature [7].

Discussions about the limitations of FMI can be found in the works by Broman et al. [8, 9]. Broman et al. [8] also formalize the main methods of FMI (`get`, `set`, `doStep`) by establishing a *contract* (pre-/post-conditions) for each method and propose a *master algorithm* (i.e., a co-simulation algorithm). Furthermore, the authors proves its termination, determinacy, and other properties. However, the paper does not discuss how FMUs can be created. A different, master-slave based, co-simulation approach is proposed by Bastian et al. [5], but formal properties such as determinacy are not discussed in this work.

Broman et al. [9] defines a suite of test models that should be supported by a hybrid co-simulation environment, giving a mathematical model of an ideal behavior, plus a discussion of practical implementation considerations. Furthermore, the paper describes a set of basic modeling components in the spirit of Ptolemy actors (constant, gain, adder, integrator, etc.). Finally, the authors provide a kind of denotational description for each component (input and output signals), but no encoding into FMUs is discussed.

Translating different formalisms to FMUs is discussed in the paper by Tripakis et al. [20]. This work only refers to a generic model of timed machines which does not include the particularities of UPPAAL's timed automata. In addition, hybrid automata are not considered in this work.

Recently, the co-simulation algorithm presented by Broman et al. [8] has been implemented in the open-source Ptolemy tool. As mentioned above, we use this framework in order to import FMUs into Ptolemy and co-simulate them. However, this framework does not address the FMU creation problem.

Feldman et al. [14] present a plugin for Rhapsody for generating FMUs from Statechart SysML blocks. They provide high level guidelines for how to generate Statechart FMUs, but do not provide a formalization. Pohlmann et al. [19] also discuss how to encode statechart models, described in MechatronicUML.

Co-simulation is one, but not the only approach to solve the tool interoperability problem. A further attempt to solve this problem is the *Hybrid Systems Interchange Format* (HSIF), designed with the goal of being "a sort of 'maximum common denominator' among all hybrid system modeling environments" [17]. HSIF aims therefore at defining a "maximal syntax" where all the syntax of different languages could be translated into. It could be seen as a type of XML schema for hybrid systems. HSIF is primarily aimed at enabling model *translation* between different hybrid system tools. Bak et al. [4] present the tool Hyst which provides an automatic source-to-source model translation between a number of up-to-date hybrid model checkers. In their approach, Bak et al. do not use any intermediate format like HSIF. Both model translation-based approaches outlined above provide support only of the *common subset* of the tool features. Our co-simulation framework does not limit the model designer to use the "maximal syntax" among all the tools because every FMU takes care of its features independently.

6 Conclusions

We have shown how two state-of-the-art modeling and verification tools for hybrid systems, SPACEEX and UPPAAL, can be integrated using the FMI co-simulation standard. The result is a powerful framework which allows users to build submodels separately in each of the two tools (as well as in other tools potentially), then generate individual FMUs for each submodel, and then combine all the FMUs into a single model, which can be co-simulated within the Ptolemy FMI implementation. We demonstrated the feasibility of our framework by comparing the co-simulation results of a simple two-FMU model to the simulation results that are obtained when we model and simulate the entire system in a single tool. We found that, provided time steps are small enough, individual components can ensure timely reactions to continuous signals, and the simulations can be made arbitrary close to self-contained model simulation. In addition to individual tool export to FMUs, we showed how the non-deterministic models can be determinized using stochastic semantics and included into FMI co-simulation. We also provided an example how statistical model checking can be performed using numerous FMI simulations which is an essential feature evaluating stochastic behavior. The integration of model-checkers into co-simulation frameworks provides further possibilities of analyzing early design models like conformance monitoring by checking that a simulation trace of a refined (e.g. hybrid) model is included in a more a abstract (e.g. timed automata) specification. We envision our work being a further step towards integrating tools developed in the formal methods community into the industrial system design and modeling workflow of cyber-physical systems.

[3]`https://www.fmi-standard.org/`

7 Acknowledgments

We are grateful to Christopher Brooks, Fabio Cremona, and Edward Lee from UC Berkeley, for their work on the Ptolemy framework. This work was partially supported by Academy of Finland, by the National Science Foundation (awards #1329759 and #1139138), and by the Industrial Cyber-Physical Systems Research Center (iCyPhy), which is supported by IBM and United Technologies Corporations.

References

[1] D. N. Agut, D. van Beek, and J. Rooda. Syntax and semantics of the compositional interchange format for hybrid systems. *The Journal of Logic and Algebraic Programming*, 82(1):1 – 52, 2013. ISSN 1567-8326. doi:10.1016/j.jlap.2012.07.001. URL `http://www.sciencedirect.com/science/article/pii/S1567832612000689`.

[2] R. Alur and D. L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126:183–235, 1994.

[3] R. Alur, C. Courcoubetis, N. Halbwachs, T. A. Henzinger, P.-H. Ho, X. Nicollin, A. Olivero, J. Sifakis, and S. Yovine. The algorithmic analysis of hybrid systems. *Theoretical Computer Science*, 138(1):3–34, 1995.

[4] S. Bak, S. Bogomolov, and T. T. Johnson. HYST: a source transformation and translation tool for hybrid automaton models. In *Proceedings of the 18th International Conference on Hybrid Systems: Computation and Control, HSCC'15, Seattle, WA, USA, April 14-16, 2015*, pages 128–133. ACM, 2015.

[5] J. Bastian, C. Clauß, S. Wolf, and P. Schneider. Master for Co-Simulation Using FMI. In *8th International Modelica Conference*, 2011.

[6] H. Beohar, D. E. N. Agut, D. A. van Beek, and P. J. L. Cuijpers. Hierarchical states in the compositional interchange format. In L. Aceto and P. Sobocinski, editors, *Proceedings Seventh Workshop on Structural Operational Semantics, SOS 2010, Paris, France, 30 August 2010.*, volume 32 of *EPTCS*, pages 42–56, 2010. doi:10.4204/EPTCS.32.4. URL `http://dx.doi.org/10.4204/EPTCS.32.4`.

[7] T. Blochwitz, M. Otter, M. Arnold, C. Bausch, C. Clauß, H. Elmqvist, A. Junghanns, J. Mauss, M. Monteiro, T. Neidhold, D. Neumerkel, H. Olsson, J.-V. Peetz, and S. Wolf. The Functional Mockup Interface for Tool independent Exchange of Simulation Models. In *8th International Modelica Conference*, Dresden, Germany, Mar. 2011. Modelica Association.

[8] D. Broman, C. Brooks, L. Greenberg, E. A. Lee, S. Tripakis, M. Wetter, and M. Masin. Determinate Composition of FMUs for Co-Simulation. In *13th ACM & IEEE International Conference on Embedded Software (EMSOFT'13)*, 2013.

[9] D. Broman, L. Greenberg, E. A. Lee, M. Masin, S. Tripakis, and M. Wetter. Requirements for Hybrid Cosimulation Standards. In *Hybrid Systems: Computation and Control (HSCC)*, 2015.

[10] A. David, K. G. Larsen, A. Legay, M. Mikučionis, D. B. Poulsen, J. van Vliet, and Z. Wang. Statistical model checking for networks of priced timed automata. In U. Fahrenberg and S. Tripakis, editors, *Formal Modeling and Analysis of Timed Systems*, volume 6919 of *Lecture Notes in Computer Science*, pages 80–96. Springer Berlin Heidelberg, 2011.

[11] A. David, K. G. Larsen, A. Legay, M. Mikučionis, and D. B. Poulsen. Uppaal smc tutorial. *International Journal on Software Tools for Technology Transfer*, pages 1–19, 2015.

[12] J. Eker, J. Janneck, E. Lee, J. Liu, X. Liu, J. Ludvig, S. Neuendorffer, S. Sachs, and Y. Xiong. Taming heterogeneity – the Ptolemy approach. *Proceedings of the IEEE*, 91(1):127–144, Jan. 2003.

[13] A. Fehnker and F. Ivancic. Benchmarks for hybrid systems verification. In *In Hybrid Systems: Computation and Control (HSCC 2004)*, pages 326–341. Springer, 2004.

[14] Y. A. Feldman, L. Greenberg, and E. Palachi. Simulating Rhapsody SysML Blocks in Hybrid Models with FMI. In *10th Modelica Conference*, pages 43–52, 2014.

[15] G. Frehse, C. Le Guernic, A. Donzé, S. Cotton, R. Ray, O. Lebeltel, R. Ripado, A. Girard, T. Dang, and O. Maler. SpaceEx: Scalable Verification of Hybrid Systems. In S. Q. Ganesh Gopalakrishnan, editor, *23rd International Conference on Computer Aided Verification (CAV)*, LNCS. Springer, 2011.

[16] K. G. Larsen, P. Pettersson, and W. Yi. Uppaal in a nutshell. *International Journal on Software Tools for Technology Transfer*, 1(1-2):134–152, 1997.

[17] A. Pinto, A. L. Sangiovanni-Vincentelli, L. P. Carloni, and R. Passerone. Interchange formats for hybrid systems: review and proposal. In *Hybrid Systems: Computation and Control*, HSCC'05, pages 526–541. Springer, 2005.

[18] A. Pinto, L. P. Carloni, R. Passerone, and A. Sangiovanni-Vincentelli. Interchange format for hybrid systems: Abstract semantics. In J. P. Hespanha and A. Tiwari, editors, *Hybrid Systems: Computation and Control*, volume 3927 of *LNCS*, pages 491–506. Springer Berlin Heidelberg, 2006.

[19] U. Pohlmann, W. Schäfer, H. Reddehase, J. Röckemann, and R. Wagner. Generating Functional Mockup Units from Software Specifications. In *9th Modelica Conference*, pages 765–774, 2012.

[20] S. Tripakis and D. Broman. Bridging the Semantic Gap Between Heterogeneous Modeling Formalisms and FMI. Technical Report UCB/EECS-2014-30, EECS Department, University of California, Berkeley, Apr 2014. URL `http://www.eecs.berkeley.edu/Pubs/TechRpts/2014/EECS-2014-30.html`.