Integrating Tools

Co-simulation in UPPAAL Using FMI-FMU

Nyman, Ulrik; Jensen, Peter Gjøl; Larsen, Kim Guldstrand; Legay, Axel

Published in:
Proceedings - 2017 22nd International Conference on Engineering of Complex Computer Systems, ICECCS 2017

DOI (link to publication from Publisher):
10.1109/ICECCS.2017.33

Creative Commons License
Unspecified

Publication date:
2018

Document Version
Accepted author manuscript, peer reviewed version

Integrating Tools: Co-Simulation in UPPAAL using FMI-FMU

Ulrik Nyman, Peter Gjøl Jensen, Kim Guldstrand Larsen
Computer Science, Aalborg University, Denmark
{nyman,pgj,kgl}@cs.aau.dk

Axel Legay
INRIA, Rennes, France
Computer Science, Aalborg University, Denmark
legay@inria.fr

***Abstract*—While standalone tools for verification and modeling have proven useful, their chosen formalism and description-language can at times be restrictive. We demonstrate how to use UPPAAL SMC to analyze controller systems consisting of Function Mockup Units (FMU) modeled in other tools, such as Matlab and Modelica. Apart from supporting FMI-FMU modules the newly added C interface can call any external function. The only requirement for sound analysis is statelessness and determinism of the external function. We demonstrate the expressive power by implementing the FMI-FMU master algorithm as a timed automata, interfacing with external, non-native and non-trivial Function Mockup Units (FMU). We also model two components in UPPAAL SMC exporting one of them as an FMU while keeping the other as a native component. Furthermore we demonstrate the first simulation environment for the Function Mockup Units, capable of checking bounded MITL properties.**

I. Introduction

Model-checking tools often come with support for a specific formalism, imposing restrictions on the behavior of the modeled system. Such formalisms serve the purpose of giving the system a *semantics* such that the behaviour of the system carries a sound and consistent meaning. In particular without a semantics, investigating the accuracy or probabilistic behavior of a system is nonsensical. At the same time, a variety of formalisms have emerged in different domains, each well suited for a specific task (eg. digital, mechanical or thermodynamic modeling), but incapable of *co-simulating* – that is, to obtain joint results. To remedy this, the *Function Mockup Interface*-standard (FMI) [16] was proposed to enable domain-specific modeling tools to be used side by side when encapsulated in a *Function Mockup Unit* (FMU). In particular, the FMI-standard defines a protocol for how values *can* be communicated between FMUs by standardizing the interface-description and giving a common C-api.

However, while the FMI standard only specifies how values *can* be communicated, it purposely does not specify the details of how they *will* be communicated, calling for the development of so-called *Master Algorithms* (MA) for coordinating the interaction between several FMUs.

As previously demonstrated [5], the MA has an impact on the semantics of the overall system, and is thus of great importance to the overall meaning of the measures obtained. We demonstrate that our implementation of the MA will given enough simulations eventually explore all possible executions. This ensures that we will also discover cases where the ordering of events has a great impact on the outcome.

We will be implementing our approach as an extension of the statistical model checking tool UPPAAL SMC [13], thus ensuring that we have a formally defined semantics. We show that meaningful probabilistic measures can be obtained, and in particular that we can statistically verify MITL (Metric Interval Temporal Logic) properties. The work presented here extends beyond applications within the FMI-standard – we propose and implement an extension of UPPAAL that allows for calling arbitrary C-libraries during the statistical simulation. This effectively opens up UPPAAL SMC to a great number of applications – and we will argue that under statelessness of the external library, that doing so is semantically sound. By statelessness, we do not mean that the components cannot change their state, but that all relevant state information should be communicated through the FMI-FMU interface such that it is controlled by the MA. This is also required in order to retain the semantics in future settings such as classical model checking, as demonstrated in [9]. External FMUs can contain stochastic behavior, but they should not contain unresolved non-determinism, as such non-determinism would not have a meaningful interpretation when simulating the system.

Notice that for FMUs we embrace the argument of Broman et. al. [6] – that it is sane to expect that components only can give a maximum delay and must accept all smaller delays.

A. Interleaving Semantics

While the work of Bogolomov et. al [5] work showcased UPPAAL as an FMU alongside SpaceEx with Ptolemy as MA, it also exposed inconsistencies in the semantics of timed automata when recomposed via the MA as opposed to internally in UPPAAL. Let us recall the example provided by Bogolomov et. al; consider the four TAs presented in Figure 1. If the four TAs communicate in a pipeline pattern s.t. A1 outputs to A2, A2 to A3 and A3 to A4. Given that all TAs start in their initial location (marked by double-circle) and with clock-value x1=x2=x3=x4=0 nothing will happen until a single unit of time has elapsed ($x1 == 1$, $x2 == 1$, $x3 == 1$ and $x4 == 1$). After exactly one time unit (enforced by the invariants $x1 <= 1$, $x2 <= 1$, $x3 <= 1$ and $x4 <= 1$), all the automata are able to output – however, as TAs synchronize on channels (a, b, c, d, output marked by ! and input by ?), it becomes important in which order the automatas synchronize. Notice here that even though all the synchronizations happen at the same instance of time, it is important to track the "happened before" relationship. Keeping this in mind, we can observe that only a single deterministic trace of the system can occur given the pipeline communication-pattern, namely that an $a!$ always will be observed followed by $c!$, implying that on all traces, eventually the product-state A1.A && A2.AB && A3.C && A4.CD is reached. However, one can easily verify using UPPAAL that this is not the case. In fact, all possible permutations with at least one winner is possible, as UPPAAL implements interleaving semantics – which most definitions of NTA (Networks of Timed Automata) require.

We argue that such behavior is also important in real-life models, for instance due to a too coarse granularity on the observation of time. In particular in a probabilistic context, such under-approximation of the behavior of the system leads to erroneous probability estimates. While we do not address channel-based synchronization between FMUs in this paper, we note that similar erroneous behavior can be achieved using only integer-valued variables. For the specifics of channel based synchronization between FMUs, we refer the reader to [5].

Another known problem in modeling and simulation is that of *zero-crossing*; namely that it is impossible in a simulation setting to detect the exact time when a certain value crosses a zero threshold.

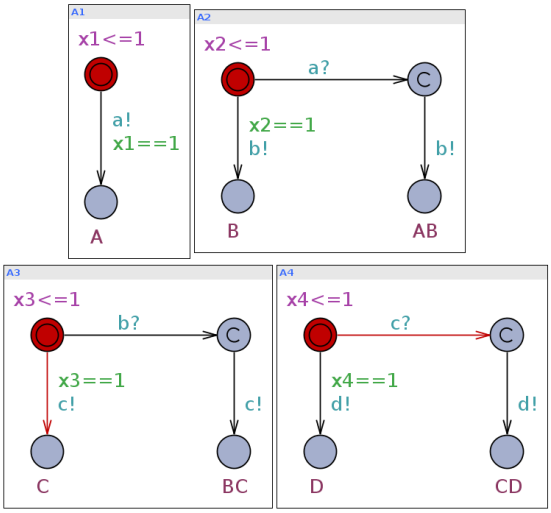


Fig. 1: Four UPPAAL timed automata exemplifying the different semantics of different Master Algorithms. An intuitive explanation of the syntax of UPPAAL timed automata can be found in Section V-B.

As mentioned in [10] this is also a problem in the context of FMI-FMU models. This is a problem that we are aware of, but that our current solution does not try to address.

B. Contributions

We demonstrate the ease with which already existing system models from Modelica can be exported as FMUs and used for verifying statistical properties of the model in UPPAAL SMC. In particular, the ability to specify a MA within a sound semantical framework facilitating probabilistic and temporal reasoning is a strength of the methods proposed in this paper. These features are essential to modeling real-world scenarios where behavior is inherently uncertain and time-dependent – such scenarios include signal noise, human interaction and general natural phenomena. As we base our approach on top of a tool that already supports the notions of time and probabilities, we automatically gain the analytical capabilities of this tool. While we only demonstrate the Statistical Modelchecking features of UPPAAL SMC in this paper, our proposed method of embedding FMUs as timed automata also enables classical model-checking, controller synthesis and controller learning on composed models using the more complex features of UPPAAL, UPPAAL TIGA and UPPAAL STRATEGO under some reasonable restrictions on the FMUs.

The contributions of the paper can be summarized as follows:

- Implementation of direct call of FMI-FMU modules from inside a statistical model checking tool that supports time and probabilities.
- Discussion on different semantics for FMI-FMU implementations.
- Flexible modeling of the master algorithm as a timed automaton template.
- Statistical model checking of bounded reachability of temporal logics (MITL).

C. Related Work

FMI-FMU has its origin in the European research project MODELISAR. It was created specifically for integrating a wide variety of modeling tools. Industry tools that support the FMI-FMU standard include: MATLAB/Simulink, MapleSim and AUTOSAR Simulation [16].

In the following we try to cover recent and relevant related work that take a practical approach to co-simulating industrial systems.

Pazold et. al. [22] compare different approaches for simulating a HVAC system for a complete building. They conclude that a weak coupling using a co-simulation strategy using sub-models exported as FMUs is a reasonable approach. A case-study by Pedersen et. al. [23] describes how to integrate a special purpose maritime embedded system into an FMI-FMU co-simulation setting at the company MAN Diesel & Turbo.

We also find that it is relevant to look at related work that considers the semantics of the complete co-simulation system and Master Algorithm (MA).

In [18] Guermazi et al. provide a framework for co-simulating the UML models specified in Papyrus (the open-source UML/SysML modeler of the Eclipse foundation) in an FMI-FMU context. The work is build on top of the formal semantics foundation of UML (fUML [21])

In [6] Broman et al. argue for an number of sanity conditions for a MA. In this work we deviate by 1. allowing the order of FMI/FMU-definition determine the outcome of the simulation and 2. not requiring an order on the input/output-relation. However, both properties can be ensured by modifying the proposed timed automaton template for a MA. At the same time, Broman et al. propose the `getMaxStepSize`-extension of the FMI-FMU standard to enable step-size negotiation between FMUs.

In [5] Bogomolov et al. argue for allowing Zero-delay step-sizes in the `doStep`-procedure to enable timed automata style synchronizations across FMUs – a value which is otherwise strictly disallowed by the FMI-FMU standard. This paper also discusses using Hybrid- and timed automata as FMUs and the related semantic difficulties and utilizes the step-negotiation strategy introduced in [6].

In [10] Cremona et al. introduce the concept of step revision, similar to that presented in [6], as a method for ensuring the accuracy of modeling a mix of continuous-time and discrete-event systems in an attempt to address the *Zero-crossing* problem.

Statistical modelchecking of Priced Timed Automata and Stochastic Hybrid Systems was introduced into the UPPAAL tool-chain by Bulychev et al. in [7], [11]. In their work they present methods and algorithms for obtaining various statistical measures over stochastic systems using expressive logics [13], [8]. While the UPPAAL SMC tool has been used in a number of case-studies ([17], [12], [20], [3], [1]) it previously did not facilitate co-simulation as master-algorithm using the FMI-FMU standard.

The PLASMA statistical model-checking tool [19] supports statistical analysis over a System of Systems composed of FMUs [2]. However, their work does not provide a formal semantics of the composed system or individual components, nor does it allow `C`-functions to be embedded directly in the model. Furthermore, the integration of FMUs in `C` allows for analysis, synthesis and learning using the more complex features of classical UPPAAL [4] and UPPAAL STRATEGO [13] – features that are out of the scope of this paper.

II. SEMANTICS

We shall here describe the semantics of Timed Automata (TA), Network Of Timed Automata (NTA), Stochastic Timed Automata (STA) and Function Mockup Units (FMUs). We will then discuss a semantical embedding of FMUs into the the STA framework.

A. Stochastic Timed Automata

Formally, a TA is a finite automaton extended with a set of real-valued, time-progress-measuring counters ($\mathcal{X}$) called clocks. In addition, a TA allows for synchronization with other TAs over a finite set of so-called channels (Σ). For a set of channels Σ we let $\Sigma_o = \{a! \mid a \in \Sigma\}$ be the set of output actions over Σ while we let $\Sigma_i = \{a? \mid a \in \Sigma\}$ be the set of input actions. For a set of clocks $\mathcal{X}$ we call an element $c \bowtie n$ where $c \in \mathcal{X}$ and $n \in \mathbb{N}$ and $\bowtie \in \{\leq, <\}$ ($\bowtie \in \{\geq, >\}$) an upper (lower) bound over $\mathcal{X}$. Let $\mathcal{B}^{\leq}(\mathcal{X})$ ($\mathcal{B}^{\geq}(\mathcal{X})$) be the set of all upper (lower) bounds over $\mathcal{X}$.

We call a mapping $\nu : \mathcal{X} \to \mathbb{R}$ for a valuation over $\mathcal{X}$ and denote all valuations over $\mathcal{X}$ by $\mathcal{V}(\mathcal{X})$.

Definition 1 (Timed Automaton): A Timed Automaton (TA) is a tuple $\mathcal{A} = (L, \ell_0, \mathcal{X}, \Sigma, \to, I, \mathcal{R})$, where

1) L is a finite set of control locations,
2) $\ell_0 \in L$ is the initial location,
3) $\mathcal{X}$ is a finite set of clocks,
4) Σ is a finite set of channels,
5) $\to \subseteq L \times \mathcal{B}^{\geq}(\mathcal{X}) \times (\Sigma_o \cup \Sigma_i) \times 2^{\mathcal{X}} \times L$ is a set of edges. We write $\ell \xrightarrow{g,a,\mathcal{U}} \ell'$ for an edge where ℓ is the source and ℓ' the target location, $g \in \mathcal{B}^{\geq}(\mathcal{X})$ is a guard, $a \in \Sigma_o \cup \Sigma_i$ is a label, and $\mathcal{U} \in \mathcal{V}(\mathcal{X}) \to \mathcal{V}(\mathcal{X})$ is a partial function giving the discrete-clock updates,
6) $I \colon L \to \mathcal{B}^{\leq}(\mathcal{X})$ is an invariant function, mapping locations to a set of invariant constraints and
7) $\mathcal{R} : L \to \mathcal{X} \to \mathbb{R}$ assign rates to the individual clocks in each location[1].

Let $\nu \in \mathcal{V}(\mathcal{X})$ be a valuation, $R : \mathcal{X} \to \mathbb{R}$ give clocks rates, $d \in \mathbb{R}$ be a real-valued number and let $\mathcal{U} \in \mathcal{V}(\mathcal{X}) \to \mathcal{V}(\mathcal{X})$ be an update-function ; then we let $(\nu + d \cdot R)$ be the valuation ν' where $\nu'(x) = \nu(x) + d \cdot R(x)$ and we let $\nu'' = \mathcal{U}(\nu)$. If $g = c \bowtie n$ is a clock bound over $\mathcal{X}$ and $\nu \in \mathcal{V}(\mathcal{X})$ then ν satisfies g ($\nu \vDash g$) iff $\nu(c) \bowtie n$. This generalizes in a natural way to a set of clock bounds.

The state of TA $\mathcal{A} = (L, \ell_0, \mathcal{X}, \Sigma, \to, I, \mathcal{R})$ is a tuple (ℓ, ν) where $\ell \in L$ and $\nu \in \mathcal{V}(\mathcal{X})$. From a state (ℓ, ν) the TA may

1) do a timed transition $(\ell, \nu) \xrightarrow{d} (\ell, \nu')$ if $\nu' = (\nu + d \cdot \mathcal{R}(\ell))$ and $\nu' \vDash I(\ell)$ or
2) do a discrete transitions $(\ell, \nu) \xrightarrow{a} (\ell', \nu')$ if there exists $\ell \xrightarrow{g,a,\mathcal{U}} \ell'$ such that $\nu \vDash g$, $\nu' = \mathcal{U}(\nu)$ and $\nu' \vDash I(\ell')$.

We define a partition over the clocks of a TA into time-independent, real-valued variables ($\mathcal{X}^{\mathtt{V}}$) and time-dependent clocks ($\mathcal{X}^{\mathtt{T}}$) s.t. $\mathcal{X}^{\mathtt{V}} = \{x \in \mathcal{X} \mid \forall \ell \in L$ we have $\mathcal{R}(\ell)(x) = 0$ and x is not restricted by $I(\ell)\}$ and $\mathcal{X}^{\mathtt{T}} = \mathcal{X} \setminus \mathcal{X}^{\mathtt{V}}$.

We define the infix-operator valuation-join operator $\overset{X,Y}{\otimes} : \mathcal{V}(X) \times \mathcal{V}(Y) \to \mathcal{V}(X)$ as

$$\nu \overset{X,Y}{\otimes} \nu' = \nu'' \text{ where } \nu''(x) = \begin{cases} \nu'(x) \text{ if } x \in Y \\ \nu(x) \text{ otherwise} \end{cases}$$

We shall simply write $\otimes$ for this operation and let X, Y be implicitly defined by the given valuations.

Following the compositional framework of [15] we require that a TA for any state s is

1) *input-enabled* i.e. for any $a! \in \Sigma_o$ there exists some s' such that $s \xrightarrow{a!} s'$ and
2) *action-deterministic* i.e. if $s \xrightarrow{a} s'$ and $s \xrightarrow{a} s''$ then $s' = s''$.

Let us now define a Network of Timed Automata (NTA) with shared clocks.

a) Network Of Timed Automata: Let $\mathcal{A}_1, \mathcal{A}_2, \ldots, \mathcal{A}_n$ be TA where $A_i = (L_i, \ell_0^i, (\mathcal{X}_i^{\mathtt{V}} \cup \mathcal{X}_i^{\mathtt{T}}), \Sigma, \to_i, I_i, \mathcal{R}_i)$ with the implicit indices ordering $1 < 2 < \cdots < n$. Let $\mathcal{X}^{\mathtt{V}} = \bigcup_{i \in 1,\ldots,n} \mathcal{X}_i^{\mathtt{V}}$ be the set of shared clocks, then it holds for all $i \in 1, \ldots, n$ that $\mathcal{X}^{\mathtt{V}} = \mathcal{X}_i^{\mathtt{V}}$ and for all $j \in 1, \ldots, n$ where $i \neq j$ that $\mathcal{X}_j^{\mathtt{T}} \cap \mathcal{X}_i^{\mathtt{T}} = \emptyset$. Also let $\mathcal{S}(\mathcal{A}_i) = L_i \times \mathcal{V}(\mathcal{X}_i^{\mathtt{T}}) \times \mathcal{V}(\mathcal{X}^{\mathtt{V}})$; then we define the states of the network $\mathcal{A}_1 \| \mathcal{A}_2 \| \ldots \| \mathcal{A}_n$ to be a pair $\langle (s_1, s_2, \ldots, s_n), \nu \rangle$ where if $(\ell_i, \nu_i) = s_i$ then $(\ell_i, \nu_i \otimes \nu) \in \mathcal{S}(\mathcal{A}_i)$. A network may transit from $\langle (s_1, s_2, \ldots, s_n), \nu \rangle$ by

- a timed transition $\langle (s_1, s_2, \ldots s_n), \nu \rangle \xrightarrow{d} \langle (s_1', \ldots, s_n'), \nu \rangle$ if for all i, $(\ell_i, \nu_i \otimes \nu) \xrightarrow{d} (\ell_i, \nu_i' \otimes \nu)$ where $(\ell_i, \nu_i) = s_1$, $(\ell_i', \nu_i') = s_2$, and
- a discrete transition $\langle ((\ell_1, \nu_1), \ldots, (\ell_n, \nu_n)), \nu \rangle \xrightarrow[i]{a!} \langle ((\ell_1', \nu_1'), \ldots, (\ell_n', \nu_n')), \nu^n \rangle$ if (assuming w.log. that $i = 1$)
  1) $(\ell_1, \nu_1 \otimes \nu) \xrightarrow{a!} (\ell_1', \nu_1' \otimes \nu^1)$, and
  2) for $j \in \{2, \ldots, n\}$ we have

$$(\ell_j, \nu_j \otimes \nu^{j-1}) \xrightarrow{a?} (\ell_j', \nu_j' \otimes \nu^j)$$

Notice that the discrete transition-rule can be generalized beyond $i = 1$ by a temporary reordering of the indices.

B. Stochastic Semantics

David et al. provides the full stochastic semantics of Stochastic Timed Automata in [14]. Here the semantics is given as a series of repeated races among components making up the network. In essence, each sub-component will choose a delay in accordance with the probability distributions defined locally to that component. The component with the smallest delay will then win the race and gets to do a discrete step – again chosen according to a local distribution. However, this discrete step can synchronize with neighboring components, possibly altering their internal state. This procedure is then repeated over and over.

In the semantics the delays are chosen as follows: if the possible delays are bounded, the distribution is a uniform distribution between the minimal delay before some action is possible and the maximal delay where a delay is still possible.

Formally, we assume there for any state (s) of any TA $\mathcal{A}$ exists a delay-density $\delta_s^{\mathcal{A}} : \mathbb{R} \to \mathbb{R}$ and a probability mass function $\gamma_s^{\mathcal{A}} : \Sigma_o \to \mathbb{R}$. Naturally we will require these functions to be "sane" in the sense that they do not assign probability mass (density) to impossible actions (delays) i.e. $\gamma_s^{\mathcal{A}}(a!) \neq 0$ ($\delta_s^{\mathcal{A}}(a!) \neq 0$) implies $s \xrightarrow{a!} s'$ ($s \xrightarrow{d} s'$).

Let $\omega = a_1!a_2! \ldots a_n!$ be a finite sequence of output-actions: then we define the probability of a network $\mathcal{A}_1 \| \ldots \mathcal{A}_m$ generating the sequence from state $s = (s_1, \ldots S_n)$ recursively by Equation 1. where $s_i \xrightarrow{d} s_i^d$, $s \xrightarrow{d} \xrightarrow[i]{a_1!} s'$, $\omega^1 = a_2! \ldots a_n!$ and base case $F_s(\epsilon) = 1$.

C. Function Mockup Unit

Similar to Broman et. al. [6], we shall here define the semantics of a single FMU as a (timed) state machine. For simplicity, we shall in the definition ignore the types of the variables, and assume wlog. that they all are reals.

Definition 2: An FMU is a tuple $\mathcal{F} = (\mathcal{S}, \texttt{init}, \texttt{V}, \texttt{set}, \texttt{get}, \texttt{doStep})$ where

- $\mathcal{S}$ is a set of states,

[1] Allowing rates other than one is non-standard in TA semantics – in fact this renders most classical model-checking questions undecidable. However, as the methods presented are simulation-based, arbitrary rates on clocks are practically feasible and semantically sane [11].

$$F_s(\omega) = \sum_{i=0}^{m} \left(\int_{t>0} \delta_{s_i}^{A_i}(t) \cdot \prod_{j \neq i} \left(\int_{\tau > t} \delta_{s_j}^{A_j}(t) d\tau \right) \cdot \gamma_{s_i^t}^{A_i}(a_1! \cdot F_{s'}(\omega^1) dt \right) \quad (1)$$

- $\texttt{init} \in \mathcal{S}$ is the initial state,
- $\texttt{V}$ is a set of variable names,
- $\texttt{set} : \mathcal{S} \times \mathbb{R}^{\texttt{V}} \to \mathcal{S}$ is a value-setter function,
- $\texttt{get} : \mathcal{S} \times \texttt{V} \to \mathbb{R}$ is a value-getter function and
- $\texttt{doStep} : \mathcal{S} \times \mathbb{R}_{\geq 0} \to \mathcal{S}$ is the time-progression function.

For a given FMU $\mathcal{F}$, the exact semantics is defined by the underlying implementation, and we shall hence only focus on the interaction with STAs.

Definition 3: A stochastic FMU is a tuple $\mathcal{F}_s = (\mathcal{S}, \texttt{V}, \texttt{set}, \texttt{get}, \texttt{doStep}, \mathcal{P})$ s.t.

- $\mathcal{S}, \texttt{V}, \texttt{set}, \texttt{get}, \texttt{doStep}$ are defined as for a regular FMU and
- $\mathcal{P} : \mathcal{S} \to [0, 1]$ is the probability that $\mathcal{F}_s$ starts in $s \in \mathcal{S}$ and we have that $1 = \Sigma_{s \in \mathcal{S}} \mathcal{P}(s)$.

Let $\mathcal{F}_s$ be a stochastic FMU, then we let $\texttt{pick}(\mathcal{F}_s) = (\mathcal{S}, \texttt{init}, \texttt{V}, \texttt{set}, \texttt{get}, \texttt{doStep})$ where $\texttt{init} \in \texttt{init}_s$ is the initial state chosen according to $\mathcal{P}$ – by agreement for a non-stochastic FMU we let $\texttt{pick}(s) = 1$ where $\texttt{init}_s = \{s\}$ and $P(s') = 0$ for all other $s' \neq s$.

Notice here that our extension of FMUs with stochastic has the probabilistic choices resolved once; this construction is both practically and theoretically sound. In practice, such a construction can be ensured by using seeded pseudo-random number generators. Here the seed is chosen at random initially, leading to a subsequent determined execution. In a theoretical setting, similar generative constructions have been used to define the semantics of probabilistic programs – for instance for the semantics of the IBAL language as proposed by Pfeffer et al. in [24].

It is easy to see that the semantics of an FMU fits well within the stochastic semantics for NTA as the standard specifies a collection of (complex) update-functions over a state – which in turn can be encoded as a real. However, as FMUs have no notion of discrete-update labels, to correctly embed an FMU into an NTA, we shall in the next section describe one way of encapsulating an FMU within a single STA, such as to make it compatible with the NTA framework.

III. Extension of Uppaal

To facilitate the import of FMUs in Uppaal, we have extended Uppaal with a construct for loading dynamic C libraries at runtime. External C functions in Uppaal look and act exactly like regular functions in Uppaal and only their declaration differ by the additional import environment – an example of the new syntax can be seen in Figure 2. We shall here discuss the type-conversions between the C-like language in Uppaal and actual C. Furthermore we introduced the type string for string constants and the type ptr_t for holding pointers to external data—a type with its binary defined by, and dependent on, the hardware platform Uppaal is executing on and functionally equivalent to size_t known from regular C. Lastly, to facilitate single-initialization of external libraries, we have introduced the void __ON_CONSTRUCT__() and void __ON_DESTRUCT__() – that if existing, will be called upon model-initialization (and de-initialization respectively), but not for each proposition given to Uppaal. These can be defined at a global scope as well as in the scope of each individual TA.

Uppaal type	C type	By Value	Return	Array
bool	`bool`	✓	✓	✓
chan	`const char`			
clock	`double`			✓
double	`double`	✓	✓	✓
ptr_t	`size_t`	✓	✓	✓
int	`int32_t`	✓	✓	✓
string	`const char`			
<type>[]	<type>			

TABLE I: The type-conversion between Uppaal and C. The complex types `chan`, `string`, `clock` and array-types are sent in C-convention as pointers to raw memory and are thus forced to be sent as references. All types can be sent by reference, but the immutable types `chan` and `string` are forced `const`. Lastly, only single-dimension arrays are supported, and only of mutable types; arrays of `chan` and `string` are currently not supported.

A. Type Conversion

To maintain sanity during simulation some restrictions on the types being transferable between Uppaal and external functions are in place. Currently, the types bool, chan, clock, double, ptr_t, int and string can be used in external functions, omitting complex types constructed using the struct keyword as well as two, and more, level arrays. A chart of the type-conversion to C and other restrictions are given in Table I. We further emphasize some significant differences from linking directly between C programs and working calling external functions from Uppaal.

- A bool variable in Uppaal is either zero or one—as such, any function returning bool will return 0 if the C-function called returned 0, and 1 otherwise.
- const is enforced, implying that any variable sent as const, even if sent as an array or by reference, will not have its value changed in the Uppaal environment—regardless of the behavior of the C-function called.
- Each import-statement has a private scope, as exemplified by Figure 2.
- Integers in Uppaal can be given a bounded range. If this range is violated – either when values are sent by reference or returned – the model is said to have violated model-sanity, causing a run-time error.

Utilizing this extension of Uppaal and the FMI-standard, we will implement a co-simulation-algorithm directly as a timed automaton.

IV. FMI/FMU in Uppaal

The FMI/FMU standard, version 2.0 is a standard for conducting co-simulation between different simulation environments. While the standard also supports "Model Exchange", we shall here focus only on co-simulation.

To conduct co-simulation using the FMI/FMU standard, one needs only two components: (1) A master algorithm controlling the overview and coordination of the composed simulation and (2) one or more FMUs for the master-algorithm to coordinate and facilitate communication between. This approach shows its strength by outsourcing the heavy computation of the simulation to specialized tools while maintaining a global overview and coordination of the

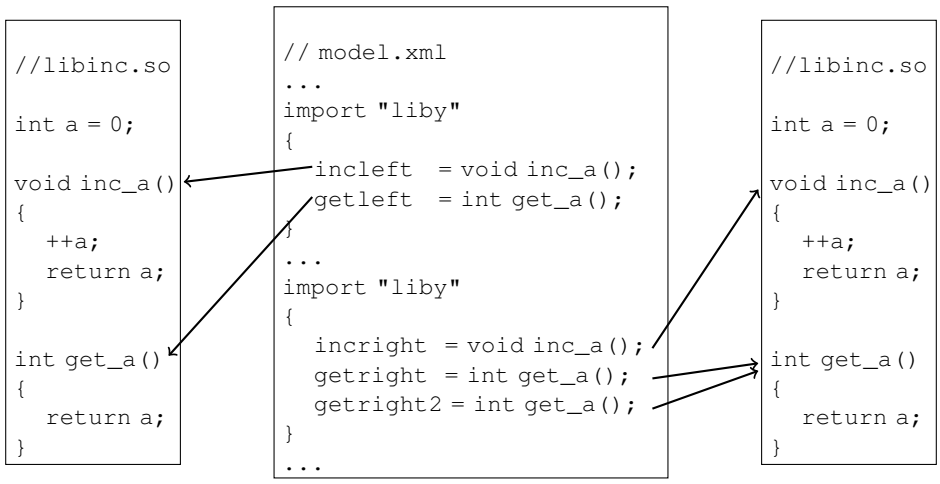


Fig. 2: Scoping rules of external libraries when loading into UPPAAL. Each import-statement creates a new environment – this implies that the variable a has two logical instances, a_{left} and a_{right} where incleft and getleft references a_{left} while incright, getright and getright2 references a_{right}.

composed system at any time. An example of such a composed system, using UPPAAL as a FMU for timed automata, was presented by Bogolomov et. al [5]. In general the MA is imposed on top of the system, enforcing a semantics particular to the given MA. In our work, we instead embed the external FMUs as timed automata. This allows us to reuse standard, well-defined, timed automata semantics, allowing us to construct a shallow MA, ensuring only that the FMI/FMU communication protocol is respected. In particular, from this approach, we adopt the so-called Interleaving Semantics.

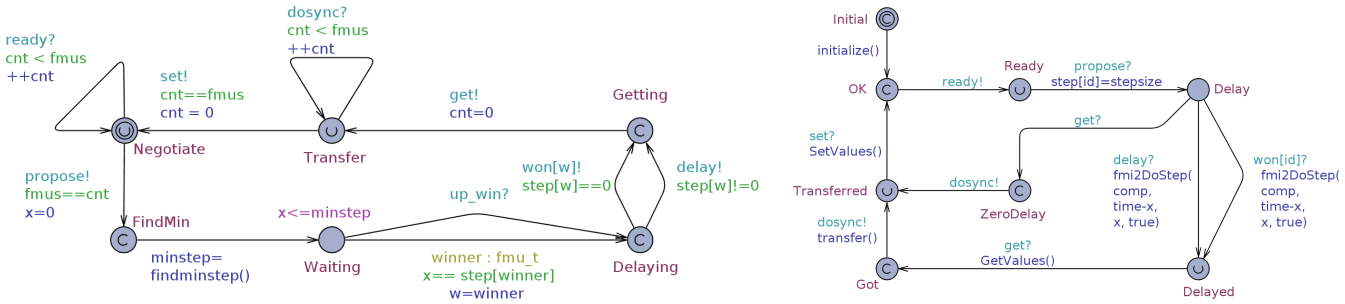
A. Master Algorithm and FMUs as Timed Automata

The TAs used for implementing the MA in UPPAAL and importing FMUs into UPPAAL are shown in Figure 3a and Figure 3b respectively. Let us walk through the computation of a single simulation step in the composed model. Initially, observe that

- the state of each FMU is encoded in the comp variable,
- time is a variable tracking the time progressed since the beginning of the simulation,
- x is a variable tracking the time since the last simulation step,
- step is an array containing all the proposed step-sizes and
- cnt is a variable tracking the number of FMUs that have completed a given stage of the MA – initially set to zero.

Initially the MA (depicted in Figure 3a) and each FMU (depicted in Figure 3b) Negotiate and Initial-states. As the MA is waiting for *ready* signals from the FMUs, all the FMUs will (in random order), call the initialize-function, abbreviating the setup function calls specified in the FMU standard. After initialization, each FMU will move from the OK location to the Ready location – and while doing so, synchronize with the MA on the *ready* channel. This forces the MA to wait until all FMUs have reached the Ready-location. At this point, the MA is forced to move from Negotiate to FindMin, synchronizing with all FMUs at once, s.t. each FMU moves from Ready to Delay. This makes each FMU propose the step-size (which could be computed by more complex functions, such as proposed by Bogolomov et. al [5]), after which it will wait for the MA to synchronize on either *delay*, *won*[*id*] or *get*. The MA will now, stepping from FindMin to Waiting, choose the minimal proposed step-size and let time progress by the given amount. Whenever time has progressed exactly minstep time-units, the MA will with even probability determine a "winner" in between FMUs proposing exactly minstep time-units as the delay. If the delay is non-zero, the MA will synchronize on the *delay* channel, in which case all the FMUs will end up in the Delayed location, ready to receive a synchronization on *get*. If the delay is zero, only the winning FMU will progress to the Delayed location while the remaining FMUs will await further synchronization. From the Delayed location, each FMU, upon synchronization with the MA on *get*, will call the appropriate getter-functions defined in the FMU-standard, synchronizing the data-arrays in UPPAAL with those in the external FMU. Notice here that if a zero-delay occurred, some FMU will have won, and all the losing FMUs will at this point in time move from the Delay to the ZeroDelay location – implying that data is not fetched from the losing FMUs as it cannot have changed since the last call to GetValues(). At this point in a simulation-step the MA has reached the Transfer location while all the FMUs are in the Got location. Here each FMU will, in random order according to a uniform distribution, transfer their local values (synchronized with the external FMU) to other FMUs in the system. The randomness here plays a key part in implementing interleaving semantics, as two FMUs can write to the same variable in a third FMU, effectively yielding a race-condition. Again, if we are in the special case of a zero-delay, only the winning FMU will transfer its values. Until all FMUs have moved to the Transferred location, the MA will wait in the Transfer location. Eventually the MA will be able to move from Transfer to the Negotiate location, triggering all the FMUs to move from Transferred to OK, pushing the updated values to the individual FMUs. This completes the cycle of a single step in the total simulation.

While our implementation here focuses on a specific MA, one can easily extend and test different MA algorithms within this framework. In [5], the authors restrict their MA to impose an ordering of the value-transfers between the FMUs – such a restriction could be implemented by imposing an order on each FMU, and checking if this order is respected for each FMU when synchronizing on *dosync*. In a similar manner, interoperability between different versions of the FMU/FMI standard can be achieved by adapting the general template in Figure 3b.



(a) The MA as a TA.

(b) The template of a single FMU.

Fig. 3: The TAs used for importing and simulating FMUs in UPPAAL.

V. CASE STUDY

To demonstrate our approach we shall construct a model of three small houses sharing a single heating unit. Each house is composed of two rooms which each individually can be heated. Heat can be transferred between the two rooms, but as the houses are placed apart, heat is not transferred between the houses. The shared heating unit is only capable of heating a single room at any point in time – furthermore, it takes some time for the heating unit to be transferred from one house to another.

A. How to model one house

Each single house is modeled using OPENMODELICA and entirely composed of standard components. The entire model of a house can be seen in Figure 4. Here each of the rooms have a heat-capacity of $2649600\frac{J}{K}$ and the wall between the rooms have a thermal conductivity of $6.4\frac{W}{K}$. At the same time, each of the rooms are affected by the outside temperature, here separated by slightly better insulated walls but with a larger surface-area with a thermal conductivity of $27.20\frac{W}{K}$ in total. As it can be seen from Figure 4, the house receives three inputs; `in_room1`, `in_room2`, and `in_outside` for the influence of the heater in either of the rooms and the influence of the (ever changing) outside temperature. As outputs, the two temperature-converters `troom1` and `troom2` give us the variables `out_room1` and `out_room2`. We will not consider the inner dynamics of the room, but simply view a room as a single mass with a heat-capacity. OPENMODELICA supports the export of models as Co-Simulation FMUs – in the case of our model from Figure 4, we get an FMU with three real-valued inputs `in_room1`, `in_room2` and `in_outside` (in Watts, Watts and degrees Celsius respectively) as well as the two outputs in degrees Celsius `out_room1` and `out_room2`.

B. A Controller as a Timed Automaton

The controller is implemented using UPPAAL timed automata and can be seen in Figure 5. Let us informally introduce the notation used for UPPAAL timed automata; circles denote *locations* and arrows denote *edges*. The key feature of the timed automata is the *clock* construction; variables that all progress at the same rate and track time. Each location can be labeled with an *invariant*, colored *purple* – a predicate that must evaluate true when the TA is in the given location. Locations can also be marked with `U` or `C` for *urgency* and *committed urgency* – forcing immediacy (and prioritized immediacy), implying that time cannot elapse when the TA is in the given location. In Figure 5 we can see that the Heating location is marked by a double-circle, indicating that it is the initial location. Edges can be

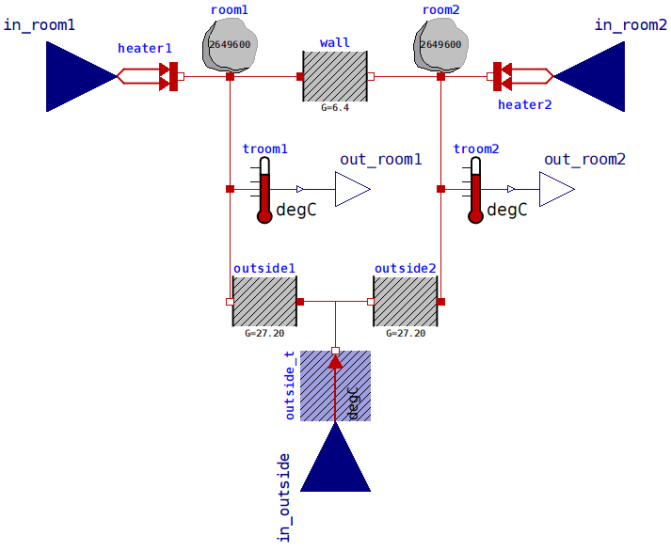


Fig. 4: A single house with two rooms modeled using OPENMODELICA.

labeled with `select`-statements (*yellow*), guards (*green*), updates (*blue*) and synchronizations (*turquoise*) where

- select-labels duplicate the single edge into multiple instances, one for each value possible in the declared type,
- guard-labels must evaluate to true for the edge to be admissible,
- update-labels can reset clocks and update variables declared using the `C`-language,
- synchronization-labels allows two or more TAs in the same system to move in unison when one is outputting (eg. *a*!) and multiple are inputting (*a*?) on the *channel a*.

Notice here that we only use the *broadcast* synchronization – for a given channel *a*, any TA is free to output *a*! (if allowed by a guard), and all other TAs which can receive *a*? must do so.

Aside from standard TA features, UPPAAL supports a `C`-like language for complex constructions – including variable-declarations, function-calls and such.

The controller shown acts in a bang-bang manner; given a target-temperature for each room (the tgt array), it randomly assigns the heating unit (the heat array) to any room with a temperature (the temp array) lower than or equal the target-temperature (right edge). If all rooms have reached their target-temperature, we can take the left edge, heating no room. When taking the right edge, if the controller

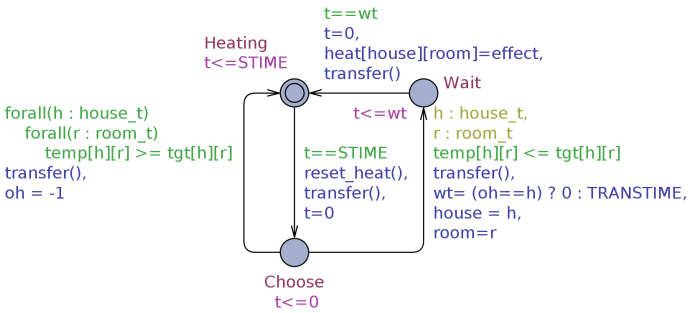


Fig. 5: The bang-bang controller implemented in UPPAAL. The controller samples the values of the system each 60 time units and chooses at random to heat a room in a house where the measured temperature is lower than the set-temperature of the given room (right-hand side transition). If no such exist, no room is heated (left-hand side transition). Whenever a *room* and *house* is chosen, if the house differs from previous choice, the controller will wait TRANSTIME between heating one room and another.

decides to change the house being heated, a delay will incur between turning of the heater in one house and turning on the heater in another house. This is controlled by the variable wt which forces our controller to let time progress in the Wait-location (due to the combination of the invariant $t <= wt$ and the guard $t == wt$) – but only if the chosen house differs, determined by the in-line if $wt = (oh == h)?0 : TRANSTIME$. Here one can also see the *select*-statement in action; an edge is created for each value in the type *house_t* ($\mathtt{h} \in \{1, 2, 3\}$) and each value in *room_t* ($\mathtt{r} \in \{1, 2\}$) – allowing for a concise description.

C. Composition of models

With the house and controller modeled, we can now focus on composing the entire system. Our system, as illustrated by Figure 6, aside from three houses and the controller includes the weather – giving the outside temperature. For simulating the weather, we here assume a simple sinus-curve with a frequency of 24 hours, oscillating between 4 and 20 degrees Celsius. We can also see from Figure 6 that the houses have no direct interaction with each other, and are only indirectly communicating via the temperatures they report and the choices of the controller.

Each of these individual components (even UPPAAL models [5]) can be exported as Co-Simulation FMU's – and thus composed into our system from Figure 6 using existing tools. However, as our initial controller exhibits randomness, we would like to answer the questions: *What is the expected minimal and maximal temperatures of any room?* and: *If a room becomes too cold, will it become heated again within two hours?*

These questions both contain probabilistic measures over continuous time – and thus are not answerable by current tools. To remedy this, we below propose an extension of the Statistical Modelchecker UPPAAL SMC that allows for dynamically linking and calling arbitrary C-functions from inside the tool – a feature that facilitates Co-Simulation of FMU's.

VI. EXPERIMENTS

This section presents the experiments that we perform on our case study when using the controller presented in Figure 5. In this example, all the components are exported into FMUs and recomposed in UPPAAL given the framework presented in Section IV – including

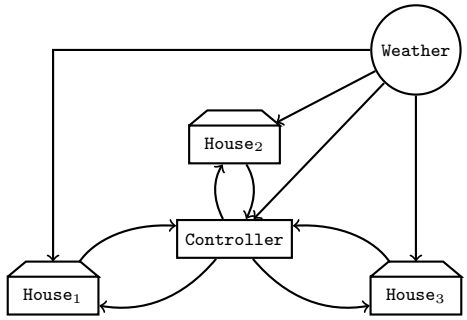


Fig. 6: An informal overview of the composed system. The arrows indicate the flow of information.

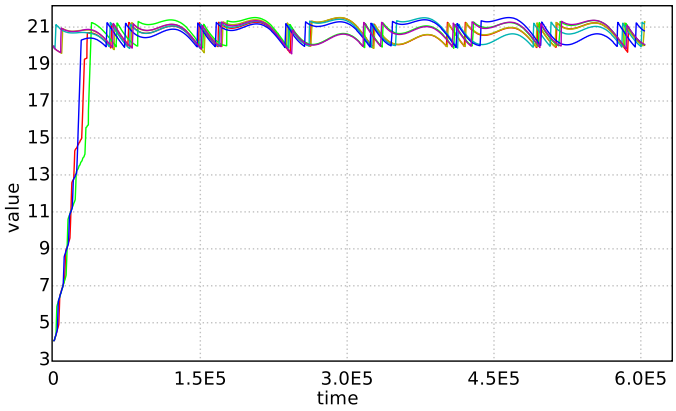


Fig. 7: A single simulation of the system. Each color represents the temperature of a room. The starting temperature for the first room in each house is 20 degrees Celcius and 4 degrees Celsius for the second room.

the controller. We shall consider the model presented in Section V and demonstrating three different features of UPPAAL SMC; Simulation, Estimation and Statistical Modelchecking.

The composed model, FMUs and extended version of UPPAAL SMC for 64-bit Linux-systems can be found at http://people.cs.aau.dk/~pgj/UPPAAL_FMIFMU.zip. Notice that the extended version of UPPAAL SMC includes functionality to export timed automata as FMUs as presented by Bogolomov et al. [5]. We also provide a prototype tool for embedding FMUs into a UPPAAL-importable timed automata for ease of use.

A. Simulation

In Figure 7 one can observe the results of executing following query in UPPAAL

```
simulate 1 [<=3600*24*7]{h1_output[0], h1_output[1],
                         h2_output[0], h2_output[1],
                         h3_output[0], h3_output[1]}
```

This proposition monitors the temperatures of each house over a period of seven days for a single simulation. As expected, initially our controller rapidly heats the coldest rooms in all of the houses. We can also observe that the temperature of the rooms after the first day is kept within a 19 to 22 degrees window in the current setting.

B. Estimation

However, such a single simulation does not quantify over the probabilistic behavior of our system – one might be interested in knowing

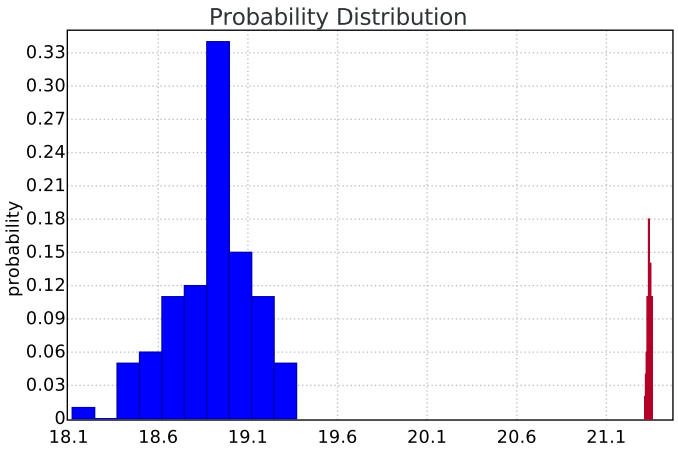


Fig. 8: The super-imposed probability distributions of the expected minimal and maximal temperatures over all the rooms, over a week, disregarding the first day, based on 100 samples. The x-axis is given in degrees Celsius. Blue indicates minimal expectation while red indicates maximal. The mean of the minimal expectation is 18.9 while the mean of the maximal expectation is 21.3.

the expected maximal and minimal temperatures (disregarding the first day as our rooms are in an abnormal state).

Such a measure can be achieved by the following propositions.

```
E[<=3600*24*7;100] (min: mintemp())
E[<=3600*24*7;100] (max: maxtemp())
```

Here mintemp and maxtemp are functions defined in UPPAAL C computing the minimum and maximum of the temperatures of all the rooms. The proposition computes the estimated minimal (resp. maximal) value of the expression over the course of a weeks simulation – and it does so on the basis of 100 simulations.

As we can see in Figure 8, the performance of our constructed controller is fairly stable. Both the minimal and maximal temperature of any room stays (on average) within 18.9 degrees and 21.3 degrees Celsius. However, while the average peak temperature is stable at 21.3 degrees with less than a tenth of a degree span on the observed values, the minimal temperature over a week was observed to vary by more than a degree across all runs.

C. Statistical Modelcheking

While the estimation and simulation propositions provide some quantitative measure on different metrics of the system, we can use the statistical model-checking features of UPPAAL SMC to reason on the probabilistic behavior of our system.

Let us try to quantify the fairness of our controller; what is the probability that the temperature difference between the coldest room and the hottest room is greater than two degrees Celsius after the first day? This we can express as follows.

```
Pr[<=3600*24*7] (<>time > 3600*24 &&
                 (maxtemp() - mintemp()) >= 2)
```

As we can observe in Figure 9, there is a fairly high chance (more than 50%) that the temperature difference between the coldest and hottest room will grow beyond two degrees within a week.

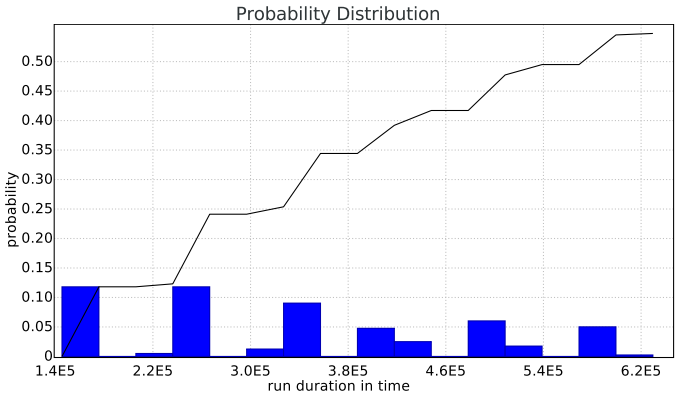


Fig. 9: The probability distribution of having only a two-degree difference between the coldest and warmest room with the cumulative probability superimposed (black line). The measure is obtained using the statistical parameters $\epsilon = 0.05$ and $\alpha = 0.05$ – yielding 398 samples needed by UPPAAL SMC. With a 95% confidence the true probability lies within $[0.50, 0.60]$ of having a greater than 2-degrees difference at any point within a week.

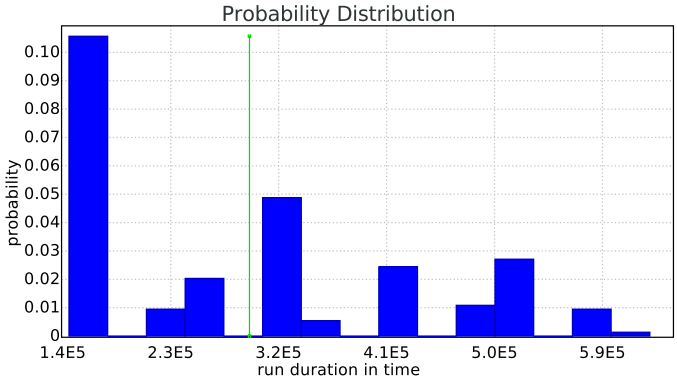


Fig. 10: The probability distribution of not recovering from a threshold violation. The statistical parameters are $\epsilon = 0.05$ and $\alpha = 0.05$ – yielding 738 samples needed by UPPAAL SMC. With a 95% confidence the true probability lies within $[0.21, 0.31]$ of not recovering.

We can also construct more complex propositions using the temporal MITL-logic. As we already know, the set-temperature cannot be respected in general in our example – however, we might be able to accept deviations as long as they are not for extended periods of time. We therefore construct the following proposition capturing: *after the first day, can it ever happen that if the second room of the first house has a temperature below 20 degrees for more than two hours at a given time*.

```
Pr (<> [86400,604800]( [] [0,7200] (h1_output[1] < 20)))
```

We can observe in Figure 10 that this property is not surely satisfiable. Already in the second day of the simulation there is more than 10% chance that the controller will not be able to recover from a threshold-violation within two hours. We can also observe that this tendency seems to be repeating every night of our simulation.

VII. CONCLUSION

In this paper we demonstrated the ease with which FMI-FMU models exported from other tools can be integrated into the setting of

UPPAAL SMC. We provide the flexibility of modeling different Master Algorithms (MAs) in a sound semantical framework facilitating probabilistic and temporal reasoning.

We believe that the possibility of using already existing domain models is essential in order to facilitate the use of formal methods in industrial systems, as the correct re-modeling of entire systems is a very time consuming exercise. The expressiveness of the modeling language available in UPPAAL SMC allows for the efficient modeling of real-world scenarios with inherently uncertain and time-dependent behaviour – such as signal noise, human interaction and general natural phenomena. Thus this paper aims to make recent advances in statistical model checking and statistical validation of systems available for use in an industrial setting.

VIII. Future Work

Adding the possibility of calling arbitrary C-libraries during statistical simulation opens up for a number of new applications. This is true both for UPPAAL SMC and the related tool UPPAAL STRATEGO.

We consider the option of utilizing UPPAAL STRATEGO to perform controller synthesis for heterogeneous FMI-FMU systems as the most promising direction. This would make it much easier to generate optimized controllers using machine learning for systems being developed using other modeling tools. With the current approach all components in the system must be re-modeled in UPPAAL STRATEGO.

Calling external C-libraries can also be applied within the domain of classical model-checking in UPPAAL. Here there is the added restriction that the external calls can only be used to update the discrete variables and not the clock variables. On top of *statelessness* we also require that the external FMUs are strictly *deterministic* for the model checking to be semantically sound. For application of this in a non FMI-FMU setting see [9].

References

[1] W. Ahmad, M. Jongerden, M. Stoelinga, and J. v. d. Pol. Model checking and evaluating QoS of batteries in MPSoC dataflow applications via hybrid automata. In *2016 16th International Conference on Application of Concurrency to System Design (ACSD)*, pages 114–123, June 2016.

[2] A. Arnold, M. Baleani, A. Ferrari, M. Marazza, V. Senni, A. Legay, J. Quilbeuf, and C. Etzien. An application of SMC to continuous validation of heterogeneous systems. In *Proceedings of the 9th EAI International Conference on Simulation Tools and Techniques*, SIMUTOOLS'16, pages 76–85, ICST, Brussels, Belgium, Belgium, 2016. ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering).

[3] D. Basile, F. Di Giandomenico, and S. Gnesi. Statistical model checking of an energy-saving cyber-physical system in the railway domain. In *Proceedings of the Symposium on Applied Computing*, SAC '17, pages 1356–1363, New York, NY, USA, 2017. ACM.

[4] G. Behrmann, A. David, K. G. Larsen, J. Hakansson, P. Petterson, W. Yi, and M. Hendriks. UPPAAL 4.0. In *Quantitative Evaluation of Systems, 2006. QEST 2006. Third International Conference on*, pages 125–126. IEEE, 2006.

[5] S. Bogomolov, M. Greitschus, P. G. Jensen, K. G. Larsen, M. Mikučionis, T. Strump, and S. Tripakis. Co-simulation of hybrid systems with SpaceEx and Uppaal. In *Proceedings of the 11th International Modelica Conference, Versailles, France, September 21-23, 2015*, number 118, pages 159–169. Linköping University Electronic Press, 2015.

[6] D. Broman, C. Brooks, L. Greenberg, E. A. Lee, M. Masin, S. Tripakis, and M. Wetter. Determinate composition of FMUs for co-simulation. In *Proceedings of the Eleventh ACM International Conference on Embedded Software*, EMSOFT '13, pages 2:1–2:12, Piscataway, NJ, USA, 2013. IEEE Press.

[7] P. Bulychev, A. David, K. G. Larsen, M. Mikučionis, D. B. Poulsen, A. Legay, and Z. Wang. UPPAAL-SMC: Statistical model checking for priced timed automata. *arXiv preprint arXiv:1207.1272*, 2012.

[8] P. E. Bulychev, A. David, K. G. Larsen, A. Legay, G. Li, and D. B. Poulsen. Rewrite-based statistical model checking of WMTL. *RV*, 7687:260–275, 2012.

[9] F. Cassez, P. G. de Aledo, and P. G. Jensen. *WUPPAAL: Computation of Worst-Case Execution-Time for Binary Programs with UPPAAL*, pages 560–577. Springer International Publishing, Cham, 2017.

[10] F. Cremona, M. Lohstroh, D. Broman, M. D. Natale, E. A. Lee, and S. Tripakis. Step revision in hybrid co-simulation with FMI. In *2016 ACM/IEEE International Conference on Formal Methods and Models for System Design, MEMOCODE 2016, Kanpur, India, November 18-20, 2016*, pages 173–183. IEEE, 2016.

[11] A. David, D. Du, K. G. Larsen, A. Legay, M. Mikučionis, D. B. Poulsen, and S. Sedwards. Statistical model checking for stochastic hybrid systems. *arXiv preprint arXiv:1208.3856*, 2012.

[12] A. David, K. Larsen, A. Legay, M. Mikučionis, D. Poulsen, and S. Sedwards. Runtime verification of biological systems. *Leveraging Applications of Formal Methods, Verification and Validation. Technologies for Mastering Change*, pages 388–404, 2012.

[13] A. David, K. G. Larsen, A. Legay, M. Mikucionis, and D. B. Poulsen. Uppaal SMC tutorial. *International Journal on Software Tools for Technology Transfer*, 17(4):397, 2015.

[14] 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 *FORMATS*, volume 6919 of *LNCS*, pages 80–96, 2011.

[15] A. David, K. G. Larsen, A. Legay, U. Nyman, and A. Wasowski. Timed I/O automata: a complete specification theory for real-time systems. In *Proceedings of the 13th ACM International Conference on Hybrid Systems: Computation and Control, HSCC 2010, Stockholm, Sweden, April 12-15, 2010*, pages 91–100, 2010.

[16] FMI Standard Orginization. FMI support in tools. http://fmi-standard.org/tools/.

[17] O. Gadyatskaya, R. R. Hansen, K. G. Larsen, A. Legay, M. C. Olesen, and D. B. Poulsen. Modelling attack-defense trees using timed automata. In *International Conference on Formal Modeling and Analysis of Timed Systems*, pages 35–50. Springer International Publishing, 2016.

[18] S. Guermazi, S. Dhouib, A. Cuccuru, C. Letavernier, and S. Gérard. Integration of UML models in FMI-based co-simulation. In *Proceedings of the Symposium on Theory of Modeling & Simulation*, TMS-DEVS '16, pages 7:1–7:8, San Diego, CA, USA, 2016. Society for Computer Simulation International.

[19] C. Jegourel, A. Legay, and S. Sedwards. A platform for high performance statistical model checking–PLASMA. *Tools and Algorithms for the Construction and Analysis of Systems*, pages 498–503, 2012.

[20] J. H. Kim, A. Boudjadar, U. Nyman, M. Mikucionis, K. G. Larsen, and I. Lee. Quantitative schedulability analysis of continuous probability tasks in a hierarchical context. In *Component-Based Software Engineering (CBSE), 2015 18th International ACM SIGSOFT Symposium on*, pages 91–100. IEEE, 2015.

[21] Object Management Group. fUML. http://www.omg.org/spec/FUML/.

[22] M. Pazold, S. Burhenne, J. Radon, S. Herkel, and F. Antretter. Integration of Modelica models into an existing simulation software using FMI for co-simulation. In *Proceedings of the 9th International MODELICA Conference; September 3-5; 2012; Munich; Germany*, number 76, pages 949–954. Linkping University Electronic Press; Linkpings universitet, 2012.

[23] N. Pedersen, T. Bojsen, J. Madsen, and M. Vejlgaard-Laursen. FMI for co-simulation of embedded control software. In *The First Japanese Modelica Conferences, May 23-24, Tokyo, Japan*, number 124, pages 70–77. Linkping University Electronic Press, Linkpings universitet, 2016.

[24] A. Pfeffer. IBAL: A probabilistic rational programming language. In *In Proc. 17th IJCAI*, pages 733–740. Morgan Kaufmann Publishers, 2001.