Model Checking FMI Co-simulation Using Timed Automata *

Kaiqiang Jiang, Chunlin Guan, Jiahui Wang, and Dehui Du*

Shanghai Key Laboratory of Trustworthy Computing,
School of Computer Science and Software Engineering, East China Normal University, Shanghai, China
{dhdu}@sei.ecnu.edu.cn

Abstract. Cyber-physical Systems(CPSs) focus on the coupling of cyber part viewed as distributed computation units and physical part covering the environment affecting the running of the system. CPSs are often treated modularly to tackle both complexity and heterogeneity. The verification of CPSs may be done modularly by Functional Mock-up Interface (FMI) co-simulation. However, the master algorithm for co-simulation may be livelock or deadlock. The architectural modelling of CPSs may introduce an algebraic loop which is a feedback loop resulting in instantaneous cyclic dependencies. To solve these problems, we propose a novel approach for model checking several properties of FMI co-simulation such as deadlock, liveness, reachability. We model the architecture of CPSs with SysML block diagrams, which captures the dependence of Functional Mock-up units (FMUs) and the orchestration of the master algorithm. Next, we encode FMU components and three various master algorithms with timed automata separately. Finally, we verify the correctness of the co-simulation and the absence of algebraic loops in the architecture with UPPAAL. To illustrate the feasibility of our approach, the case study water tank is presented. The results show that our approach helps model checking FMI co-simulation.

Keywords: Co-simulation, Master algorithm, Functional Mock-up Interface, Timed automata, Model checking

1 Introduction

Cyber-physical systems (CPSs) are integration of computation with physical processes whose behavior is defined by both computational and physical parts of the system [26]. Embedded computers and networks monitor and control the physical processes, usually with feedback loops where physical processes affect computations and vice versa. The heterogeneity is one of the main characteristics of CPSs. The components of CPSs are of various types, requiring interfacing and interoperability across multiple platforms and different models of computation. Verification of CPSs as a whole requires the use of heterogeneous simulation environments. One emerging industry standard is the Functional Mock-up Interface (FMI) [6]. It is a standard designed to support simulation of complex systems comprising heterogeneous components, by coupling the different models with their own solver in a co-simulation environment. A master algorithm(MA)[1] provides the orchestration of the entire co-simulation. However, the master algorithm is not part of the FMI standard. This implies that the user or tool vendor needs to develop a sophisticated orchestration algorithm for the problem at hand. The correctness of the master algorithm also needs to be analysed. In this paper we explicitly model the co-simulation of CPSs to model checking several properties of co-simulation such as deadlock, liveness, and reachability. To achieve the goal, we present a novel approach to model and check the properties of co-simulation with timed automata [2]. The schematic view of our approach is shown in Fig.1. At the design phase, we construct the architecture of CPSs with SysML block diagrams [22]. Each block represents a component and the communication between components is modeled with SysML connector. To simulate the whole system with co-simulation techniques, the block can be modeled with a Functional Mock-up (FMU) and the connector can be modeled with a master algorithm. The MA orchestrate FMUs to accomplish the communication between FMUs. To verify the correctness of the architecture, we encode the FMU and the master

* This work was supported by NSFC (Grant No.61472140, 61202104) and NSF of Shanghai (Grant No. 14ZR1412500).

algorithm by timed automata, which facilitates the verification livelock or deadlock with the UPPAAL model checker [5].

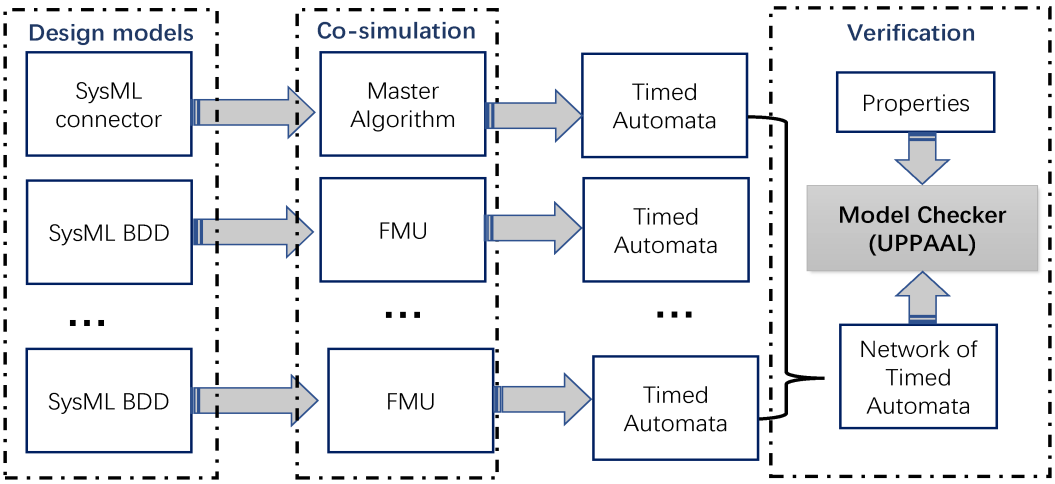


Fig. 1. A schematic view of our approach.

Contributions. Our contributions are as follows:

- We present a novel approach to model checking several properties of the co-simulation, such as livelock, deadlock and reachability. Moreover, we propose to check the absence of algebraic loops of SysML architectural model.
- We model the master algorithms for co-simulation with timed automata. Three various master algorithms are analysed with model checking. Besides, FMU is also modeled with timed automata and the orchestration between FMUs is modeled with a network of timed automata. With the help of UPPAAL, we analyse the reachability, livelock and deadlock of the architecture.
- We present how to model the architecture of CPSs with SysML block diagrams. We deal with the heterogeneity of CPSs by supporting multi-modelling and co-simulation based on FMI standard.
- The prototype for model checking co-simulation of CPSs is developing, which is integrated in our Mondana platform [11](https://github.com/ECNU-MODANA/AL-Modana.git). We have developed the SysML modelling environment and the *co-simulator* for simulating CPSs[16].

The remainder of this paper is organized as follows. In Section 2, we briefly review the technical background including FMI, FMU and timed automata, and then provide encoding FMU by timed automata. Section 3 encodes three master algorithms with timed automata and verify the livelock and deadlock of three master algorithms. Section 4 presents the architecture modelling with SysML block diagrams. Section 5 presents the model checking FMI co-simulation with the example water tank. The experimental results show that our approach is feasible and useful. Finally we position our work with respect to related work before concluding and discussing possible future extensions.

2 Background

We give some background on three main ingredients of our work: FMI 2.0, FMU and timed automata. Then, we propose to encode FMUs by timed automata.

2.1 FMI 2.0

FMI [10] defines an standard applied to composing model components which are designed with different modeling tools. The FMI standard was first developed in the MODELISAR project started in 2008 and supported by a large number of software companies and research centers. FMI offers the means for model based development of systems and is particularly appropriate way to develop complex CPSs. The FMI standard supports both co-simulation and model exchange. However, in this paper, we focus on the co-simulation in FMI 2.0. Compared with the FMI 1.0, there are two important additions: $fmiGetFMUstate$ and $fmiSetFMUstate$, which allow the master to copy and later restore the complete state of an FMU slave. The two functions provide an ordinary mechanism for rollback, however, they are not practical for the reason that it's infeasible to restore so many states for a huge system. The function $fmiDoStep$ is to control the computation of time steps, called by the MA to orchestrate the exchange of data among the FMUs during the entire co-simulation.

$fmiStatus\ fmiDoStep($
$\quad fmiComponent\ c,$
$\quad fmiReal\ currentCommunicationPoint,$
$\quad fmiReal\ communicationStepSize,$
$\quad fmiBoolean\ noSetFMUStatePriorToCurrentPoint);$

The $currentCommunicationPoint$ argument is the current communication time of the master. The $communicationStepSize$ is the time step that the master proposes to the slave. Given a time step, the slave may accept or reject the step. For example, it may reject it because the step size is so large that causes a discrete event within the step. If the argument $noSetFMUStatePriorToCurrentPoint$ is true, the master will no longer call $fmiSetFMUstate$ for time instants prior to $currentCommunicationPoint$ in this simulation run.

2.2 FMU

FMU is the model component which implements the methods defined in the FMI API [25].

Definition 1. ***FMU syntax*** *We can look back on the definition of FMU. An FMU is a tuple* $F = (S, U, Y, D, s_0, set, get, doStep)$*, where:*

- S denotes the set of state valuations of F.
- U denotes the set of input port variables of F. Note that an element $u \in U$ is a variable, not a value, which ranges over a set of values $\mathbb{V}$.
- Y denotes the set of output port variables of F. Each $y \in Y$ ranges over the same set of values $\mathbb{V}$.
- $D \subseteq U \times Y$ denotes a set of input-output dependencies. $(u, y) \in D$ means that the output y is directly dependent on the value of u. The I/O dependency information is used to ensure that a network of FMUs does not contain cyclic dependencies, and also to identify the order in which all variables are computed during a step.
- $s_0 \in S$ denotes the initial state of F.
- $set : S \times U \times \mathbb{V} \to S$ denotes the function that sets the value of an input variable. Given current state $s \in S$, input variable $u \in U$, and value $v \in \mathbb{V}$, it returns the new state obtained by setting u to v.
- $get : S \times Y \to \mathbb{V}$ denotes the function that returns the value of an output variable. Given state $s \in S$ and output variable $y \in Y$, $get(s, y)$ returns the value of y in s.
- $doStep : S \times \mathbb{R}_{\geqslant 0} \to S \times \mathbb{R}_{\geqslant 0}$ denotes the function that implements one simulation step. Given current state s, and a non-negative real value $h \in \mathbb{R}_{\geqslant 0}$, $doStep(s, h)$ returns a pair (s', h') such that:
  When $h' = h$, it indicates that F accepts the time step h and reaches the new state s';
  When $0 \leqslant h' < h$, this means that F rejects the time step h, but making partial progress up to h', and reach the new state s'.

Definition 2. ***FMU semantics*** *Consider an FMU* $F = (S, U, Y, D, s_0, set, get, doStep)$.

The behavior of F depends on functions such as *doStep*. So the behavior of F is a function of a timed input sequence (TIS). A TIS is an infinite sequence $v_0h_1v_1h_2v_2h_3...$ of alternating input assignments v_i, and time delays h_i. An input assignment is a function $v : U \to \mathbb{V}$. That is, v assigns a value to every input variable in U. A TIS like the above defines a run of F, which is an infinite sequence of quadruples (t, s, v, v'), where $t \in \mathbb{R}_{\geqslant 0}$ is a time instant, $s \in S$ is a state of F, v is an input assignment, and $v' : Y \to \mathbb{V}$ is an output assignment : $(t_0, s_0, v_0, v'_0)(t_1, s_1, v_1, v'_1)(t_2, s_2, v_2, v'_2)...$ defined as follows:

- $t_0 = 0$ and s_0 is the initial state of F.
- For each $i \geqslant 1$, $t_i = t_0 + \sum_{k=1}^{i} h_k$
- Given the current state s_i, the function *set* is used to set all input variables to the values specified by v. Then F reaches a new state s'_i. Next, the function *get* is used to get the values of all output variables v'_i.
- We assume that $doStep(s'_i, h_{i+1}) = (s_{i+1}, h_{i+1})$ based on the assumption that every h_i is accepted by F, and results in the next state s_{i+1}.

2.3 Timed Automata

Timed automata [5] is a theory to model the behavior of real-time systems. Its definition provides a powerful way to annotate state-transition graphs with many real-valued clocks. In this section, we introduce the syntax and semantics of timed automata.

Definition 3. ***Timed automata syntax*** *A timed automaton is a tuple $A = (L, X, l_0, E, E_O, E_I, I)$, where:*

- L is a finite set of locations;
- X is a finite set of clocks;
- $l_0 \in L$ is an initial location;
- The set of guards $G(x)$ is defined by the grammar $g := x \bowtie c \mid g \wedge g$, where $x \in X$, $c \in \mathbb{N}$ and $\bowtie \in \{<, \leqslant, \geqslant, >\}$. $E \subseteq L \times G(X) \times 2^X \times L$ is a set of edges labelled by guards and a set of clocks to be reset;
- E_O is a set of input events.
- E_I is a set of output events.
- $I : L \to G(X)$ assigns invariants to clocks.

A clock valuation is a function $v : X \to \mathbb{R}_{\geqslant 0}$. If $\delta \in \mathbb{R}_{\geqslant 0}$, then $v + \delta$ denotes the valuation such that for each clock $x \in X$, $(v + \delta)(x) = v(x) + \delta$. If $Y \subseteq X$, then $v[Y := 0]$ denotes the valuation such that for each clock $x \in X$ Y, $v[Y := 0](x) = v(x)$ and for each clock $x \in Y$, $v[Y := 0](x) = 0$. The satisfaction relation $v \models g$ for $g \in G(x)$ is defined in the natural way.

Definition 4. ***Timed automata semantics*** *The semantics of a timed automaton $A = (L, X, l_0, E, E_O, E_I, I)$ is defined by a transition system $S_A = (S, s_0, \to)$,*

where $S = L \times \mathbb{R}^X_{\geqslant 0}$ is the set of states, $s_0 = (l_0, v_0)$ is the initial state, $v_0(x) = 0$ for all $x \in X$, and $\to \subseteq S \times S$ is the set of transitions defined by :

- $(l, v) \xrightarrow{\in(\delta)} (l, v + \delta)$ if $\forall 0 \leqslant \delta' \leqslant \delta : (v + \delta') \models I(l)$;
- $(l, v) \to (l', v[Y := 0])$ if there exists $(l, g, Y, l') \in E$ such that $v \models g$ and $v[Y := 0] \models I(l')$.

The reachability problem for an automaton A and a location l is to decide whether there is a state (l, v) reachable from (l_0, v_0) in the transition system S_A. As usual, for verification purposes, we define a symbolic semantics for timed automata. For universality, the definition uses arbitrary sets of clock valuations.

2.4 Encoding FMUs by timed automata

Given an FMU $F = (S, U, Y, D, s_0, set, get, doStep)$, we model this FMU by a timed automaton $A = (L, X, l_0, E, E_O, E_I, I)$, where:

- L is a set of finite locations. Note that a location of TA has the same form as a state of the F.
- The initial location l_0 is such that s is set to s_0.
- Each input variable $u \in U$ ranges over $E_I \cup \{absent\}$.
- Each output variable $y \in Y$ ranges over $E_O \cup \{absent\}$.
- An input event in $e \in E_O$ is such that the function *set* of F sets the input variable u to a given value.
- An output event in $e \in E_I$ indicates that the function *get* of F gets the output variable y. The set of values in the E_I is the Y.
- The communication with input and output events in TA is the same as the I/O dependencies information in FMU. $(u, y) \in D$ denotes that output y depend on input u. The output events also depend on the input events.
- For any $e \in E$ of A, there is a transition $l \xrightarrow{e} l'$, which may be found after the function *doStep* is executing. For instance, if there is a transition $l \xrightarrow{e} l'$, at the same time $doStep(s, h)$ may be called which indicates that F accepts the time step h and reaches the new state s'. However, F maybe rejects the time step for the reason which we have discussed above and this process can also be described in the TA. If there is a rollback behavior happens after the F rejects the time step h, the transition in TA could be a edge $l \xrightarrow{e} l$, which denotes that a location travels to itself.

Although there are semantic gaps between FMUs and timed automata, we provide a appropriate transformation above to solve the problem. So, we believe that it's feasible to encode the FMUs by timed automata. In the following sections, we will model the FMUs with the timed automata in UPPAAL.

3 Co-simulation Execution

In this section, we use timed automata to encode three master algorithms and verify some expected properties of co-simulation such as deadlock, liveness and reachability with UPPAAL.

3.1 I/O Dependency Information

When it comes to co-simulation, I/O dependency information [10] is inevitably required to be well considered. The master algorithm calls function *Set* to provide input value to an FMU and function *Get* to retrieve an output value. So it is essential to know which outputs of an FMU depend immediately on which inputs. In the design of a MA, the direct dependency information can be used to call the function *Set* and *Get* in a well-defined order. In FMI 2.0 this information can be provided using the element $ModelStructure$ [7]. However, sometime there may be an algebraic loop in the dependency information, which may not converge. Since we are interested in non-diverging and deterministic composition of FMUs, we need to distinguish these two cases.

3.2 Master Algorithm (MA)

The master algorithm is to orchestrate the execution of different subsystems. Each subsystems serves as an FMU block whose simulation is triggered by a particular MA. FMUs can be seen as black boxes comprising input. The process of simulation is divided into several steps where each FMU can be simulated independently until it needs to exchange data or implement synchronization. Master algorithms commonly are illustrated as following.

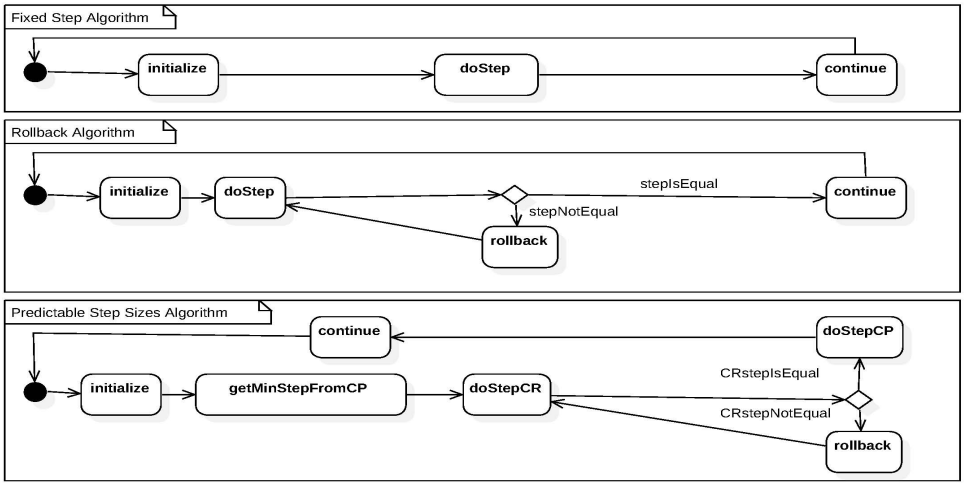


Fig. 2. Activity diagrams for three master algorithms

Fixed Step Algorithm In fixed step algorithm, all FMUs have the same step size. When master algorithm calls *doStep* with the step size h, time will advance from a communication point t to the next communication point $t + h$. During the simulation step, an FMU with it's own solver will simulate independently according to its input value and generate a running result as output value. MA will wait until all FMUs finish their simulation step and then get their output values to exchange data for preparing next simulation step. The activity diagram of fixed step algorithm is illustrated in the top of Fig.2. There are mainly three activities in the control flow: *initialize*, *doStep* and *continue*.

Rollback Algorithm In the fixed step algorithm [10], the process can maintain correct when all FMUs are reliable. When some error happens during a simulation step, the process will be affected after the wrong simulation step. Then we need the FMU rollback mechanism. There are some important features proposed in the FMI 2.0. We can save the FMU state if necessary and the state we saved in the process can be restored. For example, MA calls *doStep* on FMU_1 and FMU_2 while FMU_1 can accept the request and FMU_2 can reject it. If we save the state of FMU_1 and FMU_2 at the communicating point t, we can restore the scene after FMU_2 rejects *doStep*. Under this circumstances, we need all FMUs support rollback. The activity diagram of rollback algorithm is clearly shown in Fig.2. Compared with the fixed step algorithm, all FMUs are required to support *rollback*, that is, all FMUs need to return to the previous state if the simulation steps of all FMUs are not equal.

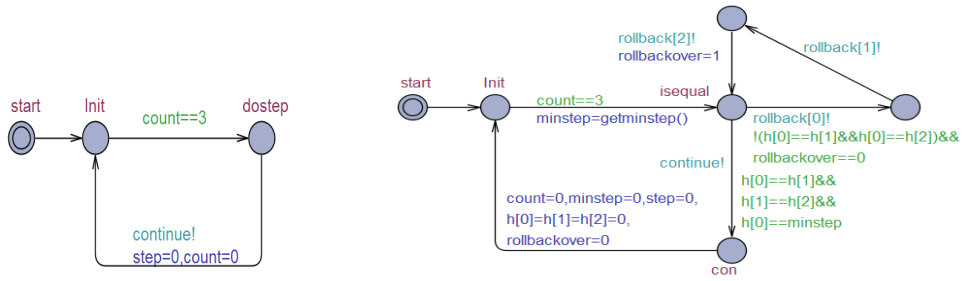
Predictable Step Sizes Algorithm The rollback algorithm require all FMUs support rollback. However, this approach is inefficient in many cases. The function $GetMaxStepSize$ was introduced to optimize the performance of rollback algorithm. This function return the maximum step size and state flag of a predictable FMU. Maximum step is the largest step that a predictable FMU can perform. State flag includes *ok*, *discard* and *error*. *OK* denotes the predictable FMU can accept the simulation step size. *Discard* denotes the predictable FMU only implement partial step during simulation. *Error* denotes the predictable FMU can't continue the simulation because of its unacceptable state or unreasonable input value. Also, when *discard* and *error* happen, the FMU need to rollback to the previous state saved before. Whether an FMU is a predictable FMU or not should be indicated in FMU's *xml* file. Moreover, if an FMU supports rollback and predictable step size at the same time, the predictable step size algorithm only uses predictable ability to get the maximum step of a predictable FMU. On the other hand, a predictable FMU can accept any step size less than or equal to the maximum step returned by $GetMaxStepSize$.

First off, the master algorithm get the maximum step of all predictable FMUs and find the smallest communication step size h that all predictable can accept. Then, we save the states of

all FMUs supporting rollback. MA calls $doStep(h)$ on FMUs supporting rollback. The function $doStep()$ will return the real performed step size. If all performed step sizes are equal to h, MA will call $doStep(h)$ on FMUs supporting predictable. Otherwise, MA will find the smallest performed step h_{min}, then all rollback FMUs will restore the state saved before the simulation step. Finally, MA will call $doStep(h_{min})$ on all FMUs. The control flow of predictable step size algorithm is shown in Fig.2. For example, $getMinStepFromCP$ is an activity that MA will call $GetMaxStepSize$ on all predictable FMUs to find their maximum simulation steps and then return the smallest one of them.

3.3 Modelling and Analysis of MA

We model three master algorithms using timed automata in UPPAAL. The following Fig.3 and Fig.4 show the execution of three master algorithms in the form of timed automata respectively. Fixed step algorithm has $Init$, $doStep$ states and synchronize with FMU by channel $continue$. Rollback algorithm has $Init$, $isEqual$, and con states. If all FMUs don't have the same step size, rollback algorithm will communicate with FMUs by $rollback$ signal, otherwise, it will send $continue$ signal and move to con state. Predictable step size algorithm has $Init$, $findCP5$, $getCRh$, $writecp$ states. It obtains the minimal step size ($step2$) of FMUs supporting $GetMaxStepSize$ function and the maximal step size ($step1$) of FMUs supporting rollback function. If $step1$ is greater than $step2$, FMUs receive $rollback$ signal and return to $getCRh$ state. Otherwise, FMUs receive $continue$ signal and do next step. We check the properties of master algorithms including reachability , liveness



(a) Timed automata for fixed step algorithm

(b) Timed automata for rollback algorithm

Fig. 3. Timed automata for fixed step algorithm and rollback algorithm.

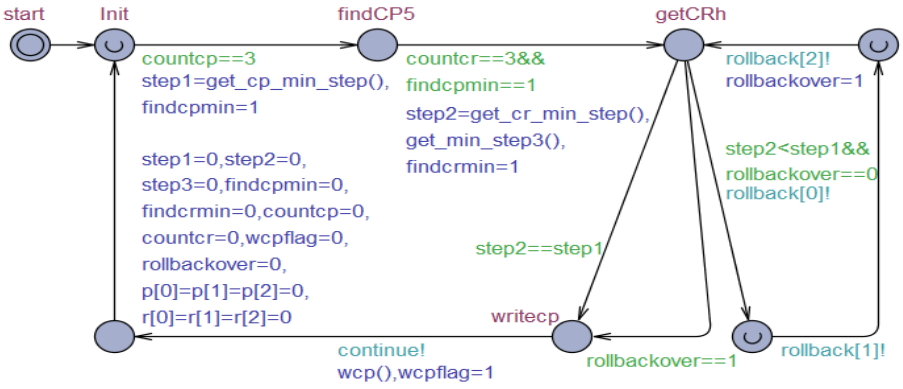


Fig. 4. Timed automata for predictable step sizes algorithm

and deadlock. Experimental results are shown in Table 2.

- $E\langle\rangle$ *master.dostep*, $E\langle\rangle$ *master.con* and $E\langle\rangle$ *master.writecp* are reachability properties checking whether the model can reach these states;
- *master.Init* $\rightarrow$ *master.dostep*, *master.Init* $\rightarrow$ *master.con* and *master.Init* $\rightarrow$ *master.con* are liveness property. If the master algorithm arrives at former state, it eventually reaches the latter state;
- $A[]$ *not deadlock* is safety property checking whether the model will be deadlock.

In Table 2, we can find that the properties such as deadlock, liveness and reachability are satisfied, which proves the correctness of three master algorithms. For an example, $A[]$ *not deadlock* is satisfied, which means that the execution of the master algorithm will not be deadlock. *master.Init* $\rightarrow$ *master.doStep* is satisfied, which means that if the model reach the former state *Init*, it must reach the later state *doStep*. $E\langle\rangle$ *master.doStep* is satisfied, which means that it can reach *doStep* state.

Table 1. Checking Properties of Master Algorithms

Master Algorithm	Property	Verification Result
Fixed Step Algorithm	$A[]$ *not deadlock*	True
	master.Init $\rightarrow$ *master.dostep*	True
	$E\langle\rangle$ *master.dostep*	True
Rollback Algorithm	$A[]$ *not deadlock*	True
	master.Init $\rightarrow$ *master.con*	True
	$E\langle\rangle$ *master.coninue*	True
Predictable Step Size Algorithm	$A[]$ *not deadlock*	True
	master.Init $\rightarrow$ *master.writecp*	True
	$E\langle\rangle$ *master.writecp*	True

4 Architectural Modelling in SysML

To illustrate the approach, we take an example (water tank) inspired by [3]. According to the I/O dependency information between the FMUs, the architectural model for water tank is constructed using SysML.

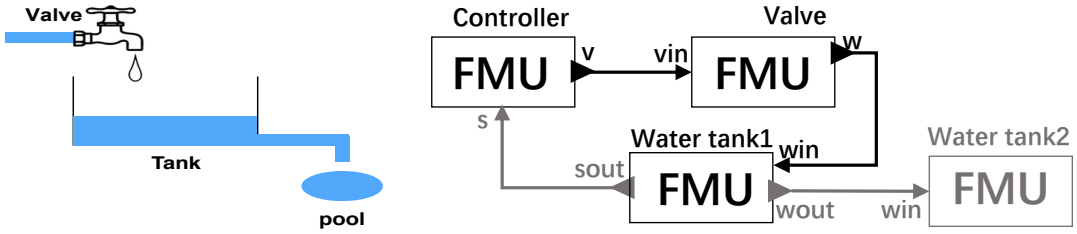
4.1 Case Study: Water Tank

The water tank system, as shown in Fig.5(a), is our running example. A source of water flows into the water tank whose water flows into the drain, and the source is controlled by a valve; when the valve is open the water flows into the water tank. The valve, managed by a software controller, is opened or closed stochastically or depending on the water level. There are two variants of water tank system depending on the various connections between controller, valve and tank.

Figure 5(b) is the architecture connection of water tank. There are three connection cases between the FMUs. The first case contains three FMU components (*Controller*, *Valve* and *Tank1*) and two channels(*v_vin*, *w_win*). The controller and valve are connected with channel *v_vin*. The valve and tank1 are connected with channel *w_win*. For the second case, there could be a channel *sout_s* between tank1 and controller, which means the water level of tank1 affecting the control strategy of the controller. It is denoted with the gray line. Besides, there could be another tank2 (the gray box). The tank1 and tank2 are connected by the channel *w_out*. The orchestration of FMUs for water tank is modeled with SysML block diagram in the next subsection.

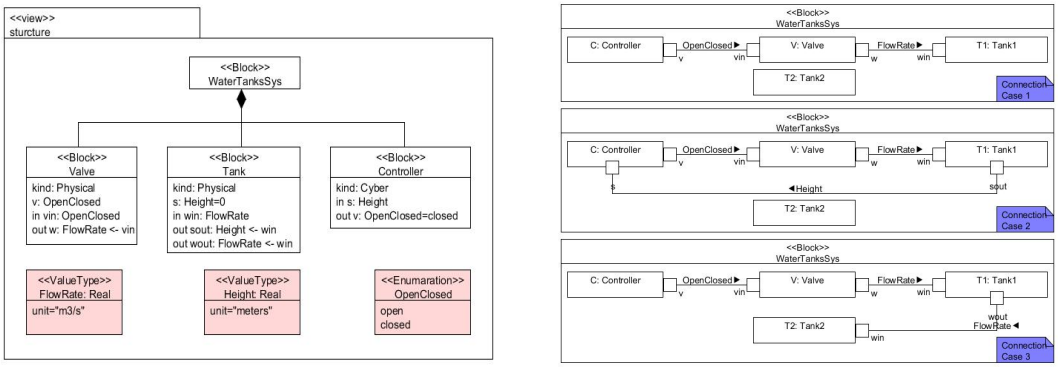
4.2 Architecture Modelling in SysML

SysML is a general purpose domain-specific language (DSL) [23] for model-based systems engineering (MBSE) [15], which is originated as an initiative of the International Council on Systems



(a) Sketch map of water tank system (b) FMUs connection of water tank system.

Fig. 5. Water tank system.



(a) SysML BDD (b) SysML CD

Fig. 6. The SysML block diagrams for water tank system.

Engineering (INCOSE) [21] in January 2001. SysML is implemented as a UML profile. The *Block Definition Diagram* (BDD) describes the system blocks and their features (structural and behavioural). The *Connection Diagram* (CD) describes the internal structure of blocks. The ports of blocks are connected by the connector. The I/O dependence of blocks describes the communication between blocks. SysML block diagrams are usually used to describe the architecture of systems.

Figure 8(a) shows the block definition diagram for the water tank system. The system consists of three blocks, i.e., *Valve*, *Tank* and *Controller*, in which *Valve* and *Tank* are physical components. *Controller* is the cyber component. Each component has its own input and output. For instance, the input interface of *Valve* is named as *vin*, which is used to input the *Open-Closed* signal.

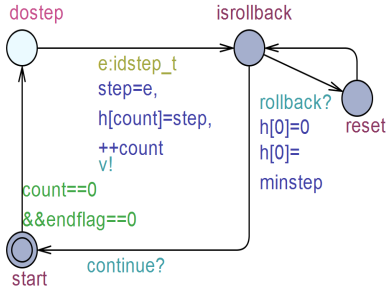
Figure 8(b) shows the connection diagram for the system. There are three cases for connections. The first case is that the system has one valve, one controller and one tank. The controller sends stochastic signals to control the valve on/off leading to various rate of water flow. The second case is that the signal from the controller is affected by the water level of the tank. The last case is on the basis of the first case and adds another tank2 which is affected by the flow rate of the tank1. How can we assure the correctness of the architecture models? We attempt to verify it with model checking based on timed automata. More details on verification process can be found in the next section.

5 Model Checking Co-simulation Using Timed Automata

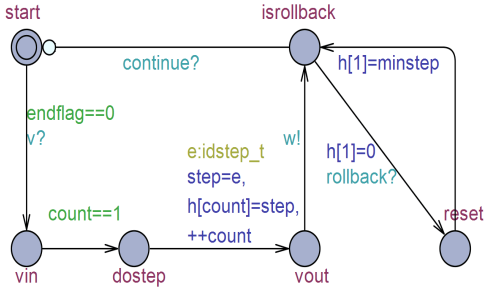
This section performs a formal analysis of the SysML architectures of water tank shown in Fig.8. First off, we model FMUs which are the components of the SysML architectures and the master algorithm which is the director of FMUs using timed automata in UPPAAL. Next, the verification and analysis checks whether the model is accurate and satisfies certain desired properties.

5.1 Modelling FMU component in UPPAAL

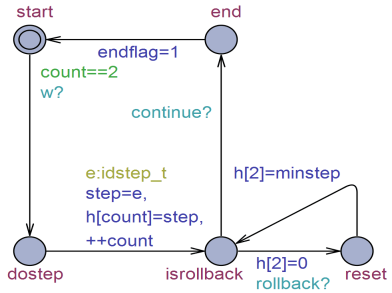
UPPAAL [5] is a toolset for verification of real-time systems represented by (a network of) timed automata which is extended with integer variables, structured data types, and channel synchronization. The execution of FMU component and co-simulation is time-related, therefore it is convenient to model the FMU with timed automata in UPPAAL. In this subsection, we abstract the execution of FMU, and encode it with the states and transitions in timed automata. Besides, we also model the master algorithm as a timed automata to coordinate the execution between several FMU components. The timed automata template for FMU components and master algorithm are shown in Fig.7. Here, we choose rollback algorithm as the master algorithm to coordinate the FMU components. The other two master algorithms can be analyzed with the similar way.



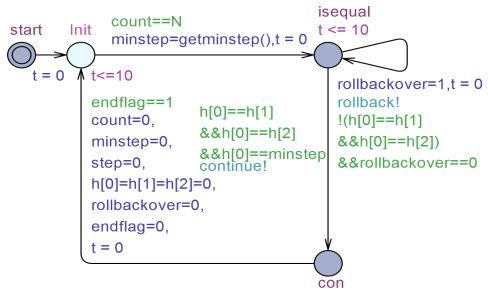
(a) Timed automata template for F-MU controller



(b) Timed automata for FMU valve



(c) Timed automata for FMU tank1

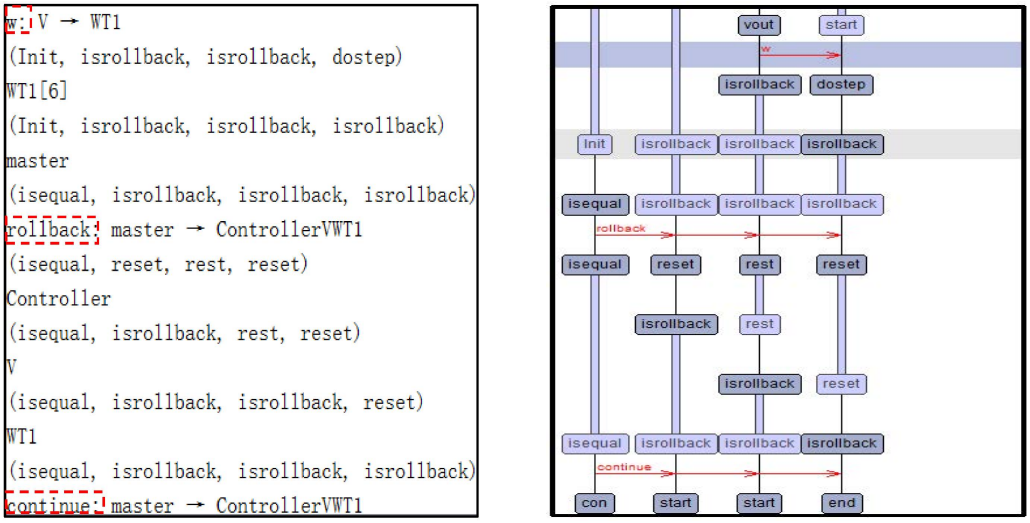


(d) Timed automata for master algorithm

Fig. 7. Timed automata templates for connection case 1.

Figure 7(a), 7(b), 7(c) are the templates for controller, valve and tank1 respectively, they model FMU components which supports rollback function. These FMU components contain four main states, e.g., *start*, *dostep*, *isrollback* and *reset*. Figure 7(a) shows the template for controller which supports a random step size. It synchronizes with valve by signal v and jump to *isrollback* state, and then waits for a signal from the timed automata for the master algorithm. Until the controller receives the *continue* signal, it does data exchange, and return to *start* state. Otherwise it receives *rollback* signal, once it obtains the minimize step size of all FMU components, it travels to *isrollback* state. The states and transitions of valve and tank1 template are similar with the template of controller. Figure 7(d) shows the template for the master algorithm. Firstly, the master algorithm initialize the parameters, and then it get minimize step size of FMU components until all FMU components visit *dostep*. Next, the master algorithm decides which signal should be sent according to the guard. If the step sizes of all FMU components are equal, the master algorithm will send *continue* signal, otherwise, send *rollback* signal.

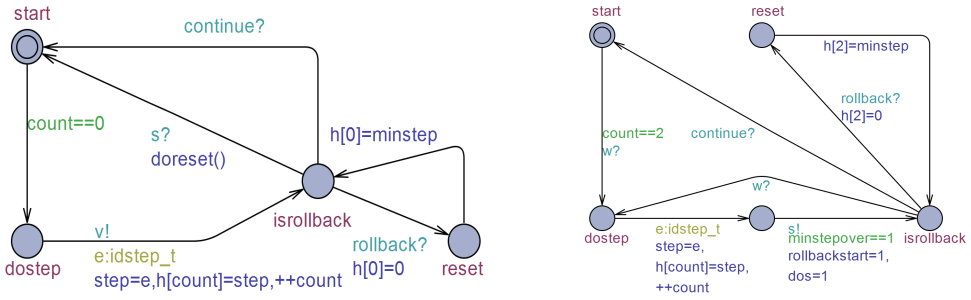
Figure 8 is the execution fragment of the co-simulation, we can find that valve send a w signal to perform data exchange with tank1. After that tank1 move to *dostep* state. The master algorithm send a *rollback* signal to all templates, which leads to all of them arrive at *reset* state. Finally, the master algorithm send a *continue* signal to others. All templates return to *start* state, and then do next step. The execution process shows that our model performs correctly.



(a) Execution trace (b) Execution sequence diagram

Fig. 8. The execution fragment of the co-simulation.

In order to compare the behavior of three connection cases of water tank presented in Section 4.2, we also model the other two connection cases in UPPAAL. We add a channel s to the templates for controller and tank1, which is the model of connection case 2 as shown in Fig.9. We add a template (tank2) and channel $w2$, which is the model of connection case 3 shown in Fig.10. In next subsection, we verify some properties of various connection cases to analyse the correctness of the architecture.



(a) Timed automata for FMU controller (b) Timed automata for FMU tank1

Fig. 9. Timed automata for connection case 2.

5.2 Verification and Analysis

UPPAAL uses a simplified version of TCTL [9] to express the requirement specification. We verify the following TCTL properties of each connection case:

- $E\langle\rangle$ $WT1.isrollback$ and $E\langle\rangle$ $master.con$ are reachability properties checking whether FMU tank1 can reach $isrollback$ state and whether the master algorithm can reach con state respectively.
- $master.start \rightarrow master.con$ are liveness property. If the master algorithm arrive at $start$ state, it eventually reaches con state.
- $A[]$ not $deadlock$ is safety property checking whether the model will be deadlock.

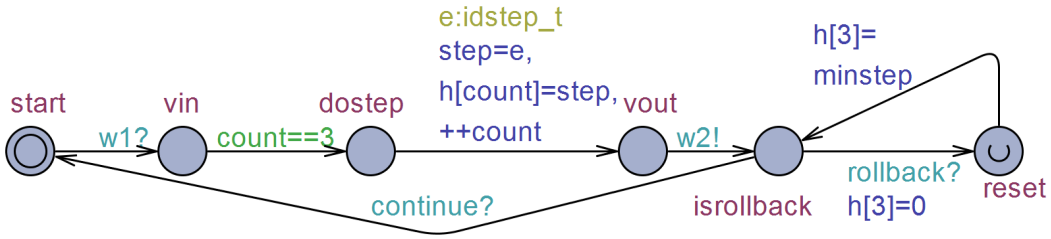


Fig. 10. Timed automata for connection case 3.

The verification results are shown in Table2. We can find that all properties of connection case 1 and 3 are satisfied. It proves that our master algorithm is correct and the composition of FMU components is determinate. However, the liveness and reachability properties of connection case 2 are not satisfied. We find that there is a algebraic loop which may be introduced with the I/O dependency in this connection case. The experiments show that our approach is feasible and useful for model checking the FMI co-simulation. The approach facilitates the verification of the CPSs architecture models.

Table 2. Experimental results

Connection case	Property	Verification Result
Connection case 1	$E\langle\rangle\ WT1.isrollback$	True
	$E\langle\rangle\ master.con$	True
	$master.start \rightarrow master.con$	True
	$A[]\ not\ deadlock$	True
Connection case 2	$E\langle\rangle\ WT1.isrollback$	True
	$E\langle\rangle\ master.con$	False
	$master.start \rightarrow master.con$	False
	$A[]\ not\ deadlock$	True
Connection case 3	$E\langle\rangle\ WT1.isrollback$	True
	$E\langle\rangle\ master.con$	True
	$master.start \rightarrow master.con$	True
	$A[]\ not\ deadlock$	True

6 Related Work

For simulating CPSs [17], distinct simulation domains need to be integrated for a comprehensive analysis of the interdependent subsystems. Co-simulation [8] can maintain all system models within their specialized simulators and synchronizes them in order to coherently integrate the simulation domains. FMI [6][7] is an industry standard which enables co-simulation of complex heterogeneous systems using multiple simulation engines.

Jens Bastian et al. adopts fixed step size master algorithm to simulate heterogeneous systems in [4]. David Broman et al. discussed the determinate composition of FMUs for co-simulation. To do that, they extended the FMI standard to designs FMUs that enables deterministic execution for a broader class of models. Besides, rollback and predictable step size master algorithms are proposed in their work. In [13], Fabio Cremona et al. presents FIDE, an Integrated Development Environment (IDE) for building applications using FMUs. We also implemented the prototype *co-simulator* between continuous-time Markov chains (CTMCs) [14], discrete-time Markov chains (DTMCs) [18] and Modelica in [20]. We also have presented an improved co-simulation framework that focuses on the capture of nearest future event to reduce the number of running steps and the frequency of data exchange between models. In short, the existing work focus on how to achieve deterministic execution of FMUs and improve the efficiency of the master algorithms, however, there is few work to analysis the correctness of master algorithms. In this paper, our work focuses on

the model and analyse I/O dependency information and master algorithms for FMI co-simulation. PG Larsen et al. [19] presented formal semantics of the FMI described in the formal specification language CSP. They formally analyse the CSP model with the FDR3 refinement checker. Nuno Amalio et al. [3] presented an approach to verify both healthiness and well-formedness of an architecture design modeled with SysML. They attempt to check the conformity of component connectors and the absence of algebraic loops to ensure the co-simulation convergence.

In [24], Mladen Skelin et al. reports on the translation of the FSM-SADF formalism to UPPAAL timed automata that enables a more general verification than currently supported by existing tools. Stavros Tripakis [25] discussed the principles for encoding different modeling formalisms, including state machines (both untimed and timed), discrete-event systems, and synchronous dataflow, as FMUs.

Compared with the existing work, the main advantage of our approach proposed in this paper is that it formally model the FMI co-simulation with timed automata. The correctness of the co-simulation between FMUs can be verified with the model checker. Moreover, our approach facilitates the detection the algebraic loop of CPSs architecture. These features make the proposed timed automata-based approach complementary to existing approaches to the formal analysis of the co-simulation.

7 Conclusion and Future Work

This paper has presented our novel approach to check the FMI co-simulation , which facilitates the formal analysis of CPSs. This involves model checking the reachability, livelock and deadlock of three various master algorithms. Besides, the correctness and relevant system properties of the architecture are also analysed. To achieve the goal, we encode the FMU and master algorithms with timed automata. Then the properties of the co-simulation are verified with UPPAAL. We evaluate this approach using the example water tank. The results show that our approach is feasible and useful.

An interesting direction of future work is that we attempt to analyse and compare the performance of various master algorithms in the future. Besides, more complex case studies will be conducted to check the scalability of proposed approach. The tool support for our approach should be improved further.

Acknowledgement

This work was supported by NSFC (Grant No.61472140, 61202104) and NSF of Shanghai (Grant No. 14ZR1412500).

References

1. Acker, B.V., Denil, J., Vangheluwe, H., Meulenaere, P.D.: Generation of an optimised master algorithm for FMI co-simulation. In: Proceedings of the Symposium on Theory of Modeling & Simulation: DEVS Integrative M&S Symposium, part of the 2015 Spring Simulation Multiconference, SpringSim '15, Alexandria, VA, USA, April 12-15, 2015. pp. 205–212 (2015), http://dl.acm.org/citation.cfm?id=2872993
2. Alur, R., Dill, D.L.: A theory of timed automata. Theor. Comput. Sci. 126(2), 183–235 (1994), http://dx.doi.org/10.1016/0304-3975(94)90010-8
3. Amálio, N., Payne, R., Cavalcanti, A., Woodcock, J.: Checking sysml models for co-simulation. In: Formal Methods and Software Engineering - 18th International Conference on Formal Engineering Methods, ICFEM 2016, Tokyo, Japan, November 14-18, 2016, Proceedings. pp. 450–465 (2016), http://dx.doi.org/10.1007/978-3-319-47846-3_28
4. Bastian, J., Clau, C., Wolf, S., Schneider, P.: Master for co-simulation using fmi (2011)
5. Behrmann, G., David, A., Larsen, K.G., Håkansson, J., Pettersson, P., Yi, W., Hendriks, M.: UPPAAL 4.0. In: Third International Conference on the Quantitative Evaluation of Systems (QEST 2006), 11-14 September 2006, Riverside, California, USA. pp. 125–126 (2006), http://dx.doi.org/10.1109/QEST.2006.59

6. Blochwitz, T.: The functional mockup interface for tool independent exchange of simulation models (2011-03-22), 105–114 (2011)
7. Blochwitz, T., Otter, M., kesson, J., Arnold, M., Clauss, C., Elmqvist, H., Friedrich, M., Junghanns, A., Mauss, J., Neumerkel, D., Olsson, H., Viel, A.: Functional mockup interface 2.0: The standard for tool independent exchange of simulation models. In: Proceedings of the 9th International Modelica Conference. pp. 173–184. The Modelica Association (2012), http://dx.doi.org/10.3384/ecp12076173
8. Bogomolov, S., Greitschus, M., Jensen, P.G., Larsen, K.G., Mikucionis, M., Podelski, A., Strump, T., Tripakis, S.: Co-simulation of hybrid systems with spaceex and uppaal (2015)
9. Boucheneb, H., Gardey, G., Roux, O.H.: TCTL model checking of time petri nets. J. Log. Comput. 19(6), 1509–1540 (2009), http://dx.doi.org/10.1093/logcom/exp036
10. Broman, D., Brooks, C.X., Greenberg, L., Lee, E.A., Masin, M., Tripakis, S., Wetter, M.: Determinate composition of fmus for co-simulation. In: Proceedings of the International Conference on Embedded Software, EMSOFT 2013, Montreal, QC, Canada, September 29 - Oct. 4, 2013. pp. 2:1–2:12 (2013), http://dx.doi.org/10.1109/EMSOFT.2013.6658580
11. Cheng, B., Wang, X., Liu, J., Du, D.: Modana: An integrated framework for modeling and analysis of energy-aware cpss. In: IEEE Computer Software and Applications Conference. pp. 127–136 (2015)
12. Cremona, F., Lohstroh, M., Broman, D., Natale, M.D., Lee, E.A., Tripakis, S.: 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. pp. 173–183 (2016), http://dx.doi.org/10.1109/MEMCOD.2016.7797762
13. Cremona, F., Lohstroh, M., Tripakis, S., Brooks, C.X., Lee, E.A.: FIDE: an FMI integrated development environment. In: Proceedings of the 31st Annual ACM Symposium on Applied Computing, Pisa, Italy, April 4-8, 2016. pp. 1759–1766 (2016), http://doi.acm.org/10.1145/2851613.2851677
14. Danos, V., Heindel, T., Garnier, I., Simonsen, J.G.: Computing continuous-time markov chains as transformers of unbounded observables. In: Foundations of Software Science and Computation Structures - 20th International Conference, FOSSACS 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22-29, 2017, Proceedings. pp. 338–354 (2017), http://dx.doi.org/10.1007/978-3-662-54458-7_20
15. Dori, D.: Model-Based Systems Engineering with OPM and SysML. Springer (2016), http://dx.doi.org/10.1007/978-1-4939-3295-5
16. Fritzson, P., Engelson, V.: Modelica - a unified object-oriented language for system modelling and simulation. Lecture Notes in Computer Science 1445(1445), 67–90 (1998)
17. Georg, H., Müller, S.C., Rehtanz, C., Wietfeld, C.: Analyzing cyber-physical energy systems: The INSPIRE cosimulation of power and ICT systems using HLA. IEEE Trans. Industrial Informatics 10(4), 2364–2373 (2014), http://dx.doi.org/10.1109/TII.2014.2332097
18. Guerry, M.: On the embedding problem for discrete-time markov chains. J. Applied Probability 50(4), 918–930 (2013), http://dx.doi.org/10.1017/S002190020001370X
19. Larsen, P.G., Fitzgerald, J., Woodcock, J., Fritzson, P.: Integrated tool chain for model-based design of cyber-physical systems: The into-cps project. In: International Workshop on Modelling, Analysis, and Control of Complex Cps. pp. 1–6 (2016)
20. Liu, J., Jiang, K., Wang, X., Cheng, B., Du, D.: Improved co-simulation with event detection for stochastic behaviors of cpss. In: 40th IEEE Annual Computer Software and Applications Conference, COMPSAC 2016, Atlanta, GA, USA, June 10-14, 2016. pp. 209–214 (2016), http://dx.doi.org/10.1109/COMPSAC.2016.133
21. Pepper, I.K., Wolf, R.: International council on systems engineering. Police Journal 88(3), 7–7 (2015)
22. Rahim, M., Hammad, A., Ioualalen, M.: A methodology for verifying sysml requirements using activity diagrams. ISSE 13(1), 19–33 (2017), http://dx.doi.org/10.1007/s11334-016-0281-y
23. Semeráth, O., Barta, Á., Horváth, Á., Szatmári, Z., Varró, D.: Formal validation of domain-specific languages with derived features and well-formedness constraints. Software and System Modeling 16(2), 357–392 (2017), http://dx.doi.org/10.1007/s10270-015-0485-x
24. Skelin, M., Wognsen, E.R., Olesen, M.C., Hansen, R.R., Larsen, K.G.: Model checking of finite-state machine-based scenario-aware dataflow using timed automata. In: 10th IEEE International Symposium on Industrial Embedded Systems, SIES 2015, Siegen, Germany, June 8-10, 2015. pp. 235–244 (2015), http://dx.doi.org/10.1109/SIES.2015.7185065
25. Tripakis, S.: Bridging the semantic gap between heterogeneous modeling formalisms and FMI. In: 2015 International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation, SAMOS 2015, Samos, Greece, July 19-23, 2015. pp. 60–69 (2015), http://dx.doi.org/10.1109/SAMOS.2015.7363660
26. Zanero, S.: Cyber-physical systems. IEEE Computer 50(4), 14–16 (2017), https://doi.org/10.1109/MC.2017.105