Turkish Journal of Electrical Engineering & Computer Sciences

http://journals.tubitak.gov.tr/elektrik/

Research Article

Turk J Elec Eng & Comp Sci
(2019) 27: 3599 – 3614

doi:10.3906/elk-1807-334

Schedulability analysis of real-time multiframe cosimulations on multicore platforms

Muhammad Uzair AHSAN*, Halit OĞUZTÜZÜN
Department of Computer Engineering, Faculty of Engineering, Middle East Technical University, Ankara, Turkey

Received: 31.07.2018 • **Accepted/Published Online:** 17.12.2018 • **Final Version:** 18.09.2019

Abstract: For real-time simulations, the fidelity of simulation depends not only on the functional accuracy of simulation but also on its timeliness. It is helpful for simulation designers if they can analyze and verify that a simulation will always meet its timing requirements without unnecessarily sacrificing functional accuracy. Abstracting the simulated processes simply as software tasks allows us to transform the problem of verifying timeliness into a schedulability analysis problem where tasks are checked as to whether they are schedulable under the timing constraints or not. In this paper we extend a timed automaton-based framework due to Fersman and Yi for schedulability analysis of real-time systems, for the special case of real-time multiframe cosimulations. To the best of our knowledge, this work is the first to analyze the schedulability of single- or multiframe real-time simulations. We found that there are some special requirements posed by multiframe simulations, which necessitate changes and improvements in the existing framework designed for actual real-time systems. We made the required theoretical extensions to the framework and implemented our extended framework in UPPAAL, a tool for modeling, simulation, and verification of real-time systems modeled as timed automata. The functional correctness and resource requirements of the implemented framework are then demonstrated using simple examples.

Key words: Schedulability analysis, real-time simulations, cosimulations, task automaton, timed automaton

1. Introduction

The feature that distinguishes real-time simulation from others is that the simulation fidelity does not only depend on the functional accuracy of the simulation model but also on the time correctness of the simulation execution. In other words, the wall-clock time during a real-time simulation run must remain less than or equal to the simulation time of the system models of the simulation. Let us call this constraint the real-time constraint. The simulated models in a simulation are run by a simulation scheduler. If it can somehow be proven that a particular scheduling technique used in a real-time simulation always results in an execution of simulation models such that it never violates its real-time constraint then such a proof enhances our confidence in the simulation results. The process of obtaining such a proof is referred to as schedulability analysis.

In this work, we develop a schedulability analysis framework for the case of real-time multiframe cosimulation. This work is the first attempt, to the best of our knowledge, that deals with the schedulability analysis of single- or multiframe real-time simulation. All the existing works that we are aware of on schedulability analysis are meant for operational real-time systems only. The major contributions of this paper include presentation of a model for real-time multiframe cosimulations, development of a schedulability analysis framework for the

*Correspondence: e1952092@ceng.metu.edu.tr

presented model based on an existing timed automata-based framework for schedulability analysis of real-time systems, and finally implementation and demonstration of the new framework.

1.1. Real-time multiframe cosimulation

The kind of simulation architecture where each model is solved independently in parallel and where one simulation tool acts as the master or the orchestrator[1], executing the simulation tasks and performing coordination among them, is known as cosimulation architecture.

The frame time of a simulation model is defined as the time interval between data transfers to or from other simulation entities that are external to that model. These entities can either be hardware components or other simulation models. The simulation master asks each model to proceed one step at a time. This step is called a simulation step, and the step size is the amount of time the master wants the model to progress. A frame of a simulation model can also contain multiple simulation steps [2] of that model. The simulations that use different step sizes for different models are called multirate simulations [3] or simulations with multiframing [4]. We can now redefine the real-time constraint introduced in Section 1 in terms of the frames as the constraint that the wall-clock time must be less than or equal to the simulation time of a simulation model at the end of each of its frames. This requirement on the simulation model is also known as "meeting its deadline".

1.2. Schedulability analysis problem and the proposed approach for a solution

In a real-time system, a set of software tasks is said to be schedulable if it is possible for every task in the set to meet its deadline. Schedulability analysis can then be defined as a process that determines if a task set is schedulable given the properties (execution times, arrival times, etc.) of the tasks and the imposed constraints (task deadlines, resources requirements, etc.). Similarly, in a real-time cosimulation, the simulation steps of simulated models can be simply considered as software tasks. The problem of ensuring the timing correctness of real-time cosimulation can then be regarded as one of real-time schedulability analysis of the given task-set.

To verify that a particular scheduling algorithm will never cause any task to miss its deadline, a proof using some formal method is required. There are a number of schedulability analysis methods available in the literature and among them is a class of analysis methods that are based on the theory of timed automata [5–7] or an extension of them called task automata [8–10]. Timed automata are finite automata equipped with clocks, whereas task automata further extend the definition of timed automata to include annotations for states (locations) or edges depicting release of a task. These works use task automata to define a system's task model, a fundamental notion in the theory of real-time schedulability analysis. A task model basically represents an abstraction of different tasks' behaviors that are considered important for the problem at hand. This task automaton-based task model is considered as the most general and most expressive form, which encompasses almost all other task models in the literature [11].

Our goal in this work is to perform schedulability analysis for real-time multirate cosimulations to prove the timing correctness of these simulations. We selected the timed automata-based schedulability analysis framework proposed by Fersman and Yi [8] (FY framework) as the basis of our analysis technique. Although timed automata-based techniques may suffer from high complexity (because of state space explosion) and even undecidability [10], we mitigate these concerns by assuming that each task has a dedicated processor core. The FY framework transformed the schedulability analysis problem into a state reachability problem of a timed automaton. We worked on the same principle and transformed our problem as the state reachability problem.

After the necessary theoretical extension of the FY framework, we implemented the schedulability

analyzing timed automaton in UPPAAL [12, 13], which is a popular timed automata-based tool for model-checking, and then we tested the results and performance of our framework for a simple example.

1.3. Related work

After the introduction of timed automata by Alur and Dill in 1994 [5], researchers were quick to realize that timed automaton-based formalism could be extended to model task arrival patterns. The works of Norstrom [14] and Fersman [15] were the first to propose the use of an extension of timed automata as the task arrival pattern. This extended timed automaton was later considered for real-time tasks and their scheduling analysis and was also renamed as task automaton [10]. The task automaton turned out to be the most expressive task model in the literature [11]. Besides task automata-based research, which we will describe in more detail in one of the following paragraphs, other prominent works on scheduling analysis that used timed automata include [16–18].

There has been a considerable body of cosimulation-related literature produced in the past few years [19–21]. For an extensive survey of this literature, see [1]. However, to the best of our knowledge, none of the cosimulation work deals with the problem of schedulability. Formal verification techniques are applied to cosimulations [1], but only to verify the functional correctness of the simulation and not to check time correctness.

Our work is based on task automata, which are used in a number of scheduling analysis works. The authors of [10] analyzed the real-time schedulability of tasks on a single processor. They also determined three conditions that, if simultaneously hold true, will cause the solution to scheduling analysis to be undecidable. In [9, 22], the authors examined the schedulability of fixed-priority systems using task automata. They also studied the case of data-dependent control, where the release time of a task may depend on a specific value of some shared variable and hence on the time-point when some previous task finishes its execution. In a further improvement of [22], the same authors introduced a more generic framework for real-time schedulability analysis using task automata [8]. The new model was able to handle more general precedence relations and resource constraints and they showed that the schedulability analysis problem for this more general case can be solved using the same technique that was introduced in [22]. In another work [23], scheduling analysis using task automata was extended to a multiprocessor setting and the authors found one more negative result as compared to the single processor case in [10]. More precisely, they showed that the truth of only two conditions is sufficient for the solution of a scheduling problem to fall into the undecidable category.

2. Multiframe cosimulation model

In this section we shall present a system model that we believe accurately describes a multiframe cosimulation. In an effort to ensure that our proposed model for multiframe cosimulation reflects the standard industry practices, we based our model on the cosimulation model presented in FMI standard 2.0 [24]. Before describing the model of a multiframe cosimulation system, let us first list the assumptions that we make for our model:

1. There are as many processor cores available as there are task types.
2. Each processor core is assigned to a task type once for a simulation run. This assignment may be made at design time or at runtime, but a processor core, once assigned, remains associated with the assigned task type for the entire period of simulation.
3. Task instances can be aborted at any time during their execution.
4. An aborted task instance is responsible for leaving its acquired resources in a clean state.

The purpose of the first assumption is to avoid cyclic dependencies among different task types because of core sharing. Similarly, the precedence constraints or dependencies among tasks assigned to different cores are also required to be acyclic (see Section 3.1). Cyclic dependencies among tasks make our analysis approach, as discussed in Section 4, unusable. However, it is important to note that our model allows multiple instances of the same task type to be active at the same time and therefore a first-in first-out (FIFO) queue needs to be maintained for each core to keep the waiting instances of the assigned task type.

Assumption no. 2 forbids task migration between processor cores at runtime and relieves our framework from handling delays arising due to these task migrations. However, it also limits the framework's applicability by excluding a class of schedulers that optimize system performance by dynamically shifting tasks between cores during runtime.

We assume that tasks can be aborted or canceled at any time after their release. However, to ensure a smooth simulation run in the event of task cancellation, assumption no. 4 makes the canceled task responsible for leaving the system resources that it has been using in a clean state.

As described in the FMI standard [24], our system will be a cosimulation with a master that orchestrates and synchronizes the running of the simulation. Tasks are allowed to communicate with the master only and not directly with each other, and the master is then responsible for forwarding the communicated data to the destination task, if any. This communication between the tasks and master does not necessarily have to occur at the end of every simulation step; however, every communication time-point must coincide with the end time of a simulation step. In other words, the simulation model frames are allowed to contain multiple simulation steps. Figure 1 shows this communication architecture. For the sake of simplicity, we assume that the communication between the master and a slave task has a constant delay and that this delay is included in the worst-case execution time of each task.

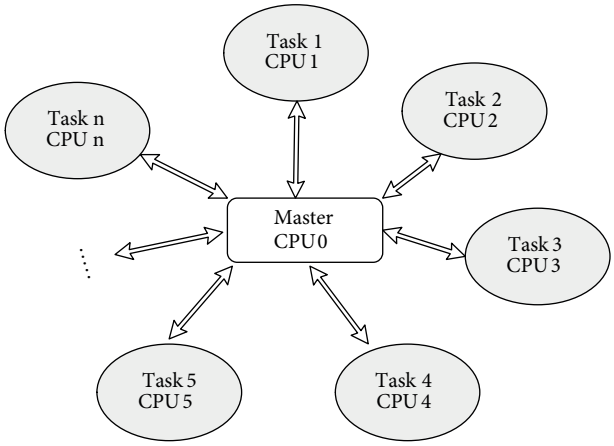

Figure 1. Cosimulation system model.

Besides communication between the tasks, the master is also responsible for scheduling or invoking each task for execution. A scheduler is therefore a part of the master and is responsible for executing tasks in such a way that no task misses its deadline while at the same time ensuring that the precedence constraints between them are not violated. However, a scheduler can only schedule a task after it is released for execution or in other words becomes active. In our system model, we use task automata to define the task release or arrival patterns.

3. The proposed framework

Real-time schedulability analysis in the case of multiframe real-time cosimulation, possibly FMI-based, actually amounts to analyzing the scheduler component of the cosimulation master. The analysis process checks if it is possible for the scheduler to schedule tasks in real time for a given task arrival pattern and the constraints defined between task types. The scheduler component makes scheduling decisions based on inputs provided to it by the simulation designer using a task model. A task model includes the definition of the tasks' arrival patterns, worst-case execution times, deadlines, and precedence relations between them.

Since we intend to extend the timed automata-based FY framework for real-time schedulability analysis, timed automata will naturally play a major role in our proposed framework as well. More specifically, a timed automaton will be defined that will analyze schedulability for the cosimulation master while a slight variation of timed automata, i.e. task automata [8], will be a major part of our task model description.

Next is the description of how the task model component of the master is represented in our proposed framework, followed by the description and discussion of the timed automaton that can analyze the real-time schedulability of any task in a given task model.

3.1. The task model

The task model component of the master is defined as a 3-tuple $\langle S, A, G \rangle$, where:

- S is a set of task types. Each member of the set S defines a type of task.
- A is a set of task automata that defines the task arrival patterns.
- G is a directed acyclic graph (DAG) that defines precedence constraints between tasks.

Following is the detailed description of each component of the task model tuple.

3.1.1. Task type

We represent a task as an abstraction called a task type that contains only those task attributes that are of interest. Multiple instances of a task type can exist in a system at the same time. We shall use the term "task" to refer to both task type and its instance wherever the actual meaning is clear from the context.

Before proceeding further, let us define the three possible states of a task in our framework:

- *Idle*: When a task is waiting to be released by the cosimulation master.
- *Waiting*: When a task is released but is waiting either for some precedent task to finish execution or for processor time.
- *Executing*: When the task is actually executing on a processor.

The three states and the transitions possible between them are shown in Figure 2. A task is termed active during waiting or executing states. The response time (*ResponseTime*) of a task is defined as the time elapsed between the instant when it was released and the instant when it finishes execution.

In most of the related works, a task type is usually defined by two static attributes: the task's worst-case execution time and its relative deadline. However, in our system model, where task periodicities are allowed to be dynamic, the deadline attribute cannot be fixed at the time of task type definition. We therefore define the

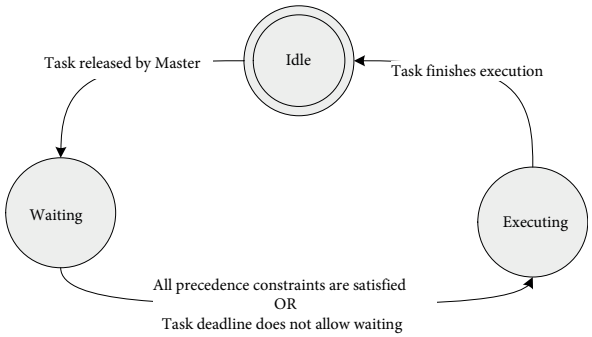

Figure 2. Task states.

Deadline attribute as a dynamic property for our task types, which will be assigned a value by the cosimulation master whenever a task instance is released for execution.

Recall that in Section 2 we assumed in our real-time cosimulation system model that a task can be canceled at any time and will leave the system in a clean and well-defined state. To reach this clean state, a canceled task typically needs to perform some additional housekeeping tasks that require processor time. We introduce a new attribute in task type definition called worst-case cancellation time ($WorstCaseCancelTime$). It is formally defined as the maximum time that can elapse between a cancellation request from the master and completion of that cancellation by the task. It is defined as a static task attribute and is assumed to be determined at design time.

Two more attributes are added in our task type definition that are required for real-time schedulability analysis. These are the task's current simulation time ($SimulationTime$) and its current integration interval ($CurrentIntegrationInterval$). $CurrentIntegrationInterval$ is required to maintain $SimulationTime$, by continuously accumulating it, and also to calculate the next arrival time of a task. $SimulationTime$ is needed in handling precedence constraints among tasks, as explained in Section 3.2.3.

The formal definition of task type is therefore a 5-tuple $\langle WCET, WCCT, DL, CII, ST \rangle$, where:

- $WCET$ and $WCCT$ are static attributes representing the task's $WorstCaseExecTime$ and $WorstCaseCancelTime$, respectively.
- DL,CII represent the task's $Deadline$ and $CurrentIntegrationInterval$ and are assigned values at runtime by the master.
- ST is the task's current $SimulationTime$ being regularly updated using the $CurrentIntegrationInterval$.

3.1.2. Task automata

To define the arrival pattern of tasks, a task automaton will be used, which is actually a timed automaton with some automaton states (termed as locations) annotated with tasks [8]. A transition to a location annotated with a task means that the task mentioned in the annotation now leaves the idle state and moves to waiting state. One minor modification in our case is that before each task gets ready for execution, the master will set the $CurrentIntegrationInterval$ and $Deadline$ parameters of that task.

3.1.3. Precedence DAG

The precedence constraints can be naturally represented as a DAG where the nodes represent the tasks and the edges between them denote the presence of a precedence constraint. Use of a DAG means that cyclic constraints are not allowed. As discussed in Section 3.2.2 below, we will have two kinds of precedence constraints in our system model so the DAG in our case needs to have two types of edges to represent the two kinds of constraints.

3.2. Precedence handling in the proposed framework

Precedence handling is the basic difference between the FY framework and our proposed framework. Therefore, we shall build upon this difference and introduce the related concepts and terms as we go forward.

3.2.1. Precedence handling in FY framework

The FY framework used boolean variables to indicate precedence between two tasks and had just one kind of precedence constraint that is mandatory for the dependent task to satisfy. In the framework, a Boolean variable $g_{i,j}$, initialized as $false$, is introduced for every pair P_i, P_j if P_j is dependent on P_i. A $true$ value of variable $g_{i,j}$ means that P_i has completed and so P_j is allowed to run. Every time a task P_i finishes execution, all $g_{i,j}$ are set to $true$ and subsequently switched back to $false$ when the dependent task P_j is completed. A task P_j is ready to run only when $g_{k,j} = false$ for no P_k; hence, a simple conjunction of all $g_{k,j}$ variables is enough to determine if every precedence constraint of P_j is satisfied or not.

3.2.2. Shortcomings of FY framework

The problem with this relatively simple precedence handling is that it forces each precedence constraint to be satisfied every time and the real-time schedulability under these mandatory precedence constraints does not depend only on the $WorstCaseExecTime$ of tasks but also on the fact that all related tasks either have the same periodicity, i.e. they all become ready for execution at the same time, or the precedent tasks have periodicity shorter than that of the dependent task. If a dependent task P has a shorter periodicity and hence shorter integration interval and deadline than its preceding task Q, then no matter how short the $WorstCaseExecTime$ of P is, a mandatory precedence constraint between P and Q will cause each successive P instance to finish closer and closer to its deadline, eventually causing a P instance to miss its deadline. However, our focus is on real-time cosimulations with tasks having different frame sizes. In such multiframe real-time cosimulations, contrary to the assumption in the FY framework, the tasks can have different periodicities. Another point of difference is that the normal data dependency kind of precedence constraints among periodic simulation tasks need not be a binding precedence constraint since it is almost always possible to advance the simulation with extrapolated old data to ensure that no task misses its deadline. This should not mean that the cosimulation master will always ignore the data dependency constraints among periodic simulation tasks because it is always desirable, for the sake of the accuracy of the simulation, to wait for fresh outputs from the preceding tasks if the dependent task's deadline permits. In other words, if there is a possibility of improving the simulation accuracy by delaying a task's execution without violating its deadline, then the master should go for the delayed execution.

Besides the nonbinding precedence constraints, there will be some precedence constraints that are binding and the master must wait for these precedent tasks to finish before proceeding ahead. We thus have two kinds of precedence constraints in our system, one that is nonbinding and another that is binding.

3.2.3. Solution presented for the proposed framework

In our framework, an $n \times n$ matrix is used to represent the two kinds of precedence constraints where n is the number of tasks in the system. Each entry $E_{i,j}$ in the matrix denotes whether or not there is a precedence constraint between tasks P_i and P_j and, if so, the type of that constraint. Therefore, $E_{i,j} \in \{B, N, \varnothing\}$, where $\varnothing$ denotes *No constraint* while symbols B and N denote the existence of a binding or nonbinding constraint, respectively.

However, the information represented in the above matrix is not enough for our purposes. There must be a way for the cosimulation master to determine how long the execution of the task under analysis has to be delayed because of some binding constraint, or for the case of nonbinding constraints whether or not it is suitable to wait at all for a nonbinding precedent task to finish, and if yes, how long. The FY framework does not provide any information to make such a decision. Our framework improves upon this and helps in deciding between accuracy and real-time requirements by providing the following two necessary pieces of information,

1- Remaining response time: The master can use the remaining response time ($RemResponseTime$) of each precedent task to determine if the execution of P can be delayed to get fresh outputs from the precedent tasks without violating P's deadline by checking the following condition:

$$Deadline(P) \geq RemResponseTime(Q) + WorstCaseExecTime(P) \tag{1}$$

The amount of time that a task P can wait is calculated by considering the largest $RemResponseTime$ of a precedent task Q for which the condition given in Eq. 1 is true.

Our framework maintains a separate clock variable for each precedent task instance that keeps track of the time since that instance is active. This clock in addition to the total response time of a task is used to keep a running estimate of $RemResponseTime$ of that particular task.

2- Simulation time: The current $SimulationTime$ of tasks also needs to be taken into consideration by the cosimulation master. In a multiframe cosimulation, tasks have different periodicities; therefore, their $SimulationTime$ values are not in synchrony with each other. In such a scenario, the master is responsible for handling the precedence relations in a way that can counter this lack of synchronization.

4. Schedulability analyzing by timed automaton

4.1. Checking automaton

The timed automaton that we define for schedulability analysis is termed as checking automaton (CA) as it basically checks if it is possible for instances of a particular task type to have a response time greater than its assigned deadline, and therefore is unschedulable. The defined CA is actually a template automaton that needs to be instantiated for schedulability analysis of each task type separately.

Like in the FY framework, the CA encodes the problem of unschedulability as a reachability problem. However, the CA checks all the task instances released during a simulation scenario of the task type it is instantiated for, and an *ERROR* state is reached whenever any one of the task instances is determined to be violating its deadline, whereas the designers of the FY framework defined an automaton that checks a single instance of a task type for unschedulability. They selected a particular task instance nondeterministically for the analysis.

We shall refer to the task type for which the CA is instantiated as P while the precedent tasks that will feature in our discussion shall be referred to as Q. Similarly, the automaton location names in the CA that are subscripted by p refer to the task under analysis whereas other subscripts are used for locations dealing with precedent tasks.

Before proceeding further, let us describe a few data structures that are used in the definition of CA:

- *queue*, a FIFO queue to hold the instances of P while they are in waiting state. Maximum size of the queue is defined by CA parameter $maxQsize$.
- *ANBTasks*, an array that holds all the active nonbinding precedent tasks.
- *ABTasks*, an array that holds active binding tasks with largest $RemResponseTime$.

A couple of constants defined in each instantiation of the CA determine the maximum number of clocks defined in that instance of the CA. These are:

1. **maxQsize:** Maximum number of instances of the task under analysis, P, that can be in the queue at one time waiting for execution. A clock is required for each waiting task.
2. **maxActiveNonBindingTasks:** Maximum number of nonbinding precedent tasks that can be active at any given moment. A clock is required for each active nonbinding task.

A clock is required for each binding precedent task as well, but since $RemResponseTime$ of only one binding precedent task is required for one instance of P, the maximum number of active binding precedent tasks that we may need to maintain is equal to $maxQsize$. Therefore, the total number of clocks defined in a CA is given by the following equation:

$$nClocks = 2 \times maxQsize + maxActiveNonBindingTasks + 2 \tag{2}$$

4.2. The checking automaton construction

A timed automaton consists of nodes called locations and edges denoting transition between locations. Table 1 provides the description of all the locations and the edges between them as defined in the CA shown in Figure 3.

5. Experiments

The developed framework was implemented in UPPAAL version 4.1 and was tested on Middle East Technical University's Computer Engineering Department's *inek* machine with 64-bit 3.10 GHz Intel Core i7-4770S processor and 1600 MHz, 2x4 GB DDR3 RAM. A simple example was developed to determine the accuracy of our framework implementation in analyzing the schedulability of a simulation scenario. Having proved the functionality of the framework, another set of tests was performed to empirically assess two performance parameters: time taken and total memory used during schedulability analysis. Following is a description of these two kinds of experiments.

5.1. Functionality verification experiment

A simple case of a car's power window simulation was selected for this experiment [25]. The power window system works by reacting to the user pressing the window's up or down button. A microcontroller reads the user input and outputs a command signal to the DC motor, which runs to move a scissor mechanical assembly

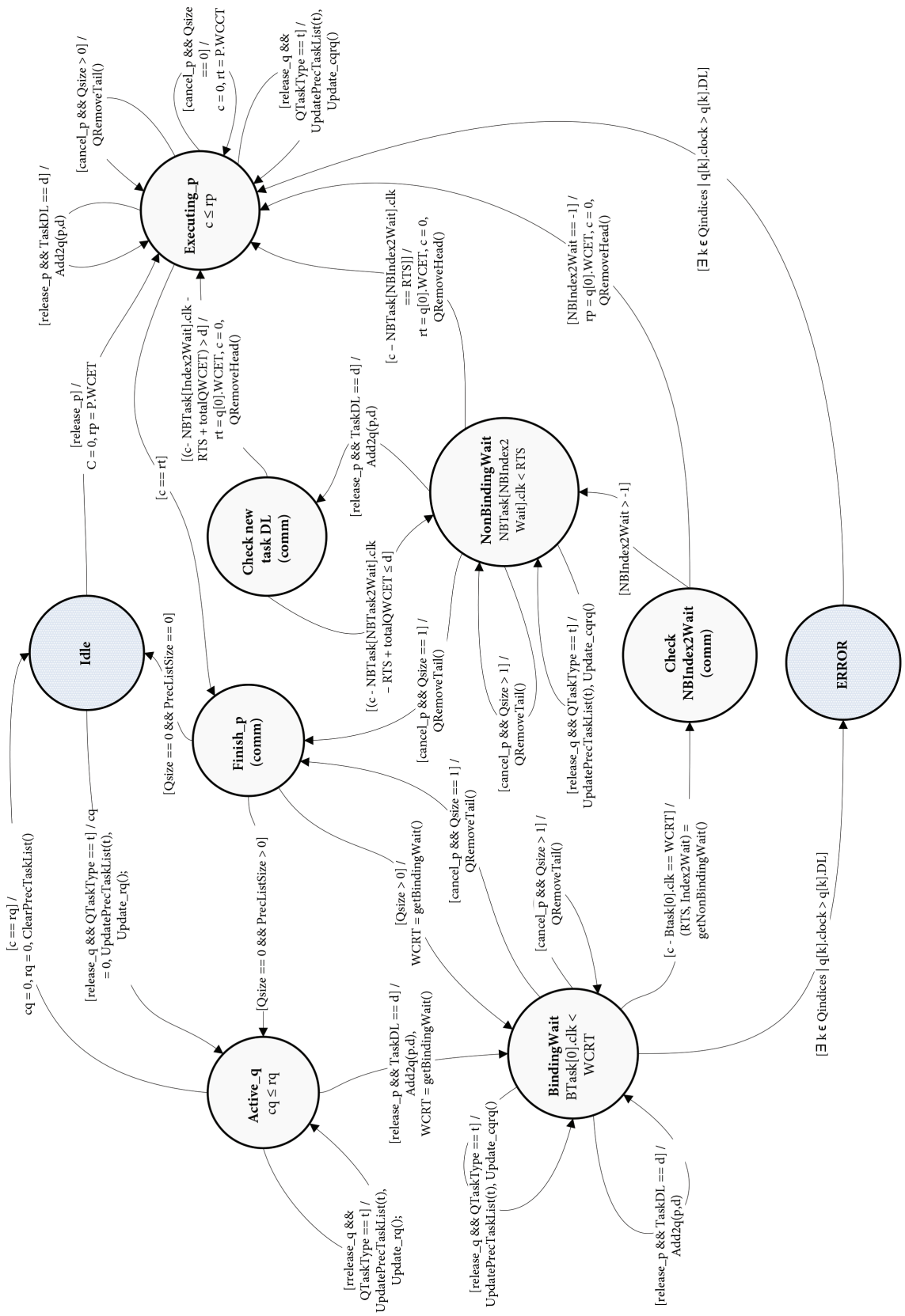

Figure 3. The checking automaton (CA).

that in turn moves the car window. However, the user input is overridden if an obstacle is detected in the path of an upward moving window. The obstacle event causes the controller to cancel its current command to move the window up and instead issue a new command to move the window down a few centimeters. To realize the defined functionality, the simulation was assumed to be composed of six simulation tasks, which are given in Table 2.

The DAG that describes the precedent constraints or dependencies among these tasks is given in Figure 4. All the dependency relations in the graph are nonbinding precedent constraints with the only exception of the relation between *Controller* and *ObstacleEventHandler* tasks, which is defined as a binding constraint and shown by a solid line in the figure. In our experiment, we analyzed the schedulability of the *Controller* task.

Table 1. Description of locations for checking automaton.

Location Name	Description
Idle	To start with, the CA is in *Idle* location. The automaton leaves this location when either an instance of task under analysis, i.e. P, or an instance of one of the precedent tasks is released for execution.
Active_q	In *Active_q* location, the CA keeps track of all the active nonbinding precedent tasks and an active binding task, one with the largest *RemResponseTime*. A transition back to *Idle* location is made when all the active precedent tasks finish execution or, alternatively, the CA moves to *BindingWait* location when an instance of P is released.
BindingWait	At this location, the CA imitates delaying execution of task P instance due to an active binding precedence task. The duration of this mandatory delay equals the largest *RemResponseTime* value among active binding precedent tasks. At the end of binding wait time, the CA moves to *CheckNBIndexToWait* location to check if it can wait for any active nonbinding precedent task to finish execution. Other transitions that are possible include moving to *Finish_p* state if simulation master cancels all waiting P instances or to *ERROR* location if clock of a waiting P instance in the queue exceeds its deadline.
CheckNBIndexToWait	*CheckNBIndexToWait* location is defined as a committed location for the CA, which means that no time can pass while the automaton is in this location. Therefore, the location is immediately exited by simply checking if there is a nonbinding precedent task instance that can be waited upon safely. If yes, then the CA moves to *NonBindingWait* location; otherwise, a move to *Executing_p* location is made.
NonBindingWait	This location imitates the delay due to waiting for a nonbinding precedent task. From here, the CA can move to *Executing_p* location when the waiting time expires, to *CheckNewTaskDL* location if a new instance of P is released, or to *Finish_p* location in the event that the simulation master cancels all waiting P instances.
Executing_p	This location emulates the time spent while executing an instance of task P. It can be entered either from *Idle* location when an instance of task P gets released with no active precedent tasks or after exhausting all the waiting periods, binding and/or nonbinding. A normal exit from this location will cause the CA to move to *Finish_p* location. However, in the case of a P instance's clock surpassing its associated deadline, the CA moves to the *ERROR* location.
Finish_p	*Finish_p* location is entered either when an instance of task P, which is either at the head of the queue or is executing, gets canceled or when a P instance finishes execution. It is again a committed location and so the CA exits this location immediately and moves to one of the three other locations: *Idle*, *Active_q*, or *BindingWait*, based on the current system state.
CheckNewTaskDL	This location was mentioned in description of the *NonBindingWait* location above. At this location the CA checks the deadline of a new instance of P, released while the system is waiting for a nonbinding precedent task to finish, and determines if the new P instance can safely wait for the remainder of the nonbinding wait period or not.
ERROR	*ERROR* is a special location that is only entered when an instance of P is found to be violating its deadline, i.e. when a clock associated with any P instance passes the corresponding deadline. The locations where a P instance's clock can surpass its own deadline are either when CA is in *BindingWait* location or in *Executing_p* location

To verify a task's schedulability for a simulation scenario, the entire scenario can be thought of as composed of similar subscenarios of duration equal to the least common multiple of the periodicities of the concerned tasks. From Table 2, this duration is $t_{lcm} = 40$ time units. All these subscenarios are identical

except the ones where occurrence of a sporadic task or event is possible. These exceptional scenarios can be called event-affected subscenarios. In our experiment, normalizing the time instants when the sporadic *Obstacle-EventHadndler* task can arrive within a subscenario, we got event-affected subscenarios where the sporadic event can occur at any time between 15 to 25 time units. An analysis of all the event-affected subscenarios and only one normal subscenario is enough to complete the schedulability analysis for the entire simulation run.

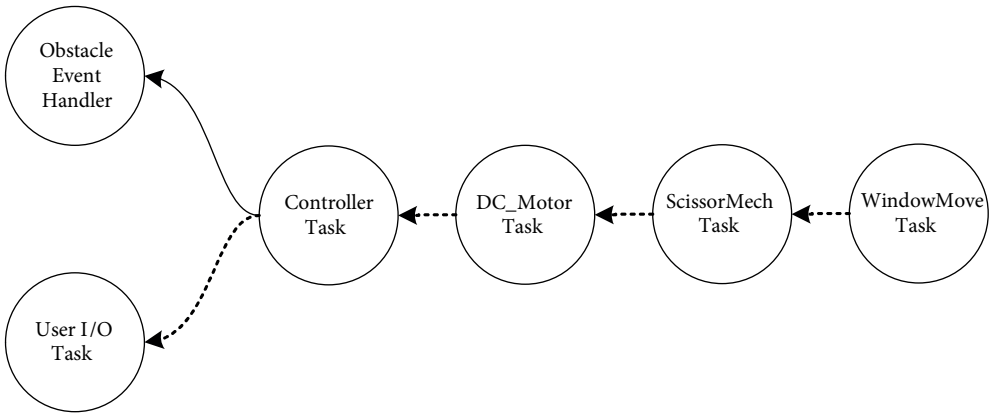

Figure 4. Graph showing precedent constraints among power window simulation tasks.

Table 2. Tasks in car's power window simulation.

Task name	Description	Attributes
Controller task	A task that emulates the actions of a microcontroller used in a car's power window	$WCET = 3$ time units $WCCT = 2$ time units $Period = 8$ time units
User input task	A task that periodically checks if the user has pressed window up or down button	$WCET = 4$ time units $WCCT = 1$ time units $Period = 5$ time units
Obstacle event handling task	A task that is triggered if an obstacle is detected in the path of the car window while it is moving up	$WCET = 1$ time units $WCCT = 1$ time units Sporadic: Can arrive between 495 and 505 time units
DC motor task	A task that simulates the working of a power window's DC motor	$WCET = 2$ time units $WCCT = 1$ time units Invoked from controller task
Scissor mechanism task	A task that simulates the mechanical assembly that moves a window up or down	$WCET = 3$ time units $WCCT = 2$ time units Invoked from DC motor task
Window move task	A task that simulates the window movements	$WCET = 3$ time units $WCCT = 1$ time units Invoked from scissor mechanism task

The CA is then instantiated by taking the *Controller* task as the *AnalyzedTask* while considering *UserInput* and *ObstacleEventHandler* as the precedent tasks. Since the problem of schedulability is now transformed into a state reachability problem, the only query that needs to be tested in order to check the schedulability is *Is there any system state where CA is in ERROR location?*, or formally, $E <> CheckingAutomaton.ERROR$. This query is verified using UPPAAL's verifier for each subscenario.

5.1.1. Functional verification result

The results of the analysis showed that for the subscenarios where the obstacle event occurred between the 16th and 18th time units or at either the 24th or 25th time unit, the *Controller* task was found schedulable. In rest of the subscenarios, the *Controller* task was found unschedulable. These results match with the ones obtained through manual analysis, verifying the functionality of our framework.

5.2. The performance experiments

The time and memory utilization while performing model checking or property verification of a task automata system in UPPAAL depends on the number of clock variables used and the duration for which the system is to be checked. Another factor that affects the verification performance greatly is the presence of indeterminate events in the system. Indeterminate events are basically automaton transitions that are declared to fire at an indeterminate time instant.

For simplicity in these performance experiments, we have assumed that there are no indeterminate events in the system. The experiments, therefore, are used to evaluate the affect of clock variable count and the time duration for which the system is checked. An example scenario with arbitrary functionality was used in which total number of tasks was 6. The task automata that defined the arrival patterns of these tasks used a total of 6 clock variables. These 6 clocks are in addition to the clocks defined in the CA, whose count depends upon parameters $maxQsize$ and $maxActiveNonBindingTasks$ as shown in Eq. (2). Since clock count is the primary source of complexity in a timed automata system, to see its effect on performance of schedulability analysis, one can increase the clock variables by varying either $maxQsize$ or $maxActiveNonBindingTasks$, or both. In these experiments we increased the number of clocks by keeping $maxQsize$ as 2 while increasing the $maxActiveNonBindingTask$ count.

5.2.1. Performance experiment results

A total of 35 experiments were conducted in order to evaluate the performance of our framework implementation. Each of these experiments used one out of five $maxActiveNonBindingTask$ values of 2, 4, 6, 8, or 10 and seven possible simulation durations that included $50, 100, 500, 1000, 1500, 2000$, and 3000. The time elapsed and the memory consumed during these tests were recorded and are presented here as surface plots in Figures 5 and 6. In each of the plots, the X-axis represents the number of $maxActiveNonBindingTasks$ and the Y-axis represents simulation time duration. The unit of simulation time is not fixed and depends upon the simulation time resolution for which the analysis is required. The Z-axis in Figure 5 shows the verification time in seconds while the same axis in Figure 6 shows the memory consumption in KB. As one would expect, the test results in both plots show that the time taken and memory consumed during the verification process increase with the increase in both duration of tested scenario and $maxActiveNonBindingTasks$, i.e. number of clocks. The actual values of verification time range from 0.26σ for a $maxActiveNonBindingTasks$ value of 2 and simulation duration of just 50 time units to 2355.74σ at the maximum tested values of 10 $maxActiveNonBindingTasks$ and 3000 time units. The memory consumed for the same scenarios is $10,824$ KB and $3,198,240$ KB, respectively.

6. Future work

We feel that there is room for simplifications in some aspects of the proposed framework, which will be the focus of our future work. More experimentation will also be done with the implemented framework focusing on

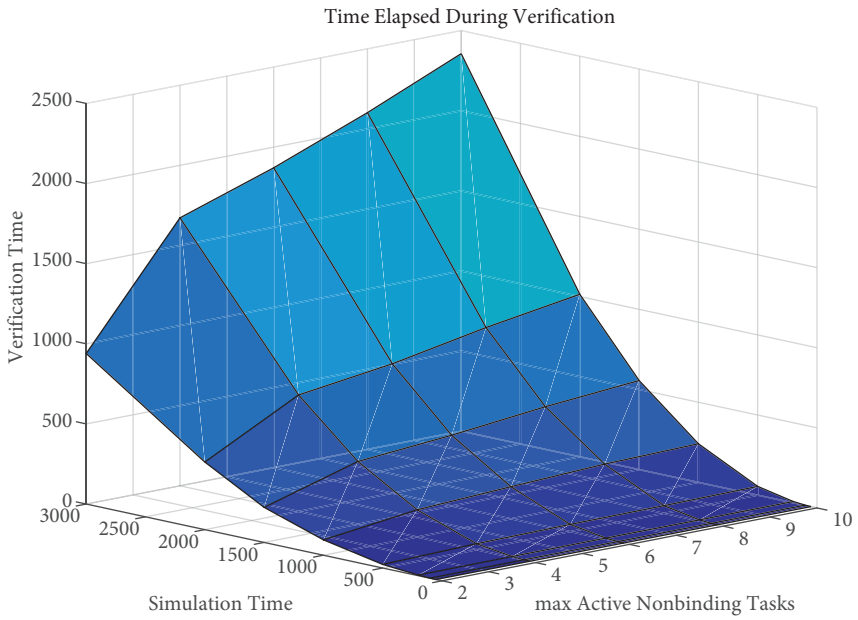

Figure 5. Verification time plot.

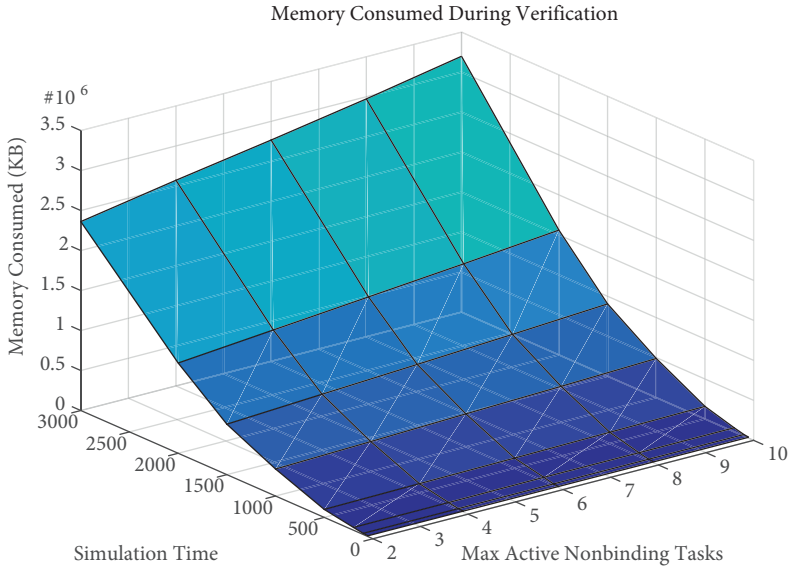

Figure 6. Memory consumption plot.

some real-world simulation scenarios adopted from industry. This will help us in refining the framework more and eventually in making it available for simulation designers to use it for their schedulability analysis and scheduler synthesis needs. Another area of work could be the automated instantiation of CA based on given task dependencies. In this work, the instantiated CA has been coded in UPPAAL manually.

7. Conclusion

This paper is the first ever attempt to address the problem of schedulability analysis in real-time simulation. We presented a timed automaton-based framework for analyzing the schedulability of real-time multiframe cosim-

ulations. A cosimulation system model based on the FMI 2.0 standard was presented with an assumption that each simulated task has its own dedicated processor core for execution. It was argued that the seemingly trivial problem of schedulability in such a cosimulation model becomes nontrivial when coupled with multiframing and precedence constraints.

A schedulability analysis framework was presented, which is an extension of an existing framework due to Fersman and Yi that was developed for the schedulability analysis of real-time systems. It was justified that the previous framework needs to be extended for the case of our cosimulation system model. The framework was then extended and related concepts were also presented.

The construction of the timed automaton, CA, used in the proposed schedulability analysis framework was presented and its UPPAAL implementation was tested. The functional accuracy test results showed that the framework detected the unschedulable scenarios correctly. Performance tests, on the other hand, provided the rate of increase in the time and memory requirements of our implementation with respect to clock variable count and tested time duration. These results can be interpreted to gain an idea about the practical limits of the framework.

References

[1] Gomes C, Thule C, Broman D, Larsen PG, Vangheluwe H. Co-simulation: a survey. ACM Computing Surveys 2018; 51: 49.

[2] Crosbie R. Real-time simulation using hybrid models. In: Popovici K, Pieter JM (editors). Real-Time Simulation Technologies: Principles, Methodologies, and Applications. Boca Raton, FL, USA: CRC Press. Taylor & Francis Group, 2013. pp. 4-31.

[3] Gear CW, Wells DR. Multirate linear multistep methods. BIT 1984; 24: 484-502.

[4] Ledin J. Simulation Engineering. Lawrence, KS, USA: CMP Books, 2001.

[5] Alur R, Dill D. A theory of timed automata. Theoretical Computer Science 1994; 126: 183–235.

[6] Abdedda Y, Asarin E, Maler O. Scheduling with timed automata. Theoretical Computer Science 2006; 354: 272-300.

[7] Abdeddaim Y, Kerbaa A, Maler O. Task graph scheduling using timed automata. In: International Parallel and Distributed Processing Symposium; Nice, France; 2003. p. 8.

[8] Fersman E, Yi W. A generic approach to schedulability analysis of real-time tasks. Nordic Journal of Computing 2004; 11: 129-147.

[9] Fersman E, Mokrushin L, Pettersson P, Yi W. Schedulability analysis of fixed-priority systems using timed automata. Theoretical Computer Science 2006; 354: 301-317.

[10] Fersman E, Krcal P, Pettersson P, Yi W. Task automata: schedulability, decidability and undecidability. Information and Computing 2007; 205: 1149-1172.

[11] Stigge M, Yi W. Graph-based models for real-time workload: a survey. Real-Time Systems 2015; 51: 602-636.

[12] Larsen KG, Pettersson P, Yi W. Uppaal in a nutshell. International Journal of Software Tools and Technology Transfer 2014; 1: 134-152.

[13] Behrmann G, David A, Larsen KG. A tutorial on Uppaal. In: Bernardo M, Corradini F (editors). Formal Methods for the Design of Real-Time Systems. SFM-RT 2004, Lectures Notes in Computer Science. Berlin, Germany: Springer, 2004. pp. 200-236.

[14] Norstrom C, Wall a. Timed automata as task models for event-driven systems. In: Sixth International Conference on Real-Time Computing Systems and Applications. RTCSA'99 (Cat. No. PR00306). Hong Kong, China: IEEE Computer Society, 1999. pp. 182-189.

[15] Fersman E, Pettersson P, Yi W. Timed automata with asynchronous processes: schedulability and decidability. In: Katoen JP, Stevens P (editors). Tools and Algorithms for the Construction and Analysis of Systems. Berlin, Germany: Springer; 2002. pp. 67-82.

[16] David A, Illum J, Larsen KG, Skou A. Model-based framework for schedulability analysis using Uppaal 4.1. In: Nicolescu G, Mosterman PJ (editors). Model-Based Design for Embedded Systems. Boca Raton, FL, USA: CRC Press, 2009. pp. 117-144.

[17] Boudjadar A, Kim JH, Larsen KG, Nyman U. Compositional schedulability analysis of an avionics system using Uppaal. In: International Conference on Advanced Aspects of Software Engineering; Constantine, Algeria; 2014. pp. 140-147.

[18] Shan L, Graf S, Quinton S, Fejoz L. A framework for evaluating schedulability analysis tools. In: Aceto L, Bacci G, Bacci G, Ingólfsdóttir A, Legay A et al. (editors). Models, Algorithms, Logics and Tools. Lecture Notes in Computer Science. Cham, Switzerland: Springer, 2017. pp. 539-559.

[19] Gonzalez Perez CA, Varmazyar M, Nejati S, Briand L, Isasi Y. Enabling model testing of cyber-physical systems. In: ACM/IEEE 21st International Conference on Model Driven Engineering Languages and Systems; Copenhagen, Denmark; 2018. pp. 179-186.

[20] Brandstetter V, Wehrstedt JC. A framework for multidisciplinary simulation of cyber-physical production systems. IFAC PapersOnLine 2018; 51: 809-814.

[21] Domenici A, Fagiolini A, Palmieri M. Integrated simulation and formal verification of a simple autonomous vehicle. In: International Conference on Software Engineering and Formal Methods; Cham, Switzerland; 2017. pp. 300-314.

[22] Fersman E, Mokrushin L, Pettersson P, Yi W. Schedulability analysis using two clocks. In: International Conference on Tools and Algorithms for the Construction and Analysis of Systems; Berlin, Germany; 2003. pp. 224-239.

[23] Krcal P, Stigge M, Yi W. Multi-processor schedulability analysis of preemptive real-time tasks with variable execution times. Lecture Notes in Computer Science 2007; 4763: 274-289.

[24] Blockwitz T, Otter M, Akesson J, Arnold M, Clauss C et al. Functional mockup interface 2.0: The standard for tool independent exchange of simulation models. In: 9th International Modelica Conference; Munich, Germany; 2012. pp. 173-184.

[25] Prabhu SM, Mosterman PJ. Model-based design of a power window system: Modeling , simulation and validation. In: IMAC-XXII: A Conference on Structural Dynamics, Society for Experimental Mechanics, Inc.; Dearborn, MI, USA; 2004.