Model-based Framework for Schedulability Analysis Using Uppaal 4.1

Alexandre David Jacob Illum Kim G. Larsen
Arne Skou
Center For Embedded Software Systems
Department of Computer Science
Aalborg Univeristy
Denmark

January 12, 2009

1 Introduction

Embedded systems involve the monitoring and control of complex physical processes using applications running on dedicated execution platforms in a resource constrained manner in terms of for example memory, processing power, bandwidth, energy consumption, as well as timing behavior.

Viewing the application as a collection of (interdependent tasks) various *scheduling principles* may be applied to coordinate the execution of tasks in order to ensure orderly and efficient usage of resources. Based on the physical process to be controlled, timing deadlines may be required for the individual tasks as well as the overall system. The challenge of *schedulability analysis* is now concerned with guaranteeing that the applied scheduling principle(s) ensure that the timining deadlines are met.

For single processor systems, industrial applied schedulability analysis tools include Sys Corporation, and RapidRMA from TriPacific based on Rate Monotonic Analysis. More recently SymTA/S has emerged as an efficient tool for system-level performance and timing analysis based on formal scheduling analysis techniques and symbolic simulation. These tools benefit from great succes in real-time scheduling theories; results that were developed in the

1970ies and the 1980ies, and are now well-established. However these theories and tools have become seriously challenged by the rapid increase in the use of multi-cores and multiprocessor system-on-chips (MPSoC).

To overcome the limitation to single-processor architectures, applications of simulation have been pursued, including – in the case of MPSoC – the ARTS framework (based on SystemC) [19, 20], the Daedaleus simulation tool [22] and Design-Trotter [21].

Though extremely useful for early design exploration by providing very adequate performance estimates for example memory usage and energy consumption as well as options for parallelizations, the use of simulation makes the schedulability analysis provided by these tools unreliable: though no deadline-violation may be revealed after (even extensive) simulation, there is no guarantee that this will never occur in the future. For systems with hard real-time requirements this is not satisfactory.

During recent years the use of real-time model checking has become an attractive and maturing approach to schedulability analysis providing absolute guarantees: if after model checking no violations of deadlines have been found, then it is guaranteed that no violations will occur during execution. In this approach, the (multiprocessor) execution platform, the tasks, the interdependencies between tasks, their execution times, and mapping to the platform are modeled as timed automata [3] allowing efficient tools such as Uppaal [24] to *verify* schedulability using model checking.

The tool TIMES [4] has been pioneering this approach, providing a rather expressive task-model called time-triggered architecture (TTA) allowing for complex task-arrival patterns, and using the verifition engine of Uppaal to verify schedulability. However, so far the tool only supports single-processor scheduling and limited dependencies between tasks. Other schedulability frameworks using timed automata as a modeling formalism and Uppaal as a backend include [8, 10, 11, 14, 23]. Also, related to schedulability analysis, a number of real-time operating systems (RTOS) have been formalized and analysed using Uppaal, [13, 17].

The MOVES analysis framework [16] presented in a different chapter of this book is closely related to the present chapter. Whereas the chapter on MOVES reports on the ability to apply Uppaal to verify properties and schedulability of embedded systems through a number of (realistic size) examples, we provide in this chapter a detailed – and compared with [5] alternative – account on how to model multiprocessor scheduling scenarios most efficiently, by making full use of the modeling formalism of Uppaal.

The chapter offers a UPPAAL modeling framework (download from [12]), that may be instantiated to suit a variety of scheduling scenarios, and that can be easily extended. In particular, the framework includes:

- A rich collection of attributes for tasks, including: off-set, best and worst case execution times, minimum and maximum interarrival time, deadlines, and task priorities.
- Task dependencies.
- Assignment of resources, for example processors or busses, to tasks.
- Scheduling policies including First-In First-Out (FIFO), Earliest Deadline First (EDF), and Fixed Priority Scheduling (FPS).
- Possible preemption of resources.

The combination of task dependencies, execution time uncertainty and preemption makes schedulability of the above framework undecidable [18]. However, the recent support for stopwatch automata [9] in UPPAAL leads to an efficient approximate analysis that has proved adequate on several concrete instances as demonstrated in [16].

The outline of the remainder of the chapter is as follows: In Section 2, we show the formalism of UPPAAL by the use of an example. In Section 3, we give an introduction to the types of schedulability problems that can be analyzed using the framework presented in Section 4. Following the framework, in Section 5 we show how to instantiate the framework for a number of different schedulability problems by way of an example system. Finally, we conclude the paper in Section 6.

2 UPPAAL And Its Formalism

In this section, we provide an introductory description of the UPPAAL modeling language.

2.1 Modelling Language

The tool UPPAAL is designed for design, simulation, and verification of real-time systems that can be modeled as networks of timed automata [2] extended with integer variables and richer user-defined data types. A timed

automaton is a finite-state machine extended with clock variables. The tool uses a dense time model of time so clock variables evaluate to real numbers. All the clocks progress synchronously.

We use in this section the train-gate example (distributed with the tool). It is a railway system that controls access to a bridge. The bridge has only one track and a gate controller ensures that at most one approaching train is granted access to this track. Stopping and restarting a train takes time. Figure 1 shows the model of a train in the editor of UPPAAL. When a train is approaching (`Appr`), it can be stopped before 10 time units otherwise it is too late and the train must cross the bridge (`Cross`). When it is stopped (`Stop`), it must be restarted (`Start`) before crossing the bridge.

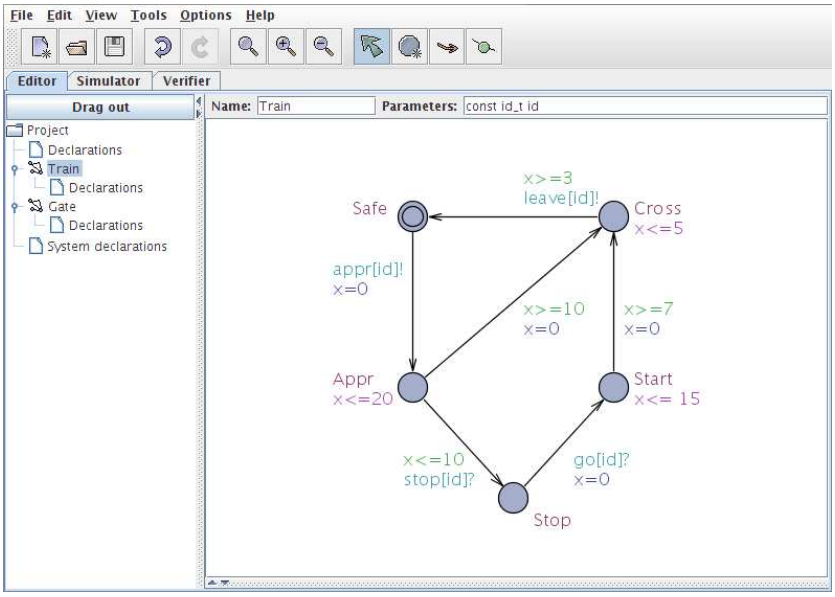

Figure 1: View of the train template in the editor.

The timing constraints associated with the locations are *invariants*. They give a bound on how long these locations can be active: A train can stay in `Appr` at most 20 time units and then must leave this location. Edges between locations have guards (`x<=10`) to constrain when they can be taken, synchronizations (`stop[id]?`) for communication, and updates (`x=0` to reset the clock x). Automata communicate with each other by means of channels. Here we have an array of channels and every train has its own id. The gate automaton selects a train and synchronizes with it with `stop[id]!`. In

Listing 1: Global declarations for the train-gate model.

```
1  const int N = 6;                    // Number of trains
2  typedef int [0, N-1] id_t;
3  chan appr[N], stop[N], leave[N];
4  urgent chan go[N];
```

Uppaal it is possible to declare arrays of clocks or any other type. Channels can be declared to be *urgent* to prevent delays if a synchronization is possible, or *broadcast* to achieve broadcast synchronization instead of handshake. Listing 1 shows the global declaration of the model with the channel declarations. A constant is declared to size the model to the desired number of trains. The train model here is in fact a *template* for trains. Trains are instantiated with a given id. In this case, having in the system declaration `system Gate, Train;` will instantiate an automaton for the Gate controller and all the possible trains ranging over their missing argument types. This is the *auto-instantiation* feature. It is possible to give specific argument values too. The type `id_t` is a user-defined type declared in listing 1. Uppaal supports more complex user-defined types such as structures. It is possible to combine arrays, structures, bounded integers, channels, and clocks.

Figure 2 shows the simulator with the Gate automaton. On the message sequence chart the different synchronizations between the automata are shown. The automaton has two main locations where the bridge is free or occupied. When a train is approaching it synchronizes with the gate and with a function call it is queued by the gate. If more trains are approaching then they are queued and stopped. Queuing followed by stopping is atomic, which is modeled by marking the location *committed*. Such a location forbids interleaving with other automata when it is active. A location can be marked *urgent* to mean that time cannot delay while it is active. After this, the gate dequeues a train and leaves it to cross the bridge. After that it will try to dequeue more trains and restart them with `go[front()]!`. Here a function call is used to return the front of the queue. The Gate "picks" a train with the *select* statement `e:id_t`. This allows the modeller to scale the model with the number of trains while still keeping the automaton compact. Listing 2 gives the complete local declarations of the Gate automaton. Uppaal supports C-like syntax that allows us to implement a queue here. One of the locations of the gate is marked "C".

Listing 2: Local declarations of the Gate template.

```
1  id_t   list [N+1];
2  int [0, N] len;
3
4  void enqueue(id_t element)  // Put an element at the end of the queue
5  {
6       list [len++] = element;
7  }
8
9  void dequeue()              // Remove the front element of the queue
10 {
11      int i = 0;
12      len -= 1;
13      while (i < len)
14      {
15           list [i] = list [i + 1];
16          i++;
17      }
18       list [i] = 0;
19 }
20
21 id_t front()               // Returns the front element of the queue
22 {
23    return list [0];
24 }
25
26 id_t tail()                // Returns the last element of the queue
27 {
28    return list [len - 1];
29 }
```

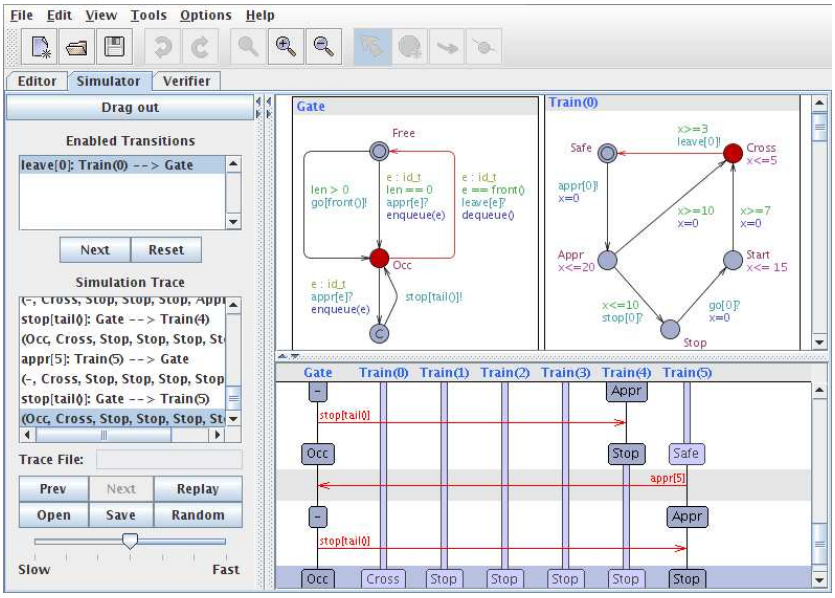

Figure 2: View of a simulation of the train-gate model showing the gate and one train.

Another feature of the language is the *scalar* type to define scalar sets. When these sets are used, the model checker takes advantage of their symmetry. Different variants of UPPAAL exist to other specific problem domains. UPPAAL-TIGA, [6, 15], is based on timed game automata and is targeted towards code synthesis. UPPAAL-CORA, [7, 25] is designed for cost-optimal reachability analysis. UPPAAL-PRO extends timed automata with probabilities.

Further features of UPPAAL include meta variables which can be used to store value such as regular variables, however, meta variables are not included in the state inclusion check when doing model checking. That is, two states are considered identical if all but the meta variables agree. Meta variables are declared using the 'meta' keyword (`meta int i`).

Finally, UPPAAL (as of version 4.1) supports stopwatches. Stopwatches are like clock variables, however, the progress of stopwatches can be set to either zero or one in automata locations, which is specified as an invariant (`x'==0` for clock `x`). When analyzing models using stopwatches, UPPAAL computes a finite overapproximation of the state space and is, thus, guaranteed to terminate even though the model checking problem for stopwatch

automata is, in general, undecidable. Checking properties such as avoidance of deadlocks can be meaningful for stopwatch automata, since if the overapproximation does not have a deadlock then neither will the real system. In Section 4, stopwatches are used to model preemptive schedulability problems using UPPAAL.

2.2 Specification Language

The specification language of UPPAAL is a subset of timed computation tree logic, [1] (TCTL). The following properties are supported: (i) A[] ϕ, (ii) E<> ϕ, (iii) A<> ϕ, (iv) E[] ϕ, and (v) $\phi --> \psi$, where ϕ and ψ are state predicates. The safety property (i) specifies that ϕ must be satisfied for all states. The reachability property (ii) specifies that there exists a path on which a state satisfies ϕ. The reachability property (iii) specifies that for all path there must be a state that satisfies ϕ. The liveness property (iv) specifies that there is a path on which all states satisfy ϕ. The "leads-to" property (v) specifies that whenever a state satisfying ϕ is reached then for all subsequent paths a state satisfying ψ is reached. In addition, UPPAAL can check for deadlocks with the property A[] not deadlock.

3 Schedulability Problems

At the core of any schedulability problem are the notions of tasks and resources. Tasks are jobs that require the usage of resources for a given duration after which tasks are considered done/completed. The added constraints to this basic setup is what defines a specific schedulability problem. In this section, we define a range of classical schedulability problems.

3.1 Tasks

A schedulability problem always consists of a finite set of tasks that we consistently will refer to as $T = t_1, t_2, ..., t_n$. Each task has a number of attributes that we refer to by the following functions:

- INITIAL_OFFSET: $T \rightarrow \mathbb{N}$
  Time offset for initial release of task.

- BCET: $T \rightarrow \mathbb{N}_{\geq 0}$
  Best case execution time of task.

- WCET: $T \rightarrow \mathbb{N}_{\geq 0}$
  Worst case execution time of task.

- MIN_PERIOD: $T \rightarrow \mathbb{N}$
  Minimum time between task releases.

- MAX_PERIOD: $T \rightarrow \mathbb{N}$
  Maximum time between task releases.

- OFFSET: The time offset into every period, before the task is released.

- DEADLINE: $T \rightarrow \mathbb{N}_{\geq 0}$
  The number of time units within which a task must finish execution after its release. Often, the deadline coincides with the period.

- PRIORITY: Task priority.

These attributes are subject to the obvious constraints that $\text{BCET}(t) \leq \text{WCET}(t) \leq \text{DEADLINE}(t) \leq \text{MIN\_PERIOD}(t) \leq \text{MAX\_PERIOD}(t)$. The periods are ignored if the task is non-periodic.

The interpretation of these attributes is that a given task t_i cannot execute for the first OFFSET (t_i) time units and should hereafter execute exactly once in every period of PERIOD (t_i) time units. Each such execution has a duration in the interval [BCET (t_i),WCET (t_i)]. The reason why tasks have a duration interval instead of a specific duration is that tasks are often complex operations that need to be executed and the specific computation of a task depends on conditionals, loops, etc. and can vary between invocations. Furthermore, for multi-processor scheduling, considering only worst-case execution times are not is insufficient as deadline violations can result from certain tasks exhibiting best-case behavior.

We say that a task t is *ready* (to execute) at time τ iff:

1. $\tau \geq \text{INITIAL\_OFFSET}(t)$

2. t has not executed in the given period dictated by τ.

3. All other constraints on t are satisfied. See 3.2 for a discussion on task constraints.

3.2 Task Dependencies

Task execution is often not just constrained by periods, but also by interdependencies among tasks., for example because one task requires data that is computed by other tasks. Such dependencies among a set of tasks $T = t_1, t_2, ..., t_n$ are modelled as a directed acyclic graph (V, E) where tasks are nodes (i.e., $V = T$) and dependencies are directed edges between nodes. That is, and edge $(t_i, t_j) \in E$ from task t_i to task t_j indicates that task t_j cannot begin execution until t_i has completed execution.

3.3 Resources

Resources are the elements that execute tasks. Each resource uses a scheduler to determine which task gets executed on a given resource at any point in time. Resources are limited by only allowing the execution of a single task at any given time.

Tasks are *a priori* assigned to resources. For a set of resources $R = r_1, ..., r_k$ and a set of tasks $T = t_1, ..., t_n$, we capture with the function ASSIGN $: T \rightarrow R$.

In a real-time system, resources function as different types of processors, communication busses, etc. Combined with task graphs we can use tasks and resources to emulate complex systems with such task interdependency on different processors. For example, if we want to model two tasks t_i and t_j with dependency $t_i \rightarrow t_j$, but the tasks are executed on different processors and t_j needs the results of t_i to be communication across a data bus, we introduce an auxiliary task t_{ic} that requires the bus resource and update the dependencies to $t_i \rightarrow t_{ic} \rightarrow t_j$. We illustrate this concept in ... **??**

3.3.1 Scheduling Policies

In order for a resource to determine which task to execute and which tasks to hold, a resource applies a certain scheduling policy implemented in a scheduler. Scheduling strategies vary greatly in complexity depending on the constraints of the schedulability problem. In this section we discuss a subset of scheduling policies for which we have included models in our scheduling framework.

- **First-In First-Out (FIFO)** Ready tasks are added to a queue in the order they become ready.

- **Earliest Deadline first (EDF)** Ready tasks are added to a sorted list and executed in the order of earliest deadline.
- **Fixed Priority Scheduling (FPS)** Each task is given an extra attribute, PRIORITY, and ready tasks are executed according to the highest priority.

Schedulers operate in such a manner that resources are never idle while there are ready tasks assigned to them. That is, as soon a task has finished execution a new task is set for execution.

3.3.2 Preemption

Resources come in two shapes, preemptive and non-preemptive. A non-preemptive resource means that once a task has been assigned to execute on a given resource, that task will run until completion before another task is assigned to the resource. Preemption means that a task assigned to a resource can be temporarily halted from execution if the scheduler decides to assign another task to the resource. We say that the first task has been preempted. A preempted task can later resume execution for the remainder of its duration.

Preemption allows for greater responsiveness to tasks that are close to missing their deadline, but that flexibility is on behalf of increased complexity of the schedulability analysis. The framework we define in the following section will include a model for schedulability analysis with preemption.

3.4 Schedulability

Now, we define what it means for a system to be schedulable. A system of tasks with constraints and resources with scheduling policies is said to be schedulable if no execution satisfying the constraints of the system violates a deadline.

4 Framework Model in UPPAAL

In this section, we will describe our UPPAAL framework for analyzing the scheduling problems defined in Section 3. The framework is constructed

such that a model of a particular scheduling problem consists of three different timed automata templates: A generic task template, a generic resource template, and a scheduling policy model for each applied policy. We will describe the templates in this order.

4.1 Modeling Idea

In order to best explain the framework models, we will provide an abstract scheduling model that will serve as a base for the framework models. The abstract scheduling model is based on the basic scheduling model defined in [7].

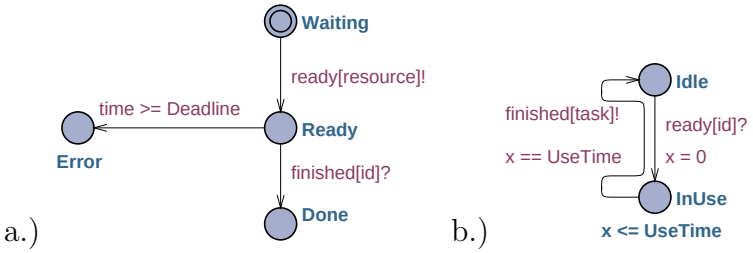

Figure 3: Abstract task and resource models.

This model, depicted in Figure 3 naturally divides the scheduling problem into tasks (Figure 3a) and resources (Figure 3b). Each task and resource has a unique identifier (id). Initially tasks are `Waiting` and when a task is ready to execute, this is signaled to the resource, to which the task is assigned, using the channel *ready*, indexed with the appropriate resource id (i.e., the variable resource). This moves the task to `Ready` where it remains until either the deadline has passed, in which case the task moves to `Error`, or it receives a signal that execution is complete via the channel *finished* indexed with the task id, in which case the task moves to `Done`.

Resources have two locations `Idle` and `InUse` indicating the state of the resource. Resources move from `Idle` to `InUse` upon a *ready* signal from a task, and return to `Idle` after the appropriate execution time. Resources signal that the task has finished execution using *finish*, indexed with the appropriate task.

With this model, schedulability can be verified with the following CTL query:

A[] forall(i : task_id) not Task(i).Error

That is, is it always the case that on all execution paths no task will ever be in the `Error` location?

This is the base of the framework model introduced in the following sections. The added complexity of these models is because of the handling of preemption, periods, and different scheduling policies. Before introducing the models, we will introduce some of the basic data structures used in the code.

4.2 Data Structures

For each scheduling problem with tasks $T = t_0, ..., t_n$ and resources $R = r_0, ..., r_k$ we define the following data types for convenience.

t_id: Task ids ranging from 0 to n.

r_id: Resource ids ranging from 0 to k.

time_t: Integer value between zero and the largest period among all tasks.

Having established the above data types we can move to more complex data types such as the data structure representing a task, which is called task_t and depicted in Listing 3. In other words, the task data structure holds all task attributes defined in Section 3.1. Note that the priority is given by pri as 'priority' is a reserved keyword in UPPAAL.

To specify a set of tasks, we create a global array called task of type task_t with one entry per task. The index of the array is the unique task identifier. See Section 5 for an example of task instantiation.

The final data structure is buffer_t which is the central data structure of the resource template. Each resource has a buffer that keeps track of the tasks ready to execute on a given resource and is sorted according to the respective scheduling policy. The buffer element is defined in Listing 4. Resource buffers are held in a global array called buffer with one index per resource.

The above data structures serve a the foundation of the template models defined in the following sections.

Listing 3: Task structure

```
typedef struct {
  time_t   initial_offset ;
  time_t   min_period;
  time_t   max_period;
  time_t   offset ;
  time_t   deadline ;
  time_t   bcet;
  time_t   wcet;
  r_id     resource ;
  int      pri ;
} task_t ;
```

Listing 4: Resource buffer.

```
typedef struct {
  int [0, Tasks] length ;
  t_id element[Tasks];
} buffer_t ;
```

4.3 Task Template

The task template serves as a model for both periodic and non-periodic tasks. The type of scheduling problem at hand is specified using the global Boolean parameter Periodic. This variable is tested in the task template to guarantee that tasks observe correct periodic or non-periodic behavior.

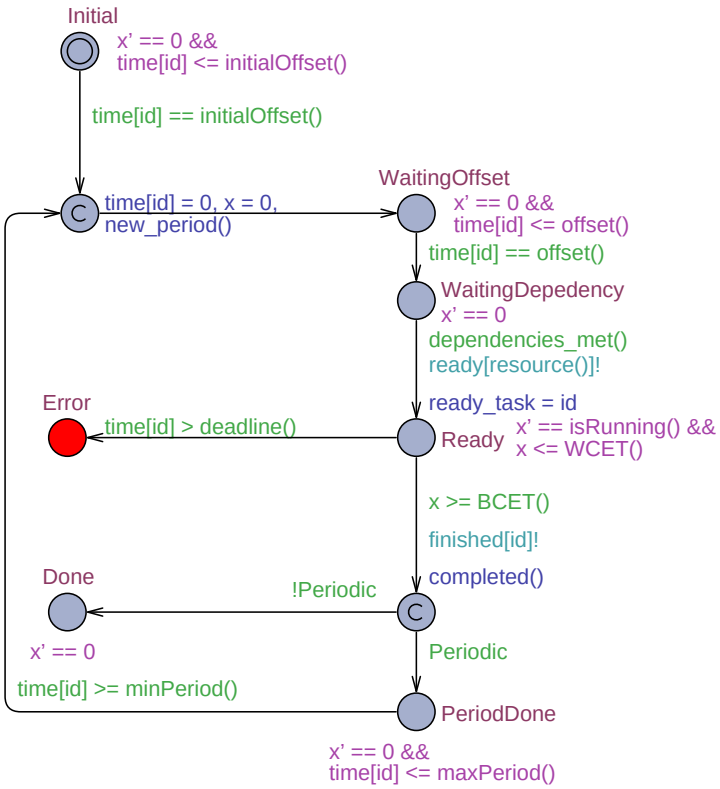

Figure 4: Task template. Takes argument id of type t_id.

The task template, depicted in Figure 4, takes a single parameter, namely the task id, which is used to index the task array. The basic structure of a task consists of five locations named:

- `Initial` (initial): The task is waiting for the initial offset time to elapse.
- `Waiting`: The task is waiting for certain conditions in order to be ready to execute. This location is actual split into two location representing

Listing 5: Function to determine clock rate of the x clock of a given task.

```
1 int [0,1] isRunning() { return ( buffer [ resource ()]. element [0] == id? 1 : 0);}
```

whether the task is waiting for the period offset to have elapsed or some other user-defined requirement. We return to the latter later in this section.

- `Ready`: The task is either executing or waiting to execute in the buffer of the respective resource.
- `Error`: The task did not manage to complete execution before the deadline.
- `Done`: The task has successfully completed execution within its deadline. This location is split into two locations. Which one of the two `Done` locations is used depends on whether the scheduling problem i periodic or not. In case the problem is non-periodic the done location is final, otherwise the done location is a holding location waiting for the next period.

For every task attribute, we define a local function to access that attribute in the global task array. That is, the function resource() returns task[id].resource, BCET() returns task[id].bcet, etc.

Each task uses two clock variables called time and x, where time represents the time since the beginning of the current period, and x represents how long the task has executed in the current period. The variable x is a local variable whereas time is global and, thus, indexed by the task id. As we are using stopwatch automata, note that the progress rate of x is set to zero in all but the `Ready` location. In `Ready`, the rate is determining whether the task is currently executing on the given resource. This is checked using the local function isRunning() defined in Listing 5.

On the other hand, time is always running and reset at every period, so that when time exceeds the deadline, the task can move to `Error`.

Upon entering `Ready`, the task updates the variable ready_task with the task id to indicate which task has signaled to the given resource. The resource utilizes this id when inserting the task in the queue for execution. We return to this in Section 4.4.

To allow for problem specific requirements, such as individual task constraints, the task template includes the following three functions new_period, dependencies_met, and completed. These functions can be used for a variety of problem specific purposes, the most obvious of which is the task graph dependency definition. How to model task graphs is illustrated below. new_period is executed on the edges leading into `Waiting` and used for updating data structures indicating that the task is beginning a new period, dependencies_met is tested in the guard leading from `WaitingDependency` to `Ready`, and completed is executed on the edge exiting `Ready` towards either `Done` location.

4.3.1 Modelling Task Graphs

To model task graphs using the customizable functions described above, we first need a global data structure to hold the task graph itself. In Listing 6, the task graph is modeled using a square Boolean dependency matrix where entry (i, j) dictates whether task i depends on task j. In the sample, we have four tasks $t_0, ..., t_3$ where task 0 depends on task 1 which depends on task 2 which depends on task 3.

Furthermore, we a have a Boolean array variable complete that determines whether a given task has finished execution. It is the responsibility of every function to reset the corresponding entry of this array in every period. In Listing 6, this is handled in the local task function new_period. The value of complete is set to true when a task finishes execution in the completed function call. Finally, tasks cannot enter the `Ready` location from `WaitingDependency` until the function dependencies_met evaluates to true. This function simply iterates the corresponding row of the task graph matrix and asserts that if the task depends on another task, the entry of the complete array for that task must be true.

With these steps, we have successfully modelled a task graph using our framework.

Note that extra care must be taken about specifying task periods when using task dependencies, as the meaning of a dependency can become unclear if the task periods are out of sync. The above handling of task graphs reset the complete variable on every new period, which is safe if dependent tasks have the same period, but not necessarily so if the tasks do not. Thus, a good rule of thumb is to only specify task dependencies between tasks that have identical periods and no non-determinism on periods.

Listing 6: Modelling Task Graphs

```
1  //Global declaration
2  const bool TaskGraph[Tasks][Tasks] = {
3   {0,1,0,0},
4   {0,0,1,0},
5   {0,0,0,1},
6   {0,0,0,0}
7   };
8  bool complete[Tasks];
9
10 //Task local declaration
11 void new_period() {
12   complete[id] = false;
13 }
14
15 bool dependencies_met() {
16   return forall (j : t_id) TaskGraph[id][j] imply complete[j];
17 }
18
19 void completed() {
20   complete[id] = true;
21 }
```

4.4 Resource Template

The resource template described in this section is identical for both preemptive and non-preemptive schedulers. The resource model is depicted in Figure 5 and the main difference from this model and the idea described in Section 4.1 is that the resource template does not include a clock. All timing is handled solely by the tasks.

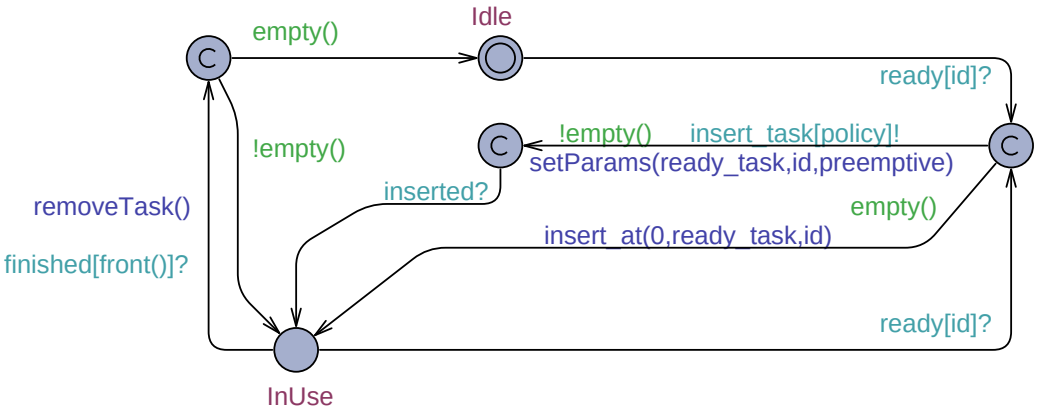

Figure 5: Resource. Takes arguments id of type r_id, preempt of type bool, and policy of type policy_t.

A resource takes three input parameters which are the resource id (id), a Boolean preempt to indicate whether the resource is preemptive, and a scheduling policy of integer type policy_t.

The template has the two main locations `Idle` and `InUse` to indicate the status of the resource. Idle resources are waiting for tasks to signal that they are ready via the *ready* channel of the resource. Upon receiving such a signal from either `Idle` or `InUse`, the resource will place the task at the front of its buffer if the buffer is empty. Emptiness is tested with the local function empty that reads the length of the respective buffer. Inserting a task in the buffer is done via the global insert_at function defined in Listing 7 and takes as argument a position, a task id, and a resource id. The inserting procedure moves all tasks in the buffer from the insert position one position back (lines 3-5) and then inserts the task at the desired position, and finally, updates the length of the buffer.

In case there is at least one element in the resource buffer already, the

Listing 7: Insert a task in the buffer of a resource at a given position.

```
1 void insert_at (int [0, Tasks] pos, t_id tid, r_id rid) {
2   int i;
3   for(i = buffer[rid].length; i > pos; i--) {
4     buffer[rid].element[i] = buffer[rid].element[i-1];
5   }
6   buffer[rid].element[pos] = tid;
7   buffer[rid].length++;
8 }
```

scheduling principle determines where in the buffer to place the ready task. Each scheduling principle is a separate model that is 'activated' by synchronization over the channel *insert_task* indexed which the scheduling policy. The scheduling policy needs information about the task, the resource, and whether the resource is preemptive in order to make the decision of where to place the incoming task in the buffer. This information is transferred via the setParams function that copies this information to meta variables that can be read by the scheduler. We return to this in Section 4.5.

The resource remains in the committed location waiting for the signal *inserted* indicating that the task has been inserted in the resource buffer according to scheduling policy. The waiting location is committed to guarantee that all task insertion processes should take place as an atomic operation. If a scheduling policy does not adhere to this constraint, the system deadlocks.

Upon inserting the task in the buffer according to the scheduling policy, the resource moves to the `InUse` location. The resource remains there until either the running task signals completion of execution through *finished* or another task signals that it is ready. In the latter case, the resource repeats the insertion process described above. In the former case, the resource removes the task from its buffer with the local function removeTask (see Listing 8). Depending on whether the resource buffer is empty the resource returns to either `Idle` or `InUse`.

The above resource template can be used to model different types of resources and supports an easy manner to implement specialized scheduling policies with minimal overhead. In the following section we describe how to model three types of scheduling policies in our framework.

Listing 8: Remove the front task from the resource buffer of the given resource.

```
1 void removeTask() {
2   int i = 0;
3   buffer[id].length--;
4   do {
5     buffer[id].element[i] = buffer[id].element[i+1];
6     i++;
7   } while (i < buffer[id].length);
8   buffer[id].element[buffer[id].length] = 0;
9 }
```

4.5 Scheduling Policies

In this section, we describe models for three types of scheduling policies: FIFO, FPS, and EDF. Common to all scheduling principles in our framework is that the scheduling policy is only called if there is at least one element in the resource buffer already. This precondition is guaranteed by the resource template by not calling the scheduling policy when the resource buffer is empty and instead inserting the ready task at the front of the buffer.

The parameters transferred to the scheduling policies via setParams function are stored in a struct meta variable called param. This way, the parameters are accessible to the scheduling policies without increasing the state space as meta variables are ignored for state comparison. The parameters are availble to the policy via param.task, param.prempt, and param.resource, respectively.

4.5.1 First-In First-Out

First-in first-out is the simplest scheduling policy as tasks are simply put into the resource buffer in the order in which they arrive. This implies that FIFO scheduling disregards whether the resource is preemptive or not. The model for the FIFO policy is depicted in Figure 6 where insertion of the task is handled on the edge synchronizing on *insert_task*.

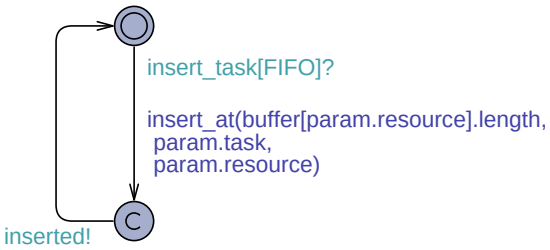

Figure 6: FIFO scheduling policy.

Listing 9: Function for task insertion according tho FPS.

```
1  void   insert_task_in_buffer  () {
2     t_id  t  = param.task;
3     r_id  r  = param.resource;
4     int  place  = (param.preempt ? 0 : 1);
5     int  i;
6     while(place<buffer[r]. length  &&
7             task[ buffer [r]. element[place ]]. pri >=task[t]. pri ){
8        place++;
9     }
10     insert_at (place, t, r);
11 }
```

4.5.2 Fixed Priority

The FPS scheduling policy model is similar to that of FIFO in structure, except that the task insertion is handled by the function call insert_task_in_buffer (Listing 9) instead of a call to insert_task. The function iterates (lines 6-8) through the resource buffer and compares the priority of the incoming task with the tasks in the buffer and sets a local variable place such that the incoming task has lower priority than all tasks in front of it and higher priority than all tasks behind it. In case the incoming task has the lowest priority of all tasks in the buffer, the iteration is terminated by reaching the end of the buffer and inserts the task here. Obviously, this method requires the buffer to be sorted. However, as the method is used for all insertion, except the first, the buffer remains sorted. When the place of insertion has been established, the task is inserted using the insert_at function call (line 9).

Note in Listing 9 that preemption is handled in line 4 where the first assumed place of the incoming task is either the first buffer entry, in case of preemption, or the second index, in case of no preemption. This code utilizes the precondition that the buffer has at least one element when the function is called.

4.5.3 Earliest Deadline First

The two scheduling policies above do not, in principle, need separate models as task insertion can be handled by a simple function call. However, because of UPPAAL modeling language constraints, scheduling policies such as EDF cannot be handled in our framework as a simple function call, but need to implement the insertion as a model. The problem lies in that to determine how far a task, i, is from its deadline, we need to look at the difference between task[i].deadline and time[i]. UPPAAL does not allow such comparison in function calls as it requires operations on the clock zone data structure. However, UPPAAL does allow such comparisons in guard expression and, thus, in order to find the position of an incoming task in a resource buffer we need to iterate through the buffer as model transitions comparing deadlines using guards.

In order to compare whether an incoming task i has earlier deadline than a buffered task j, we need to check the following:

$$\begin{aligned} &\text{task[i].deadline-time[i]} < \text{task[j].deadline - time[j]} \\ \Leftrightarrow\ &\text{time[i] -task[i].deadline} > \text{time[j] - task[j].deadline} \end{aligned}$$

Note that UPPAAL only allows subtraction of constants from clock values and not the reverse, hence the rewriting of the expression.

For convenience, we introduce a number of local variables and functions for the EDF model as defined in Listing 10. These are variables to hold the parameters that are transferred from the resource, a function readParameters to set the values of the local variables, and function resetVars to reset the variables after task insertion to minimize the search space.

Figure 7 depicts the model for EDF task insertion. Initially, the model reads the parameters and sets the initial assumed place variable of insertion according to whether the calling resource is preemptive or not (same as for FPS). Now, the deadline of the incoming task is compared to the deadline of the task at place. If place is either past the last element in the buffer or the incoming task has an earlier deadline than the task at place, the incoming

Listing 10: Local variables and function for the EDF scheduling policy.

```
1  int [0, Tasks]  place;
2  t_id  tid;
3  r_id  rid;
4  bool  preempt;
5  void  readParameters() {
6     tid  = param.task;  rid  = param.resource;  preempt = param.preempt;
7  }
8  void  resetVars() { place = tid = rid = 0; }
```

task is inserted at this place in the buffer with a call to the global function insert_at. Otherwise, place is incremented and the deadline check is performed again. Note that this is nothing more than a model implementation of a regular while-loop as used for FPS.

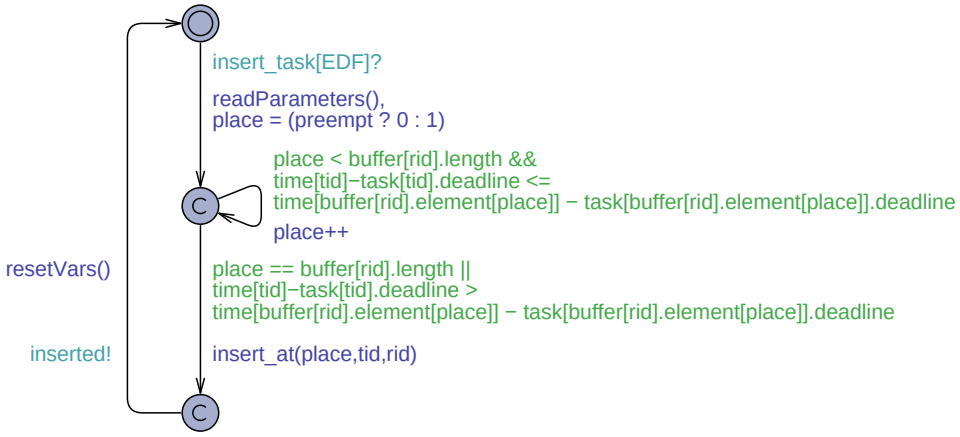

Figure 7: EDF scheduling policy

This concludes our model framework for schedulability analysis and in the following section we proceed by showing how to instantiate the framework and formulate schedulability queries.

Listing 11: Local variables and function for the EDF scheduling policy.

```
1 const bool Periodic = .. ;  // Periodic Scheduling?
2 const int Tasks = .. ;     // Number of tasks
3 const int Procs = .. ;     // Number of resources
4 const int MaxTime = .. ; // Maximum time constant
5 const task_t task[Tasks] = { ... };
```

Listing 12: Local variables and function for the EDF scheduling policy.

```
1 P1 = Resource(0, true/false , EDF/FPS/FIFO );
2 P2 = Resource(1, true/false , EDF/FPS/FIFO );
3  ...
4 system Task, P1, P2, ... , Policy_EDF, Policy_FPS, Policy_FIFO;
```

5 Framework Instantiation

In this section, we focus on instantiating our scheduling framework for standard scheduling problems. Instantiation requires data entry in the global declaration as well as in the system definition. The data needed in the global declaration is shown in Listing 11.

For a particular scheduling problem, the user needs to specify whether it is a periodic scheduling problem, how many tasks and resources there are, the maximum time constant for all tasks,[1] and the task attributes.

When defining the system, the user needs to specify the properties of the different resources, that is, preemption and scheduling policy.

In Listing 12, we show a sample system declaration with a number of resources (P1, P2,...). When declaring the actual system with the **system** keyword, all defined resources should be included with a single copy of each used scheduling policy. Note that all tasks are included with the single Task instantiation. This is because the parameter space of tasks are bound by the t_id type, so if no parameter is used, UPPAAL instantiates a copy of a task for every possible id.

[1] Used for UPPAAL state footprint reduction.

5.1 Schedulability Query

For any given scheduling problem, the schedulability of checking whether all tasks always meet their respective deadlines can be stated as in Section 4.1:

```
A[] forall ( i : t_id ) not Task(i).Error
```

In other words: "Does it hold for all paths that no task is ever in the `Error` location?". Note that UPPAAL using stopwatches creates an over-approximation of the state space meaning that if the above query is satisfiable, it is guaranteed that the system i schedulable under all circumstances. However, if the query is not satisfiable, this means that a counter-example has been established in the over-approximation. However, the system may still be schedulable since the counter-example is not necessary a feasible run of the original system.

5.2 Example Framework Instantiation

In this section, we provide a sample periodic schedulability problem and illustrate how to instantiate our framework for this problem. Parts of the schedulability problem is depicted in Figure 8 (left). It is a simple system with four tasks $T = t_0, ..., t_3$ and two resources P_0 and P_1 and with task dependencies given as a task graph. As indicated by dashed lines in the figure, tasks t_0 and t_1 are assigned to P_0 and tasks t_2 and t_3 are assigned to P_1.

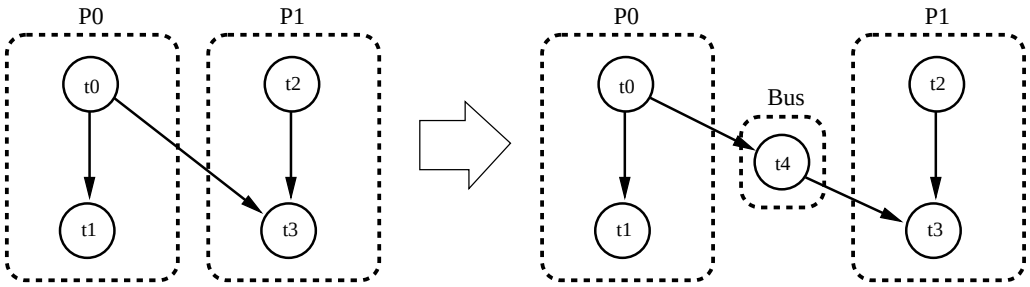

Figure 8: Sample schedulability problem.

Note that task t_3 depends on task t_0, but the tasks are assigned to different resources. This is a common scheduling situation and, usually, requires the

communication of data between resources using a transport medium such as a data bus. The communication of data takes time, but does not block either of the resources. In our framework, such a scenario can be modelled as illustrated in Figure8 (right) introducing a new task t_4 that requires a new resource Bus. The task t_4 is inserted into the task graph such that t_4 depends on t_0 and t_3 depends on t_4. We assume for the reasons explained in Section 3.2 that all tasks have fixed and identical periods. Thus, the attributes of t_4 should be set such that:

- INITIAL_OFFSET(t_4) = INITIAL_OFFSET(t_0)
- BCET(t_4): Specific to the bus.
- WCET(t_4): Specific to the bus.
- MIN_PERIOD(t_4) = MIN_PERIOD(t_0) Given our assumption about identical periods, the minimum and maximum periods are equal and the same for all tasks.
- MAX_PERIOD(t_4) = MIN_PERIOD(t_4)
- OFFSET$(t_4) = 0$. Offset not needed as task release is determined by completion of t_0.
- DEADLINE(t_4) = MIN_PERIOD(t_4). Could be tightened with respect to the deadline of t_3 and XCET(t_4).
- PRIORITY$(t_4) = 0$. Irrelevant as the bus is a FIFO resource.

The Bus resource will be modelled as a non-preemptive FIFO resource to mimic the behavior of a data bus. Assuming some specific best-case and worst-case execution times, periods, deadlines, etc., the schedulability problem described above can be instantiated as show in Listing 13.

Note in Listing 13 that the three tasks that depend on other tasks have been given an offset of one. This is for convenience, at it is the responsibility of individual tasks to reset their own entry in complete. Upon start of a new period without the offset, task t_1 could signal ready to the resource while reading an old value of complete for task t_0 before t_0 resets the value. Another way to handle this when all dependent tasks have the same fixed periods would be to let any task reset all values to complete when starting a new period. However, there is no general good way of handling this problem and it is left to the modeler to prevent unwanted situations.

Listing 13: Modelling Task Graphs

```
1  //Global Declaration
2  const bool Periodic = true;
3  const int MaxTime = 20;
4  const int Tasks = 5;                    // Number of tasks
5  const int Procs = 3;                    // Number of resources
6
7  const task_t task[Tasks] = {
8   {0,20,20,0,20,4,7,0,1},
9   {0,20,20,1,20,8,12,0,1},
10  {0,20,20,0,20,10,12,1,1},
11  {0,20,20,1,20,6,7,1,1},
12  {0,20,20,1,20,5,5,2,1}
13 };
14
15 const bool Depend[Tasks][Tasks] = {  // Task graph
16  {0,0,0,0,0},
17  {1,0,0,0,0},
18  {0,0,0,0,0},
19  {0,0,1,0,1},
20  {1,0,0,0,0}
21 };
22
23 //System Declaration
24 P0  = Resource(0,true,FPS);
25 P1  = Resource(1,true,FPS);
26 Bus = Resource(2,false,FIFO);
27
28 system Task, P0, P1, Bus, Policy_FPS, Policy_FIFO;
```

6 Conclusion

We have provided a framework allowing the modeling and analysis of a variety of schedulability scenarios. In particular, our framework supports multiprocessor systems, rich task models with timing uncertainties in arrival- and execution- times, possible dependencies, a range of scheduling policies, and possible preemption of resources. The support of approximate analysis of stopwatch automata in Uppaal 4.1 is key to the successful schedulability analysis.

Furthermore, the uncertainty on the periods used in our framework could be generalized to more general task arrival where a separate process determines the arrival of tasks. Such situations can be modeled using the structure of our framework by letting the starting of periods be dictated through channel synchronization with the model controlling arrival times. Even with such liberty, the overapproximation is still finite and termination is guaranteed.

The scheduling framework provided in this paper is structured such that adaptation can be made to accommodate other scheduling polices and inter-task constraints. The former can be achieved by adding another policy model similarly to the three built-in policies FIFO, FPS, and EDF. The latter is achieved through the use of the function calls new_period, dependencies_met, and completed.

References

[1] Rajeev Alur, Costas Courcoubetis, and David Dill. Model-checking for real-time systems. In *Proc. 5th IEEE Symposium on Logic in Computer Science (LICS'90)*, pages 414–425. IEEE Computer Society Press, 1990.

[2] Rajeev Alur and David Dill. Automata for modeling real-time systems. In *Proc. 17th International Colloquium on Automata, Languages and Programming (ICALP'90)*, volume 443 of *Lecture Notes in Computer Science*, pages 322–335. Springer, 1990.

[3] Rajeev Alur and David Dill. A theory of timed automata. *Theoretical Computer Science (TCS)*, 126(2):183–235, 1994.

[4] Tobias Amnell, Elena Fersman, Leonid Mokrushin, Paul Pettersson, and Wang Yi. Times - a tool for modelling and implementation of embedded systems. In Joost-Pieter Katoen and Perdita Stevens, editors,

TACAS, volume 2280 of *Lecture Notes in Computer Science*, pages 460–464. Springer, 2002.

[5] Jan Madsen Aske Brekling, Michael R Hansen. Models and formal verification of multiprocessor system-on-chips. *The journal of Logic and Algebraic Programming*, 77(1):1–19, 2008.

[6] Gerd Behrmann, Agnes Cougnard, Alexandre David, Emmanuel Fleury, and Didier Larsen, Kim G.and Lime. Uppaal tiga: Time for playing games! In *To appear in proc. of Computer Aided Verification (CAV'07)*, 2007.

[7] Gerd Behrmann, Kim G. Larsen, and Jacob I. Rasmussen. Optimal scheduling using priced timed automata. *ACM SIGMETRICS Perform. Eval. Rev.*, 32(4):34–40, 2005.

[8] Thomas Bœgholm, Henrik Kragh-Hansen, Petur Olsen, Bent Thomsen, and Kim G. Larsen. Model-based schedulability analysis of safety critical hard real-time java programs. In *JTRES '08: Proceedings of the 6th international workshop on Java technologies for real-time and embedded systems*, pages 106–114, New York, NY, USA, 2008. ACM.

[9] Franck Cassez and Kim Guldstrand Larsen. The impressive power of stopwatches. In Catuscia Palamidesi, editor, *11th International Conference on Concurrency Theory, (CONCUR'2000)*, number 1877 in Lecture Notes in Computer Science, pages 138–152, University Park, P.A., USA, July 2000. Springer-Verlag.

[10] Elena Fersman, Leonid Mokrushin, Paul Pettersson, and Wang Yi. Schedulability analysis of fixed-priority systems using timed automata. *Theor. Comput. Sci.*, 354(2):301–317, 2006.

[11] Elena Fersman, Paul Pettersson, and Wang Yi. Timed automata with asynchronous processes: schedulability and decidability. In *In Proceedings of TACAS 2002*, pages 67–82. Springer-Verlag, 2002.

[12] UPPAAL Scheduling Framework
. `http://www.uppaal.com/SchedulingFramework`, Jan. 2009.

[13] K. Godary, I. Augé-Blum, and A. Mignotte. Sdl and timed petri nets versus uppaal for the validation of embedded architecture in automotive.

In *Forum on specification and Design Language (FDL'04)*, Lille, France, September 2004.

[14] Nan Guan, Zonghua Gu, Qingxu Deng, Shuaihong Gao, and Gu Yu. Exact schedulability analysis for static-priority global multiprocessor scheduling using model-checking. In *Software Technologies for Embedded and Ubiquitous Systems*, Lecture Notes in Computer Science, pages 263–272. Springer, 2007.

[15] Uppaal Tiga Homepage. `http://www.cs.aau.dk/~adavid/tiga`, 2006.

[16] Aske Brekling Jan Madsen, Michael R. Hansen. A modelling and analysis framework for embedded systems. In this book.

[17] Jan Krakora and Zdenek Hanzalek. Timed automata approach to CAN verification. *INCOM*, 2004.

[18] Pavel Krcál and Wang Yi. Decidable and undecidable problems in schedulability analysis using timed automata. In Kurt Jensen and Andreas Podelski, editors, *TACAS*, volume 2988 of *Lecture Notes in Computer Science*, pages 236–250. Springer, 2004.

[19] J. Madsen, K. Virk, and M. J. Gonzalez. A systemC-based abstract real-time operating system model for multiprocessor system-on-chip. In *Multiprocessor System-on-Chip*. Morgan Kaufmann, 2004.

[20] S. Mahadevan, M. Storgaard, J. Madsen, and K. M. Virk. Arts: A system-level framework for modeling MPSoC components and analysis of their causality. In *13th IEEE International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*. IEEE Computer Society, sep 2005.

[21] Yannick Le Moullec, Jean-Philippe Diguet, Nader Ben Amor, Thierry Gourdeaux, and Jean Luc Philippe. Algorithmic-level specification and characterization of embedded multimedia applications with design trotter. *VLSI Signal Processing*, 42(2):185–208, 2006.

[22] Hristo Nikolov, Mark Thompson, Todor Stefanov, Andy D. Pimentel, Simon Polstra, R. Bose, Claudiu Zissulescu, and Ed F. Deprettere. Daedalus: toward composable multimedia MPSoC design. In Limor Fix, editor, *DAC*, pages 574–579. ACM, 2008.

[23] H. Sun. Timing constraints validation using uppaal: Schedulability analysis. In *DIPES '00: Proceedings of the IFIP WG10.3/WG10.4/WG10.5 International Workshop on Distributed and Parallel Embedded Systems*, pages 161–172, Deventer, The Netherlands, The Netherlands, 2001. Kluwer, B.V.

[24] UPPAAL. `http://www.uppaal.com`, Jan. 2005.

[25] UPPAAL CORA. `http://www.cs.aau.dk/~behrmann/cora/`, January 2006.