A Generic Simulator of Real-Time Scheduling Algorithms

Stéphane De Vroey
Université Libre de Bruxelles
Département d’Informatique CP 212, Boulevard du Triomphe
B-1050 Brussels, Belgium.

Joël Goossens
Université Libre de Bruxelles
Département d’Informatique CP 212, Boulevard du Triomphe
B-1050 Brussels, Belgium.
jgoosens@ulb.ac.be

Christian Hernalsteen
Université Libre de Bruxelles
Département d’Informatique CP 212, Boulevard du Triomphe
B-1050 Brussels, Belgium.
chernals@ulb.ac.be

Abstract

In this paper we describe a language for defining scheduling algorithms for hard real-time systems and a tool to simulate the behavior of such systems on a predefined task set. The language is suited for describing a real-time system composed of a task set, resources and a scheduling algorithm. The tasks can either be periodic or aperiodic, dependent or independent and the time constraints (e.g. deadlines) may be soft or hard. We consider two types of resources: the CPU and semaphores. Semaphores are used to describe the various possible dependences of the task set: shared memory, inter-task communication, devices,... We consider only mono-processor real-time systems. The third part of the system is given by the scheduling algorithm which is used to give resources (CPU and semaphores) to tasks.

1. Introduction

In this paper, we present a simulation language and a tool to simulate scheduling algorithms for real-time systems. First, we must introduce the peculiarities of real-time systems and scheduling algorithms.

In a real-time application, the validity of tasks depends not only on the results of computation but also on the time at which outputs are generated. Typically, a real-time task is characterized by its timing constraints (e.g. deadline constraint), precedence constraints, and resource requirements. These constraints can be soft or hard; if the violation of a constraint (e.g. a deadline) can be tolerated (with some cost), we speak in this case of soft real-time systems (e.g. soft deadline); in the opposite case, if the non respect of a constraint is fatal (e.g. flight control systems, nuclear power station control systems, railway control systems), we speak of hard real-time systems (e.g. hard deadline). In such systems we can find many kinds of tasks. These tasks can be classified according to the following task properties:

Periodic tasks: each periodic task \( \tau \) is characterized by a period \( T \), a deadline, an execution time and an offset which represents the time at which the first request of \( \tau \) occurs. The \( \tau \)’s requests are separated by \( T \). The execution of these tasks must finish before the deadline. In this case, the task’s deadline is considered to be hard.

Aperiodic tasks: each aperiodic task \( \tau \) is characterized by a deadline and execution time (sometimes also by the minimum inter-arrival time between the successive requests of \( \tau \)). Aperiodic tasks have irregular arrival times and either soft or hard deadlines.

The real-time task set can be composed of a set of independent tasks if their executions are not synchronized, or of a set of dependent tasks, if the tasks are synchronized, for ex-
ample by Ada rendez-vous, semaphores or monitors (due to accesses to shared data for instance) or more generally due to exclusive common resource usage.

The scheduler (or the scheduling algorithm) is another part of the real-time system; it orders assignment of CPU and the resources to the tasks. The function of the scheduling algorithm is to determine, for a given task set, a sequence of task step executions (a schedule). In particular, we are interested by scheduling algorithms which give (if any) a schedule for executing the tasks such that their timing, precedence and resource constraints are satisfied. We can distinguish three kinds of approaches: two priority based approaches: the static priorities [10] approach, the dynamic priorities [7] approach, and the cyclic executive approach [3]. With the static priorities approach, the priorities are computed once and for all for each task. This method uses an a priori knowledge of the system. With the dynamic priorities approach, the schedule is built during the execution, more exactly when the system changes (typically during the arrival of a task or at the end of a task). This approach needs a dynamic access to the task’s attributes (execution time, deadline,...). With the cyclic executive approach, the execution of the task set is repeated cyclicly from a pre-computed schedule. This schedule can be, for example, obtained by simulation from a priority based (static or dynamic) approach. For both priority based approaches, we must also distinguish preemptive and non-preemptive schedulers. With a preemptive scheduler, a task may be interrupted in order to give way to a task with a higher priority, while this may not happen with a non-preemptive scheduler.

In the next sections, we describe a language for defining scheduling algorithms for hard real-time systems and a tool to simulate the behavior of such systems on a set of predefined tasks. Let us notice, that in the field of real-time system simulation, the University of York has already defined a simulation tool: Stress [1]. Stress is a tool for analyzing and simulating the behavior of real-time applications, the scheduling problem is resolved by classical schedulers. Stress is suited to describe an existing and well known scheduling algorithm, but specifying a newly devised scheduler based on new concepts is very hard: it is not the purpose of this language. Moreover, Stress describes real-time systems at a lower level of abstraction than our language, the purpose being to define prototypes with implementation and application details (e.g. mail boxes, shared memory, inter-task communication,...); the main objectives of Stress and our tool are thus different. We propose in this paper a simulation language and a tool for analyzing scheduling algorithms for real-time systems.

2. The description language

A language has been defined to describe real-time systems composed by a scheduler, resources and a set of tasks. This language is quite basic but try to be as generic as possible to allow the definition of all kinds of systems. The description of a system is divided into three parts:

1. The set of tasks to schedule: This set represents the tasks for which we want to simulate their execution under the given scheduling algorithm. These tasks can be periodic or aperiodic. To define a task, a set of parameters must be defined as for example: the “period” or the “offset” of a periodic task, or the list of starting times of an aperiodic task. The behavior of a task is described by the sequential composition of independent execution blocks and synchronization points. The total execution time of a task is not directly given, but an execution time is associated to each execution block. Tasks are also characterized by a set of attributes which are managed by the simulator.

2. The resources: The most important resource is the CPU which is managed by the scheduler. Other kind of resources like shared memory, files or devices, are represented by semaphores. The allocation policy of a semaphore is by default FIFO, but it can also be defined by the specifier. It is thus possible to define complex allocation strategies of semaphores. Moreover semaphores can be used to model other kinds of synchronization mechanisms, like rendez-vous, message passing or monitors. Semaphores are used to specify dependencies between tasks.

2
3. **The scheduling algorithm:** The scheduler is responsible for the CPU allocation to the tasks and also, when specified, of the semaphore allocations. To describe a scheduler, two parts must be defined:

(a) The events on which the scheduler must be woken up: the description language gives a set of predefined events on which a scheduler can be woken up. The specifier must choose between these events which ones will actually trigger its awakening.

(b) The scheduling algorithm: the scheduler is responsible for the resource allocation, based on the system parameters and attributes. Some global variables are also available to define the allocation strategy, like the list of tasks ready to be executed or blocked by a semaphore, the task which has induced the wake-up event, or simply the current time.

The description language is not the same for the definition of the task and of the scheduler. There is only one statement in common to represent some execution time. We will now describe in a more deep way each of the three parts composing the specification of a system. The complete definition of the language is given in [5].

2.1. The set of tasks

A task is characterized by a type (periodic or aperiodic), a set of parameters and by a description of its behavior. This description specify the task resource utilization. The CPU utilization is represented by an execution time, and the other resources by the sequence of semaphore captures and releases.

The various statements describing a task are the following:

1. \([n]\): this statement represents the execution of an independent block of instructions. The parameter \(n\) represents the number of time units necessary for its execution. This statement can also be used in the description of the scheduler in order to give it an execution time.

2. \(P(\text{semajd})\): the task asks for the capture of a semaphore. If an occurrence of this semaphore is free, the task receives it and continues its execution. In the other case, the task is blocked.

3. \(V(\text{semajd})\): the task releases a previous captured semaphore. This semaphore can then be given to a blocked task, if any.

4. \(\text{Out}(\text{param})\): this is a simulation statement used to print some information to the default output channel in order to obtain traces of some task attributes during the simulation.

5. **Last Output:** a task can have two logical ends according to [4] (and two corresponding deadlines): the instant when it has produced its last output and the end of its execution. This statement represents the moment of the last output.

The instructions \(V()\) and \(P()\) take no time by default, but the specifier has the possibility to give it some execution time with the instruction “[\(n\)]” in the scheduling algorithms. This can only be done if the semaphore allocation is defined by the scheduler.

A set of parameters is associated to each task, some of those parameters are predefined in the language (like “Period”, “Offset” or “Deadline”) and others can be defined by the specifier (like for example a static priority). Adding parameters is done via the “TaskProto” construction (see Figure 1) and is applied to all the tasks of the system. Two deadlines can be specified for a task: “Data Deadline”, the deadline for the last output and “Main Deadline”, the deadline for the complete end of the task. Each of these deadlines can be defined as soft or hard. This differentiation is done via two boolean variables (“Main Hard” and “Data Hard”) which can be used by the scheduling algorithm to decide about the correct behavior to adopt on these deadlines. The scheduler can for example decide to stop the task execution when a hard deadline is reached and to continue when a soft one is reached. A task has also a set of attributes, managed by the simulator and available to the scheduler algorithm: the current status of the task (ready to execute, running, waiting for a semaphore, waiting for the next request occurrence), the cumulated time it has already been preempted, the starting time or the remaining execution time for its current occurrence, and so one. All these data can be useful to specify the scheduling algorithm.

![Figure 1. A simple task example](image)

2.2. The resources

The resources are represented by semaphores. A semaphore is created by defining; its name, the number of its initially free occurrences, and its allocation strategy which can be FIFO or specified by the user. There are two
ways to describe the allocation strategy of a semaphore; by defining an integer priority function based on the task parameters and attributes (the semaphore is given to the task with the higher value) or by defining a scheduler wake-up on “P(semia.id)” and “V(semia.id)” to decide for the semaphore allocation. In Figure 2 a semaphore

\text{Semaphore\ s1\ (1,\ -Main\_RTime)}

Figure 2. A semaphore declaration with a function based allocation strategy

“s1” is created with an allocation strategy, defined by an integer function depending on the remaining time before the deadline of the current task occurrence. In this example, the semaphore is given to the task occurrence having the nearest deadline. Each semaphore has also a set of attributes like the number of free occurrences, the list of tasks blocked on it or having an occurrence of it. Moreover, the specifier can define new parameters associated to semaphore (as for the tasks). All these attributes and parameters are available for the scheduling algorithm.

2.3. The scheduling algorithm

The scheduler is responsible for the allocation strategy of the CPU (and for the semaphores, when specified). To describe its behavior the specifier has access to some predefined data structures like boolean, integers or lists of these, and to a set of elementary statements like variable assignments, loops and conditional operations. Moreover it can use the value of all the parameters and attributes of tasks and semaphores, as well as the global data structures managed by the simulator (see section 3).

A scheduler is defined in two parts:

1. The first part describes the events on which the scheduler must be woken up. A set of predefined events is offered to the specifier, like for example: the begin of a new task occurrence, the reaching of a deadline, the end of a task, the asking for a semaphore (“PSEMA”) and so on. The specifier can also wake up the scheduler every n units of time. The user must choose one or more of these events to define its scheduler.

2. The second part describes the behavior of the scheduler when it is woken up. Its behavior can depend on its wake-up events and is described by various elementary statements and by some statements to manage the allocation of semaphores (if the semaphores are managed by the scheduler). The scheduler ends its execution on a “Stop” (end of simulation) or by a “Run(task.id)” which gives the CPU to the task task.id.

3. The simulator

The simulator interprets the scheduling algorithm on the defined task set and manages simulation data structures available to the scheduler. These data are for example: the current time, the list of semaphores, tasks waiting for a semaphore, or the list of active tasks (“Wait_CPU_List”) and so one. The simulator is responsible for the update of these various structures according to some simulation events like the beginning or the end of a task occurrence, or by the release of a semaphore.

4. Examples

Let us look to two simple examples to illustrate our description language. First (in section 4.1) we consider the case of the rate monotonic scheduler and (in section 4.2) we consider the case of the scheduling of dependent task sets.

4.1. The rate monotonic scheduler

We study here the rate monotonic scheduler (RMS) defined by Liu and Layland [7]. Basically it was defined for periodic and independent tasks. Moreover, the rate monotonic scheduler was defined for the special case where the deadline of each task coincides with its period (\(D_i = T_i \forall i\)) and for synchronous systems (all tasks are started at the same time). It is a static preemptive scheduler, which assigns a priority to each task as follows: given two tasks \(\tau_i, \tau_j\) and \(T_i, T_j\) their respective periods, if the task \(\tau_i\) has a higher priority than \(\tau_j\) then \(T_i < T_j\). When the system is running, it is always the highest priority task which is active (among all tasks waiting for the CPU). If a new task arrives with a higher priority than the active task, the active task is preempted and the new task becomes active. When a task ends, the waiting task with the highest priority (if any) becomes active.

Figure 3 represents this scheduler in our language: we first look for the running task (1). If there are other tasks waiting for the CPU (2), we compare the period of the task currently selected with the period of a task of the waiting_CPU_List (3). Then, the while loop selects then the task with the shortest period.

4.2. Example: the inversion problem

We will consider a scheduling algorithm which resolves the priority inversion problem. This problem arises when a task is blocked by a lower priority task (due to some resource sharing). Suppose that \(\tau_1, \tau_2,\) and \(\tau_3\) are three tasks arranged in descending order of priority. We assume that \(\tau_1\) and \(\tau_3\) share a resource. If the highest priority task \(\tau_1\) gains access first then the proper priority order is maintained; however, if the lower priority task \(\tau_3\) occurs first and gains ac-
Figure 3. The rate monotonic scheduler

```c
Declare t, tmp : TaskRef;
Declare bclk : Boolean;

Periodic Task T1 With
  Period := 5;
  MainDeadline := 5;
Begin
  [2];
End T1;

Periodic Task T2 With
  Period := 11;
  MainDeadline := 11;
Begin
  [6];
End T2;

SimulTime := 55;
DualDeadline := False;
Wakeup When MainDeadline Reached OR
Start_Task OR End_Task;

Scheduler Is
Begin
  If (Event_Type = Main_deadline_reached) Then
    Stop;
  Endif;
  t := NULL_Task;
  If (Not(Running_Task_List Is Empty)) Then
    t := Running_Task_List.First;
  Endif;
  If (Not(Wait_CPU_List.IsEmpty)) Then
    Wait_CPU_List.SetCurrentToFirst();
    If (t = NULL_Task) Then
      t := Wait_CPU_List.First;
    Endif;
    bclk := True;
    While (bclk) Do
      imp := Wait_CPU_List.Current;
      If (t.period > imp.period) Then
        t := imp;
      Endif;
      bclk := Not(Wait_CPU_List.IsLast);
      Wait_CPU_List.Next();
    Od;
  Endif;
End;
```

Static priority schedulers lead to arbitrarily large durations of priority inversion. A first solution to avoid too long priority inversions is to forbid the preemption of a task in a critical section. However this solution is only appropriate for very short critical sections, because it may create unnecessary blockings. There are other more efficient methods like the Priority Inheritance Protocol [11]; but they are too complex for the illustrative purpose of this paper (they may be expressed in our language however).

Figure 4 represents the scheduler algorithm implementing this simple solution. We will not give an exhaustive explanation of all the elements composing this scheduler; let us just explain two elements: "Event_Type" and "Event_Task" are two variables managed by the simulator. The first one contains the event type which has woken up the scheduler and the second one contains a reference to the task which has produced the event.

When a task asks for the capture of a semaphore, two situations are considered. First, if there is no critical section then the capture is satisfied. Second, if a critical section is in progress, the only task able to run and thus to capture a semaphore is the task in his critical section(s); so only this task is allowed to capture new semaphores. A semaphore’s capture can then be always satisfied (1). The variable “critical” is used to indicate that a critical section is in progress (2). The scheduler gives the semaphore to the task (3). Finally, it increments the number of semaphores used by this task (4). This value allows the scheduler to know when a task does not have anymore semaphores. When a task releases his lastest semaphore (5), the critical section is terminated. The last part of the algorithm looks for the next task to run: if a task is in his critical section then this is the only task allowed to run (6). Otherwise, the algorithm looks for all tasks ready to run and chooses the task with the highest priority (7).

5. The Tool

The simulator has been developed in C++ with the aid of the well known tools Flex and Bison [8]. So there must be no difficulties to use it on many platforms. The first version of the simulator has been implemented on a Linux system running on a 486DX2 66Mhz with 8Mb of RAM. The simulator is simple to use; it receives as input a description of a system in the described language; then it analyzes the source in order to build an internal representation of tasks,
semaphores, and the scheduling algorithm. The evolution of the system is generated as output during the simulation phase.

An interface, written in Tcl-Tk, has been developed to visualize the results of a simulation.

Figure 6 represents the evolution of the system described in figure 3 with the task attributes described in figure 5, figure 8 represents the evolution of the system described in figure 4 with the task attributes described in figure 7.

<table>
<thead>
<tr>
<th>Task Name</th>
<th>Period</th>
<th>Offset</th>
<th>Deadline</th>
<th>Execution Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>t1</td>
<td>5</td>
<td>0</td>
<td>5</td>
<td>2</td>
</tr>
<tr>
<td>t2</td>
<td>11</td>
<td>0</td>
<td>11</td>
<td>4</td>
</tr>
<tr>
<td>t3</td>
<td>30</td>
<td>3</td>
<td>30</td>
<td>7</td>
</tr>
</tbody>
</table>

Figure 5. Task set attributes, first example

<table>
<thead>
<tr>
<th>Task Name</th>
<th>Period</th>
<th>Offset</th>
<th>Deadline</th>
<th>Execution Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>t1</td>
<td>10</td>
<td>2</td>
<td>9</td>
<td>3</td>
</tr>
<tr>
<td>t2</td>
<td>15</td>
<td>1</td>
<td>11</td>
<td>4</td>
</tr>
<tr>
<td>t3</td>
<td>30</td>
<td>3</td>
<td>30</td>
<td>7</td>
</tr>
</tbody>
</table>

Figure 7. Task set attributes, second example

A box indicates that a task consumes a tick of processor time. The color of the box (indicated here by the grey level) indicates the imbrication level. A line at the bottom level indicates that the task is waiting for a resource (CPU or semaphore). Task releases and terminations are marked using circles. Release is indicated by a circle at the bottom of the execution boxes, and a termination by a circle at the top of the boxes. Deadlines are marked by a vertical arrow. An attempt to lock a semaphore is marked with a south-east arrow labeled with the first two characters of the semaphore’s name. A north-east arrow indicates the unlock of a semaphore.

6. Performance

In this section we present simulation results in order to evaluate the performance of our tools. We have applied our tools to a large number of representative task sets. We have chosen to apply our simulation tools with the rate monotonic scheduler [7] (defined in section 4.1) for various sets of periodic and synchronous tasks.

The task sets are generated by a pseudo-random algorithm, which constructs schedulable task sets by satisfying the sufficient condition defined by Liu and Layland based on the utilization factor $\sum_{i=1}^{n} \frac{C_i}{T_i} < n(\sqrt{2} - 1)$. Liu and Layland have shown that a set of periodic and synchronous tasks is schedulable iff all tasks reach their first deadline, hence we have simulated the system from time $t = 0$ to $t = \max_{i=1}^{n} T_i$. 

Figure 4. Scheduling of dependent task sets
Figure 6. Simulation results, first example

Figure 8. Simulation results, second example
by adding structures to algorithms (various kinds of loops, case statements, ...) or language constructs to ease the description of scheduling algorithms in several ways, for instance, by introducing new language constructs or the execution time, are defined by time intervals. In this way, it is possible to generate simulation traces for each class of tasks to study the behavior of a particular scheduling algorithm. A tool to produce statistical analysis of the traces can then be used to deduce some properties of the scheduling algorithms on the tested class of tasks. We shall not give more details on these two tools, since they are still under development.

### 8. Conclusion

We have presented in this paper a simulation language for real-time scheduling algorithms. This language is easy to use and can express a wide variety of schedulers. The user must only describe the scheduler, the resources and a set of tasks, and must not bother with simulation management. The simulator manages data structures available to the user and to ease the definition of his scheduler. Unlike other existing simulation tools like Stress, our language is well suited to express all kinds of schedulers, and not only classical ones. Some enhancement must still be done, and new tools must be developed in order to help the analysis of real-time schedulers. We think that this language and its associated tools form a good basis for further developments.

### References


