Cheddar : a Flexible Real Time Scheduling Framework

F. Singhoff, J. Legrand, L. Nana, L. Marcé
EA 2215, University of Brest
20, av Le Gorgeu
CS 93837, 29238 Brest Cedex 3
{singhoff,jlegrand,nana,marce}@univ-brest.fr

ABSTRACT

This paper describes an Ada framework called Cheddar which provides tools to check if a real time application meets its temporal constraints. The framework is based on the real time scheduling theory and is mostly written for educational purposes. With Cheddar, an application is defined by a set of processors, tasks, buffers, shared resources and messages. Cheddar provides feasibility tests in the cases of monoprocessor, multiprocessor and distributed systems. It also provides a flexible simulation engine which allows the designer to describe and run simulations of specific systems. The framework is open and has been designed to be easily connected to CASE tools such as editors, design tools, simulators, ...

Keywords

Real time scheduling, simulation tool.

1. INTRODUCTION

This paper presents a new Ada framework : Cheddar. Cheddar provides services to study the temporal behavior of real time applications. Most of the time, such kinds of applications have to meet temporal constraints such as response time, execution rate or deadline.

To check temporal constraints of an application made of concurrent tasks, system designers have to check if task temporal constraints are met. Since 1980, many models, methods and tools were proposed to check if a real time system fulfills its requirements (eg. Petri Net [20], Synchronous languages [11], ...). One of them, usually called "Rate Monotonic Analysis" is part of a larger set of quantitative methods : the real time scheduling theory. This theory helps the system designer to predict the timing behavior of a set of real time tasks with scheduling simulation and feasibility tests. Scheduling simulation requires, first to compute a scheduling on a given time interval and second, to look for timing properties in this computed scheduling. On the contrary, feasibility tests allow the designer to study a set of real time tasks without computing scheduling.

The first real time scheduling theory contributions were proposed 30 years ago [18]. The theory was strongly extended to cope with many application requirements [14] and was successfully used in many projects [21]. Nevertheless, only few tools were proposed to help system designers to automatically apply these results.

Rapid-RMA [26] and Timewiz [24] seem to be the most complete tools. These tools provide most of the classical scheduling algorithms and feasibility tests. They can be connected to different programs such as CASE tools (eg. Rational Rose), middlewares, operating systems. Unfortunately, they are not freely available.

From the academic community, a lot of tools were developed in the past but only few of them seem to be still maintained. Most of the time, they do not provide both simulation and feasibility tests.

TkRTS only focuses on task feasibility tests with various real time schedulers [19]. No scheduling simulation services are proposed and feasibility tests are limited to task response time.

On the contrary, YASA [4] and the tool of the Université Libre de Bruxelle [7] focus on scheduling simulation only. They do not provide feasibility tests. The flexible tool of the Université Libre de Bruxelle provides a language used to describe task models and various schedulers, but the language itself seems to be hard to use. By the way, all these tools were not designed to work with other programs.

Finally, the MAST [12] Ada framework also provides some basic feasibility tests and simulation services for fixed priority schedulers. The framework is portable enough and open source. Unfortunately, if simulations with a new scheduler or new task activation pattern must be done, the simulator has to be modified. Defining specific schedulers may be difficult to do for students.

The Cheddar project was motivated by the lack of free, flexible and open scheduling tools. Then, the design and the development of the Cheddar framework was conducted to fulfill three main requirements.

First, we aim at providing a framework which implements most of the classical real time scheduling theory methods [10, 14]. Implemented feasibility tests can be applied to monoprocessor and distributed systems with most of usual real time schedulers and task activation patterns. **Cheddar feasibility tests also focus on systems which are less studied by the community such as systems with buffers shared by tasks [15, 23] or task precedency**

constraints [5, 3, 25]. The program is distributed with publications which describe how to compute feasibility tests. Each result computed by Cheddar is displayed with the reference of the equation used to compute it. Cheddar can then be used by people or students to understand the foundations of real time scheduling theory.

Second, we aim at providing an open, portable and easy to use framework. The framework should remain easy to use even for students or people who do not have a large background on real time scheduling theory. By open, we mean a framework which is easy to connect with other programs (simulators, CASE tools, monitoring services from operating systems, ...). Even if the framework is object oriented, it exports a very simple interface. All data sent to the framework or produced by the framework are XML formatted. For portability and maintainability reasons, the framework is written in Ada. It runs on Solaris, Linux and win32 boxes but should run on every Gnat/GtkAda supported platform. The framework is distributed under the GNU General Public License (see http://beru.univ-brest.fr/~singhoff/cheddar).

Finally, we aim at providing a flexible framework. Since feasibility tests are only available for a few well known schedulers and task activation patterns, the Cheddar simulation engine is flexible enough to simulate systems with specific temporal behavior. We propose the use of an Ada-like language to extend the framework. User extensions expressed with this language are not compiled but interpreted by the framework at simulation time. This makes it possible for the designer to quickly write and test new scheduling features without having a deep knowledge of the framework design and of the Ada language.

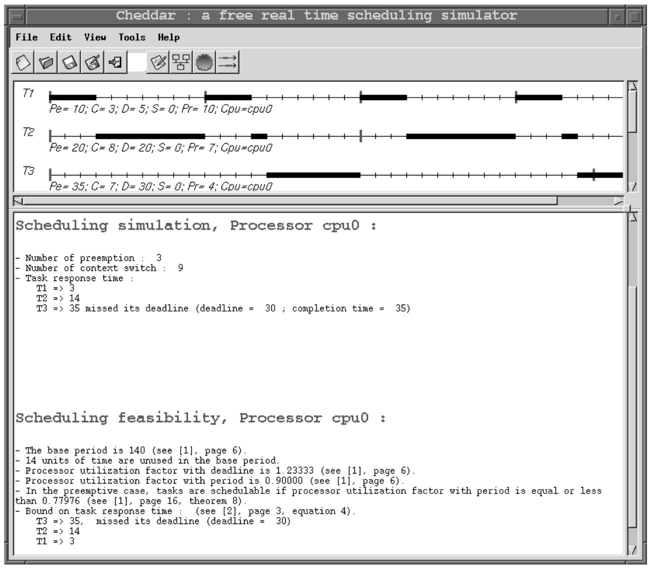
This paper is organized as follows. In section 2, we give an introduction to the real time scheduling theory and to the main services provided by the framework. Section 3 gives few details on the design of the scheduling simulation engine and explains how to extend it. Finally, we conclude and give future work in section 4.

2. FEASIBILITY TESTS AND SIMULATION SERVICES PROVIDED BY THE FRAMEWORK

With Cheddar, an application is defined by a set of processors, buffers, shared resources, messages and tasks (see Figure 2). In the most simple task model, each task periodically performs a treatment. This "periodic" task is defined by three parameters : its deadline (D_i), its period (P_i) and its capacity (C_i). P_i is a fixed delay between two wake up times of the task i. Each time the task i is woken up, it has to do a job whose execution time is bounded by C_i units of time. This job has to be ended before D_i units of time after the task wake up time.

From a set of tasks, two kinds of analysis can be performed : scheduling simulation and feasibility tests.

Scheduling simulation consists in predicting for each unit of time, the task to which the processor should be allocated. Checking if tasks meet their deadline can then be done by analyzing the computed scheduling. Figure 1 shows a set of 3 periodic tasks (T1, T2 and T3) respectively defined by the periods 10, 20 and 35, the capacities 3, 8, 7 and the deadlines 5, 20 and 30. In the top of the window, the scheduling simulation of the task set is displayed. These tasks are scheduled with a preemptive Rate Monotonic sched-


Cheddar : a free real time scheduling simulator
File Edit View Tools Help
T1
Pe= 10; C= 3; D= 5; S= 0; Pr= 10; Cpu=cpu0
T2
Pe= 20; C= 8; D= 20; S= 0; Pr= 7; Cpu=cpu0
T3
Pe= 35; C= 7; D= 30; S= 0; Pr= 4; Cpu=cpu0

```
Scheduling simulation, Processor cpu0 :

- Number of preemption :  3
- Number of context switch :  9
- Task response time :
   T1 => 3
   T2 => 14
   T3 => 35 missed its deadline (deadline =  30 ; completion time =  35)

Scheduling feasibility, Processor cpu0 :

- The base period is 140 (see [1], page 6).
- 14 units of time are unused in the base period.
- Processor utilization factor with deadline is 1.23333 (see [1], page 6).
- Processor utilization factor with period is 0.90000 (see [1], page 6).
- In the preemptive case, tasks are schedulable if processor utilization factor with period is equal or less
than 0.77976 (see [1], page 16, theorem 8).
- Bound on task response time :  (see [2], page 3, equation 4).
   T3 => 35,  missed its deadline (deadline =  30)
   T2 => 14
   T1 => 3
```

Figure 1: The main window of the Cheddar Editor

uler[1]. The Cheddar framework provides most of usual real time schedulers [6] such as Earliest Deadline First, Deadline Monotonic, Least Laxity First and POSIX schedulers with $SCHED\_FIFO$, $SCHED\_RR$, and $SCHED\_OTHERS$ queueing policies. After scheduling, information can be extracted from the simulation (worst/best/average case response time, worst/best/average case blocking time, number of preemptions, number of context switches, buffer utilization factor, end to end message communication delay ...).

For a given task set, if a scheduling simulation is very long to compute [17], feasibility tests can be applied instead (see the bottom of the window, Figure 1). Different kinds of feasibility tests exist. In the sequel, we present three of them : tests based on processor utilization factor, task response time designed to check task deadlines and tests based on buffer utilization factor designed to check buffer overflow.

The first feasibility test consists in comparing the utilization factor of a processor to a given bound. With a set of periodic tasks, the processor utilization factor can be computed with the formula $\sum_{i=1}^{n} \frac{C_i}{P_i}$, where n is the number of tasks on the processor. For instance, with a preemptive Rate Monotonic scheduler, Liu and Layland have shown that if the processor utilization factor is less than $n(2^{1/n} - 1)$, task temporal constraints are met [18].

The second feasibility test consists in comparing the worst response time of each task with its deadline. The response time is the maximum delay between the time the task becomes ready to run and the time the task ends its job. In [13, 2, 25], Joseph, Pandia, Audsley et al have shown that r_i, the worst response time of a task i, can be computed as follows :

$$r_i = \max_{q=0,1,2,\ldots} (J_i + B_i + w_i(q) - qP_i) \qquad (1)$$

[1]With such a scheduler, the task with the lowest period is the task with the highest priority.

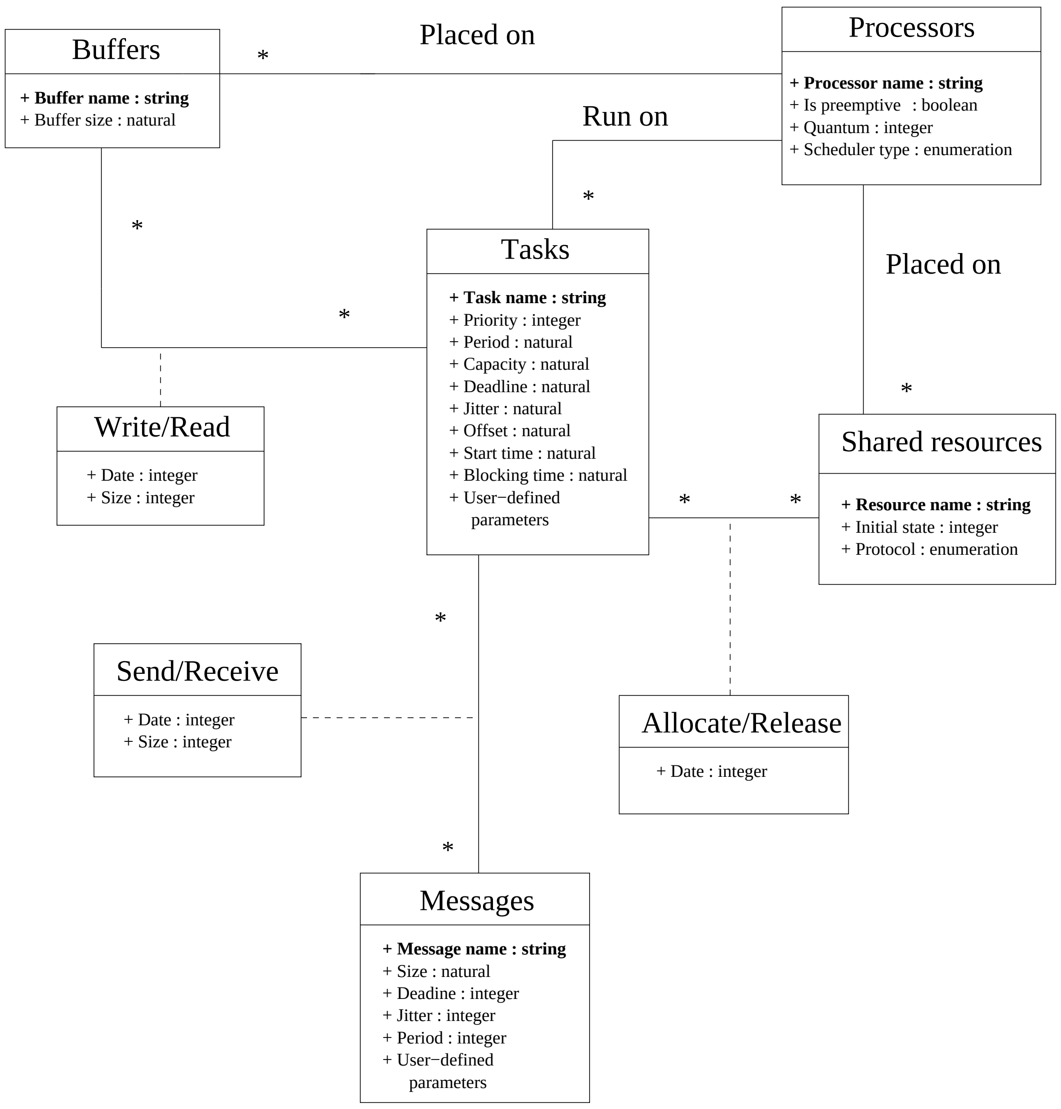


Figure 2: UML diagram of an application modeled with Cheddar

where

$$w_i(q) = (q+1)C_i + \sum_{\forall j \in hp(i)} \left\lceil \frac{J_j + w_i(q)}{P_j} \right\rceil C_j$$

and

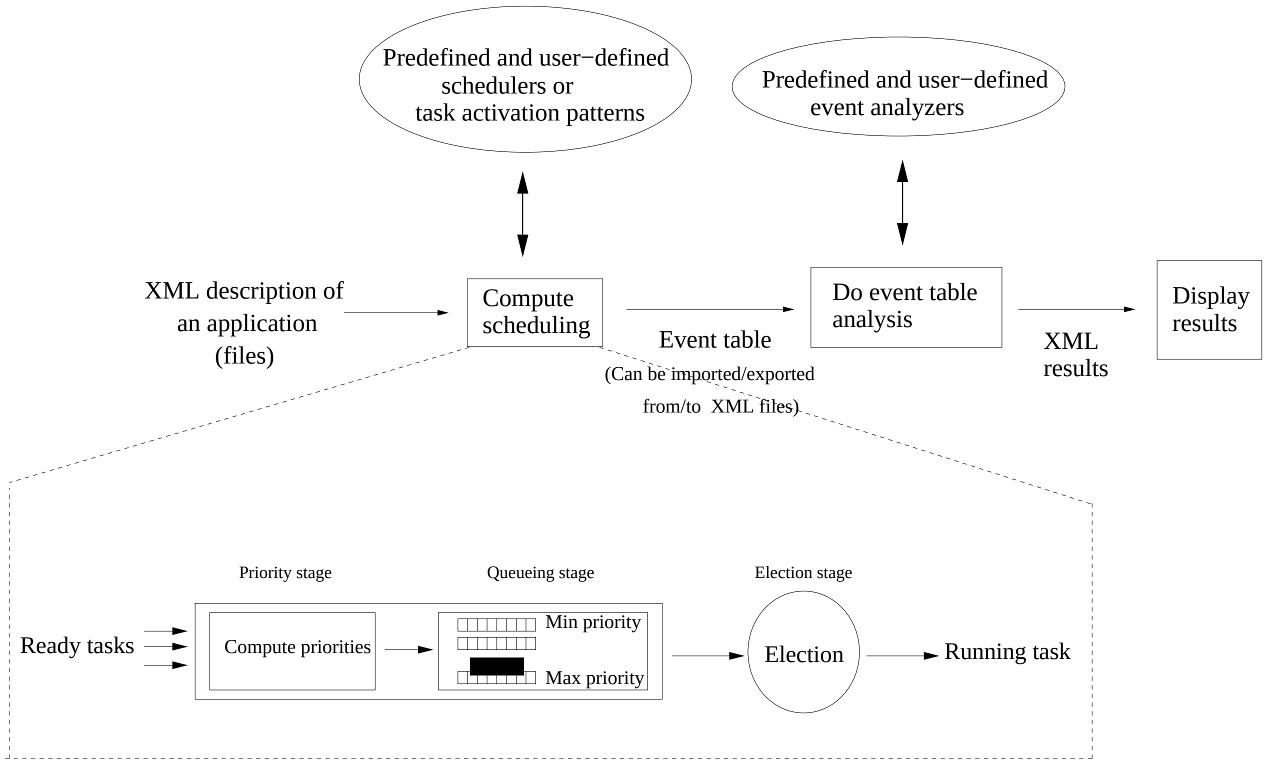
$$\forall q : w_i(q) \geq (q+1).P_i$$

where $hp(i)$ is the set of tasks which have a priority greater than i, B_i is a bound on shared resource blocking times and J_i is a bound on the task wake up time jitter.

The last feasibility test consists in finding a bound on buffer utilization factor in the case of a buffer shared by periodic tasks scheduled with a fixed priority scheduler. To understand this last test, we study the case of voice transmission service provided by the AAL2 layer of ATM networks. In AAL2/ATM, a producer sends audio packets at a fixed rate d. This throughput is expressed in cells per second, the protocol data unit of ATM networks. A bounded variable delay is required by each cell to go from the sender to the receiver. In an AAL2 communication service, the consumer should receive the cell at the same rate the producer sends it. Each received cell is then buffered during a sufficient amount of time to hide this variable transmission delay. In [8], it has been shown that the size of the buffer used to hide variable transmission delay is bounded by :

$$B = \left\lceil \frac{\delta}{d} \right\rceil \qquad (2)$$

Where δ is the maximum delay a cell stays in the buffer. We call this delay the maximum memorization delay. We can apply equation (2) to find bound on buffers shared by periodic tasks. For a buffer shared by N periodic producers and 1 periodic consumer, the buffer bound is [15] :



Predefined and user−defined schedulers or task activation patterns
Predefined and user−defined event analyzers
XML description of an application (files)
Compute scheduling
Event table
(Can be imported/exported from/to XML files)
Do event table analysis
XML results
Display results
Priority stage
Queueing stage
Election stage
Ready tasks
Compute priorities
Min priority
Max priority
Election
Running task

Figure 3: Design of the simulator engine

$$B = \max_{\forall y \geq 0} \left(\sum_{prod \in PROD} \left\lceil \frac{\delta + O_{prod}}{P_{prod}} \right\rceil - y \right) \qquad (3)$$

Where $PROD$ is the set of producers, P_{prod} the period of the producer $prod$ and O_{prod} the maximum delay between the wake up time of the consumer and the wake up time of the producer $prod$. This bound is based on the maximum memorization delay which is equal to $\delta = (y+2).P_{cons}$ for a given message i (y is the number of messages that may already be in the buffer before the message i is inserted; P_{cons} is the period of the buffer consumer). From equation (3) and for all possible values of y, it has been proven that for a buffer shared by 1 periodic consumer and N periodic producers, the buffer bound is [15]:

$$B = 2.N$$

if tasks are harmonics[2] and

$$B = 2.N + 1$$

in the other cases.

The three feasibility tests presented above are valid for preemptive fixed priority schedulers with periodic tasks. Of course, a large number of results exists for other schedulers (preemptive or not [10]) but also for more complex task sets such as tasks with PIP and PCP shared resources [22], or tasks with precedency constraints [5, 3, 25]. Different kinds of properties can also be checked with feasibility tests (eg. determinist properties such as the ones presented above, but also probabilist properties on task deadlines [16] or buffer utilization factor [23]). The system designer will find some of these extensions implemented in Cheddar. Since the number of feasibility tests is large, each result displayed is presented with the bibliography reference which describes the way to compute it. Finally, Cheddar binaries are distributed with papers introducing each feasibility test.

[2]A task set is said to be harmonic if and only if each task period is a positive integer multiple of all smaller task periods.

3. EXTENDING THE FRAMEWORK

Usual feasibility tests are limited to only few task models (mainly periodic tasks) and to only few schedulers. When an application built with a particular task activation pattern or scheduled with a particular scheduler has to be checked, feasibility tests are not necessarily available. In this case, the only solution consists in analyzing the scheduling simulation. Cheddar allows the user to design and easily build framework extensions to do simulation of user-defined schedulers or task activation patterns. By easy, we mean quickly write and test framework extensions without a deep understanding of the framework design and of the Ada language. We propose the use of a simple language to describe framework extensions. Framework extensions are interpreted at simulation time. As a consequence, they can be changed and tested without recompiling the framework itself.

Figure 3 gives an idea on the way the simulation engine is implemented in the framework. Running a simulation with Cheddar is a three-steps process.

The first step consists in computing the scheduling : we have to decide for each unit of time which events occur. Events can be allocating/releasing shared resources, writing/reading buffers, sending/receiving messages and of course running a task at a given time. At the end of this step, a table is built which stores all the generated events. The event table is built according to the XML description file of the studied application and according to a set of task activation patterns and schedulers. Usual task activation patterns and schedulers are predefined in the Cheddar framework but users can add their own schedulers and task activation patterns.

In the second step, the analysis of the event table is performed. The table is scanned by "event analyzers" to find properties on the studied system. At this step, some standard information can be extracted by predefined event analyzers (worst/best/average blocking time, missed deadlines ..) but users can also define their own event analyzers to look for ad-hoc properties (ex : synchronization constraints between two tasks, shared resources access order, ...). The results produced during this step are XML formatted and can be exported towards other programs.

Finally, the last step consists in displaying XML results in the Cheddar main window (see Figure 1).

3.1 Defining new schedulers or task activation patterns

Now, let's see how user-defined schedulers or task activation patterns can be added into the framework. Basically, all tasks are stored in a simple array called the "TCB array". A scheduler has to find a ready task to run from this array. To achieve this, schedulers are built like a three stages pipe-line with :

- **A priority stage.** For each ready task, a priority is computed.

- **A queueing stage.** Ready tasks are inserted into different queues. There is one queue per priority level. Each queue contains all the ready tasks with the same priority value. Queues are managed like POSIX scheduling queues : if a quantum is associated to the scheduler, queues work like the *SCHED_RR* scheduling queueing policy [9]. Otherwise, the *SCHED_FIFO* queueing policy is applied.

- **An election stage.** The scheduler looks for the non empty queue with the highest priority level and allocates the processor to the task at the head of this queue. The elected task keeps the processor during one unit of time if the designed scheduler is preemptive or during all its capacity if the scheduler is not preemptive.

Defining a new scheduler is simply overloading some of the pipe-line stages. Schedulers defined by users are stored together with XML application description files and are organized in "sections". Each section gives the statements that the simulator engine should run during simulation. A scheduler can be composed of a **start** section, a **priority** section, an **election** section and a **task activation** section.

In the **start section**, the designer can declare variables that he needs and can put the initialization code. Two variable families exist : static and dynamic variables. Static variables describe the studied application and are defined by XML tags of application description files (see Figure 2). Values of static variables never change during simulation and are read from the XML application description files. Dynamic variables are data collected by the framework at simulation time. They show the state of tasks, processors and the other elements of the studied application during the simulation.

All variables used in a scheduler should have a type. The framework provides two type families : scalar types (*double*, *integer*, *boolean*, ...) and arrays. An array is a type which stores one scalar data per task, message, buffer or shared resource. Vectorial operations can be done on this kind of variable.

The **priority section** contains the code necessary to compute task priorities. The code given here is called each time a scheduling decision has to be taken (at each unit of time for preemptive schedulers and when a task stops running in the non preemptive case).

The **election section**. In this section, the scheduling simulator engine decides which ready task should receive the processor for the next units of time.

Finally, **the task activation section** describes how tasks will be activated during a simulation. In Cheddar, 3 kinds of predefined task activation patterns exist : aperiodic, periodic and "Poisson process". Aperiodic tasks are activated only once and periodic or "Poisson process" tasks can be activated several times. In the case of periodic tasks, two successive task activations are delayed by a fixed amount of time called a *period*. In the case of Poisson process tasks, two successive task activations are delayed by an exponential random delay. If necessary, the designer can define more complex task activation patterns in this section (ex : sporadic task, randomly activated task, burst of activations, ...).

Now, let's see some scheduler examples. The most simple scheduler can be defined as below :

```
election_section:
    return min_to_index(period);
```

Figure 4: A simple Rate Monotonic scheduler

This first scheduler allocates the processor to the task with the smallest period : it's a Rate monotonic scheduler [6]. *period* is a predefined static variable initialized from the XML application description files modeling the studied application. To implement a Rate Monotonic scheduler, no priorities are computed and no variables are necessary. Then, the scheduler designer does not have to redefine the start and the priority sections.

The election section contains a return statement to inform the framework which task should run during the next units of time. The return statement uses a *min_to_index* function. This function performs a vectorial operation : it scans the TCB array to find the ready task with the minimum value for the static variable *period*.

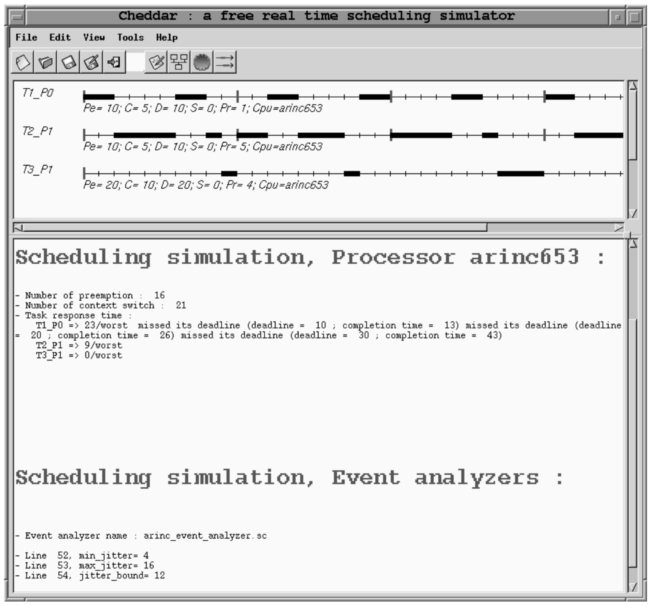
The second example is an ARINC 653[3] scheduler. It shows a complete scheduler example. An ARINC 653 system is composed of several partitions. A partition is a unit of program and is itself composed of processes and memory

[3]ARINC 653 is a standard interface for avionic applications.

spaces. A processor can host several partitions. Two levels of scheduling exist in an ARINC 653 system : partition scheduling and process scheduling.

1. **Process scheduling.** In one partition, processes are scheduled according to their fixed priority. The scheduler is preemptive and always gives the processor to the highest fixed priority ready task of the partition. When several tasks of a partition have the same priority level, the oldest one is elected.

2. **Partition scheduling.** Partitions share the processor in a predefined way. On each processor, partitions are activated according to an activation table. This table is built at design time and defines a cycle of partition scheduling. The table describes for each partition when it has to be activated and how much time it has to run.


Cheddar : a free real time scheduling simulator
File
Edit
View
Tools
Help
T1_P0
Pe= 10; C= 5; D= 10; S= 0; Pr= 1; Cpu=arinc653
T2_P1
Pe= 10; C= 5; D= 10; S= 0; Pr= 5; Cpu=arinc653
T3_P1
Pe= 20; C= 10; D= 20; S= 0; Pr= 4; Cpu=arinc653

```
Scheduling simulation, Processor arinc653 :

- Number of preemption :  16
- Number of context switch :  21
- Task response time :
    T1_P0 => 23/worst  missed its deadline (deadline =  10 ; completion time =  13) missed its deadline (deadline
=  20 ; completion time =  26) missed its deadline (deadline =  30 ; completion time =  43)
    T2_P1 => 9/worst
    T3_P1 => 0/worst

Scheduling simulation, Event analyzers :

- Event analyzer name : arinc_event_analyzer.sc

- Line  52, min_jitter= 4
- Line  53, max_jitter= 16
- Line  54, jitter_bound= 12
```

Figure 5: An example of ARINC 653 scheduling

Figure 5 displays an example of ARINC 653 scheduling. The system is made of 3 tasks hosted by one processor. The processor owns 2 partitions : partitions $P0$ and $P1$. The task $T1\_P0$ runs on the partition $P0$ and the two others run on the partition $P1$. Tasks have a fixed priority : $T2\_P1$ is the highest priority level task and $T1\_P0$ is the lowest one. The cyclic partition scheduling has to be done so that $P0$ runs before $P1$. In each cycle, $P0$ should run during two units of time and $P1$ should run during four units of time. To achieve this scheduling, a possible user-defined scheduler is the one given in Figure 6.

In Figure 6, *task_partition* is a static variable defined by the user and taken from XML application description files. For each task, *task_partition* stores the partition on which it should run. The variable *partition_duration* stores the partition cyclic activation table. Finally, *dynamic_priority* is computed according to the partition scheduling and to the task fixed priority.

```
start_section:
    partition_duration : array (tasks_range) of integer;
    dynamic_priority : array (tasks_range) of integer;
    number_of_partition : integer :=2;
    current : integer :=0;
    time_partition : integer :=0;

    – The partition scheduling table
    –
    partition_duration(0):=2;
    partition_duration(1):=4;
    time_partition:=partition_duration(current);

priority_section:
    if time_partition=0
        then current:=(current+1)
            mod number_of_partition;
        time_partition:=partition_duration(current);
    end if;

    – Choose the task with the highest priority
    – owned by the active partition
    –
    for i in tasks_range loop
        if tasks.task_partition(i)=current
            then dynamic_priority(i):=tasks.priority(i);
            else dynamic_priority(i):=0;
              tasks.ready(i):=false;
        end if;
    end loop;
    time_partition:=time_partition-1;

election_section:
    return max_to_index(dynamic_priority);
```

Figure 6: Process and partition scheduling in an ARINC 653 system

3.2 Looking for ad-hoc properties

In the same way that users can define new schedulers, Cheddar makes it possible to create user-defined event analyzers. These event analyzers are also written with an Ada-like language and interpreted at simulation time.

The event table produced by the simulator records events related to task execution and related to objects that tasks access. Event examples stored in this table can be :

- Events produced when a task becomes ready to run (event *task_activation*), when a task starts or ends running its capacity (events *start_of_task_capacity* and *end_of_task_capacity*),

- Events produced when a task reads or writes data from/to a buffer (events *write_to_buffer* and *read_from_buffer*),

- Events produced when a task sends or receives a message (events *send_to_message* and *receive_from_message*),

```
start_section:
    i : integer :=0;
    number_T1_P0 : integer :=0
    number_T2_P1 : integer :=0
    jitter_bound : integer;
    max_jitter : integer := integer'first;
    min_jitter : integer := integer'last;
    jitter : integer;
    T1_P0_end_time : array (time_units_range) of integer;
    T2_P1_end_time : array (time_units_range) of integer;

gather_event_analyzer_section:
    if (events.type = "end_of_task_capacity")
        then
            if (events.task_name = "T1_P0")
                then
                    T1_P0_end_time(number_T1_P0):=
                        events.time;
                    number_T1_P0:=number_T1_P0+1;
            end if;
            if (events.task_name = "T2_P1")
                then
                    T2_P1_end_time(number_T2_P1):=
                        events.time;
                    number_T2_P1:=number_T2_P1+1;
            end if;
    end if;

display_event_analyzer_section:
    while (i<number_T1_P0) and (i<number_T2_P1) loop
        jitter:=abs(T1_P0_end_time(i)-T2_P1_end_time(i));
        min_jitter:=integer'min(jitter, min_jitter);
        max_jitter:=integer'max(jitter, max_jitter);
        i:=i+1;
    end loop;
    jitter_bound:=abs(max_jitter-min_jitter);

    put(min_jitter);
    put(max_jitter);
    put(jitter_bound);
```

Figure 7: Example of user-defined event analyzer : computing task termination jitter bound

- Events produced when a task starts waiting for a busy resource (event *wait_for_a_resource*), allocates or releases a given resource (events *allocate_resource* and *release_resource*).
- ...

Each of these events is stored with the time it occurs and with information related to the event itself (eg. name of the resource, of the buffer, of the message, of the task ...).

The event table is scanned sequentially by event analyzers. User-defined event analyzers are composed of several sections : a **start** section, a data **gathering** section and an **analyse and display** section.

As user-defined schedulers, the **start** section is devoted to variable declarations and initializations.

The **gathering** section contains code which is called for each item of the event table. Most of the time, this section contains statements which extract usefull data from the event table, and store them for the event analyzer.

Finally, the **display** section performs analysis on data previously saved by the **gathering** section and displays the results in the main window of the Cheddar Editor.

Figure 7 gives an example of user-defined event analyzer. From an ARINC 653 scheduling this event analyzer computes the minimum, the maximum and the jitter on the delay between end times of two tasks owned by different partitions (tasks $T1\_P0$ and $T2\_P1$; see Figure 5).

4. CONCLUSION AND FUTURE WORK

This paper presents an Ada framework designed to check task temporal constraints. The framework implements most usual parts of the real time scheduling theory and has been mostly written for educational purposes. It provides two kinds of features : feasibility tests and a scheduling simulation engine. Feasibility tests allow the designer to predict task temporal constraints without computing the scheduling of the application. On the contrary, the simulation engine first computes the scheduling of the application and then, applies event analyzers to check/look for properties.

The framework has been designed to be open and flexible. By open, we mean a framework easy to connect with other CASE tools. Data sent to the framework or received from the framework are XML formatted.

By flexible, we mean a framework which can be extented to run specific schedulers, to do specific analysis or to schedule tasks with particular activation patterns. We propose the use of an Ada-like language. Schedulers, task activation patterns and event analyzers expressed with this language are not compiled but interpreted by the framework during simulation. This solution makes it possible to quickly write and test framework extensions without a deep knowledge of the framework design and of the Ada language.

The framework provides predefined services for basic task models and schedulers for both uniprocessor and multiprocessor architectures. Currently, the team is working on some important missing features.

First, Cheddar aims at doing analysis of applications that contain tasks sharing buffers. The current Cheddar release provides tools to predict buffer utilization when buffer producers and consumers are periodic tasks [15]. This feature will be extended in the next Cheddar release in order to take into account the case of randomly activated tasks [23].

Second, the simulator engine supports distributed scheduling simulation but users have to give a worst case communication delay for each message. As in the case of user-defined schedulers, a way to express specific message scheduling will be given.

Finally, we are looking for a method to do model-checking of user-defined framework extensions.

5. ACKNOWLEDGMENTS

We would like to thank the students who worked on this project and the reviewers of this article for their english advices.

6. REFERENCES

[1] Arinc. *Avionics Application Software Standard Interface*. The Arinc Committee, January 1997.

[2] A. N. Audsley, A. Burns, M. Richardson, and K. Tindell. Applying new scheduling theory to static priority pre-emptive scheduling. *Software Engineering Journal*, pages 284–292, 1993.

[3] J. Blazewicz. Scheduling Dependant Tasks with Different Arrival Times to Meet Deadlines. In. Gelende. H. Beilner (eds), Modeling and Performance Evaluation of Computer Systems, Amsterdam, Noth-Holland, 1976.

[4] J. Blumenthal, olatowski, J. Hildebrandt, and D. Timmermann. *Framework for validation and Analysis of Real time Scheduling Algorithms and scheduler implementations* . University of Rostock, Technical report available from http://yasa.e-technik.uni-rostock.de/, 2003.

[5] H. Chetto, M. Silly, and T. Bouchentouf. Dynamic Scheduling of Real-time Tasks Under Precedence Constraints . *Real Time Systems, The International Journal of Time-Critical Computing Systems*, 2(3):181–194, September 1990.

[6] F. Cottet, J. Delacroix, C. Kaiser, and Z. Mammeri. *Scheduling in Real Time Systems.* John Wiley and Sons Ltd editors, 2002.

[7] S. Devroey, J. Goossens, and C. Hernalsteen. A generic simulator of real-time scheduling algorithms. pages 242–249. Proceedings of the 29th Annual Simulation Symposium, New Orleans, Louisiana, April 1996.

[8] M. Gagnaire and D. Kofman. *Réseaux Haut Débit : réseaux ATM, réseaux locaux, réseaux tout-optiques.* Masson-Inter Editions, Collection IIA, 1996.

[9] B. O. Gallmeister. *POSIX 4 : Programming for the Real World* . O'Reilly and Associates, January 1995.

[10] L. George, N. Rivierre, and M. Spuri. Preemptive and Non-Preemptive Real-time Uni-processor Scheduling. INRIA Technical report number 2966, 1996.

[11] P. L. Guernic, T. Gautier, M. L. Borgne, and C. L. Maire. Programming real time applications with SIGNAL. INRIA-RENNES, Rapport numéro 1446, 1991.

[12] M. G. Harbour, J. G. Garca, J. P. Gutirrez, and J. D. Moyano. MAST: Modeling and Analysis Suite for Real Time Applications. pages 125–134. Proc. of the 13th Euromicro Conference on Real-Time Systems, Delft, The Netherlands,, June 2001.

[13] M. Joseph and P. Pandya. Finding Response Time in a Real-Time System. *Computer Journal*, 29(5):390–395, 1986.

[14] M. H. Klein, T. Ralya, B. Pollak, R. Obenza, and M. G. Harbour. *A Practitioner's Handbook for Real Time Analysis.* Kluwer Academic Publishers, 1994.

[15] J. Legrand, F. Singhoff, L. Nana, L. Marcé, F. Dupont, and H. Hafidi. About Bounds of Buffers Shared by Periodic Tasks : the IRMA project. In the 15th Euromicro International Conference of Real Time Systems (WIP Session), Porto, July 2003.

[16] J. P. Lehocsky. Real Time Queueing Theory. pages 186–194. Proceedings of the 17th IEEE Real-Time Systems Symposium (RTSS '96), Washington, DC, USA, December 1996.

[17] J. Leung and M. Merril. A note on preemptive scheduling of periodic real time tasks. *Information processing Letters*, 3(11):115–118, 1980.

[18] C. L. Liu and J. W. Layland. Scheduling Algorithms for Multiprogramming in a Hard Real-Time Environnment. *Journal of the Association for Computing Machinery*, 20(1):46–61, January 1973.

[19] J. Migge. *Real-time scheduling: a trajectory based model.* PhD Thesis, University of Nice Sophia Antipolis, November 1999.

[20] J. L. Peterson. *Petri Net theory and the Modelling of Systems.* Prentice Hall, 1981.

[21] SEI. The Rate Monotonic Analysis. Technical report, In the Software Technology Roadmap. http://www.sei.cmu.edu/str/descriptions/rma_body.html, September 2003.

[22] L. Sha, R. Rajkumar, and J. Lehoczky. Priority Inheritance Protocols : An Approach to real-time Synchronization. *IEEE Transactions on computers*, 39(9):1175–1185, 1990.

[23] F. Singhoff, J. Legrand, L. Nana, and L. Marcé. Extending Rate Monotonic Analysis when Tasks Share Buffers. In the DAta Systems in Aerospace conference (DASIA 2004), Nice, July 2004.

[24] TimeSys. *Using TimeWiz to Understand System Timing before you Build or Buy.* White paper, http://www.timesys.com/index.cfm?bdy=home_bdy_library.cfm, 2002.

[25] K. W. Tindell and J. Clark. Holistic schedulability analysis for distributed hard real-time systems. *Microprocessing and Microprogramming*, 40(2-3):117–134, April 1994.

[26] Tri-Pacific. *Rapid-RMA : The Art of Modeling Real-Time Systems.* http://www.tripac.com/html/prod-fact-rrm.html, 2003.