University of Pisa, Scuola Superiore Sant' Anna

Faculty of Engineering
Master Degree in Embedded Computing Systems

Master degree thesis

A real-time scheduling simulator for engine control applications

Laureando
Stefano Simoncelli

Supervisor
Prof. Ing. Giorgio Buttazzo

Correlator
Prof. Ing. Marco Di Natale

Academic Year 2013 - 2014

Contents

Summary **3**

1 Introduction **5**

2 Theoretical models **11**

2.0.1 Task Model 11

2.0.2 Rotation Source Model 15

2.0.3 Issues of this new scenario 16

3 Simulator Description **17**

3.1 Overview 17

3.2 Available Libraries 19

3.3 Metasim 19

3.3.1 Entities 20

3.3.2 Events 20

3.3.3 RandomVar 21

3.3.4 Simulation 22

3.3.5 BaseStat 22

3.3.6 Trace 23

3.4 RTLib 23

3.4.1 RTKernel 24

3.4.2 Scheduler, TaskModel 25

3.4.3 Task 26

3.4.4 Instruction 28

3.4.5 TaskStat 28

4 Task model: AVRTask **30**
4.1 Overview 30
4.2 Theoretical model 30
4.3 Implementation 31

5 Rotation Source Entity: AVRActivator **36**
5.1 Overview 36
5.2 Implementation 36
5.2.1 Tasks management 37
5.2.2 Physical engine management 40

6 Task Set Generator Entity: AVRGenerator **50**
6.1 Overview 50
6.2 Implementation 50
6.2.1 Random parameters generation 52

7 Experimental results **55**
7.1 Main programs 55
7.2 Scripts 62
7.3 Experiments 64
7.3.1 Schedulability 64
7.3.2 Maximum Normalized Tardiness 75

8 Conclusions **79**

Summary

Automotive engine control applications include computational activities released at specific crankshaft rotation angles, so their activation rate depends on the angular velocity of the engine. This phenomenon leads to the presence of variable-rate tasks that have to be executed on the Electronic Control Unit (ECU).
As the engine speed increases, the computational bandwidth required to the ECU tends to increase, so bringing the system in an overloaded condition potentially compromising the system performance as well as damaging the engine itself.
To deal with this issue, engine control tasks are designed such that their behavior is adapted as a function of the engine speed; this is done by performing a mode-change to adapt their computational demand so avoiding overload conditions. This brings to the definition of Adaptive Variable Rate (AVR) Tasks.

The goal of this Master Thesis is to design and develop a scheduling simulator handling AVR tasks in order to perform simulation experiments to better understand the behavior of such a new task model. In particular, experiments have been focused on the comparison between Dynamic-Priority scheduling (such as Earliest Deadline First scheduling) and Fixed-Priority scheduling.

The work is organized as follows: the thesis starts with an introduction about engine control; then scheduling problems and related works are presented. In the second chapter we report the theoretical models involved.
The simulator engine used in this work is RTLib 2.0 (Real-Time Library for Kernel Simulation) which relies on the Metasim 2.0 event-based simulator.
Since in this thesis we extended this simulation infrastrucure, the other three chapters are related to the implementation of the new components: namely AVRTask, AVRActivator (the simulator of the rotation source), AVRGenerator (AVR- task-set random

generator).
Then an extensive set of experimental results are reported, including a brief description of the 32-core virtual machine setup used to perform the experiments.
Finally we state the conclusion from the overall experience.

Chapter 1

Introduction

Engine control is a very important field of research in which automotive industries are interested, in order to get new models closer to their realistic requirements. Such kind of systems are characterized by computational activities that are triggered by specific crankshaft rotation angles and are designed to adapt their functionality based on the angular velocity of the engine. Although a few models have been proposed in the literature to handle such tasks, most of them are quite simplistic and do not allow expressing features that are presently used by the automotive industry. Automotive applications include tasks that are activated with different mechanisms. Some engine control tasks are activated by a timer at a fixed rate (periodic tasks), whereas other tasks are linked to the rotation of the crankshaft and are activated at specific rotation angles (angular tasks). The activation rate of such angular tasks is therefore proportional to the engine speed: the higher the engine speed, the higher the activation rate (Fig. 1.1).

Utilization values for such kind of tasks linearly increase with speed.

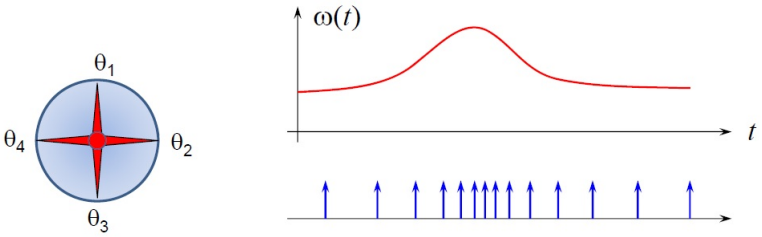

Figure 1.1: Angular Activations

A problem with such a type of activities is that, for high engine speeds, the system utilization can increase beyond a limit, generating an overload condition on the processor of the engine control unit (ECU) executing the application. If not properly handled, an overload can have disruptive effects on the controlled system, introducing unbounded delays on the computational activities, or even leading to a complete functionality loss (Fig. 1.2).

To prevent overload conditions, a common practice adopted in automotive applications

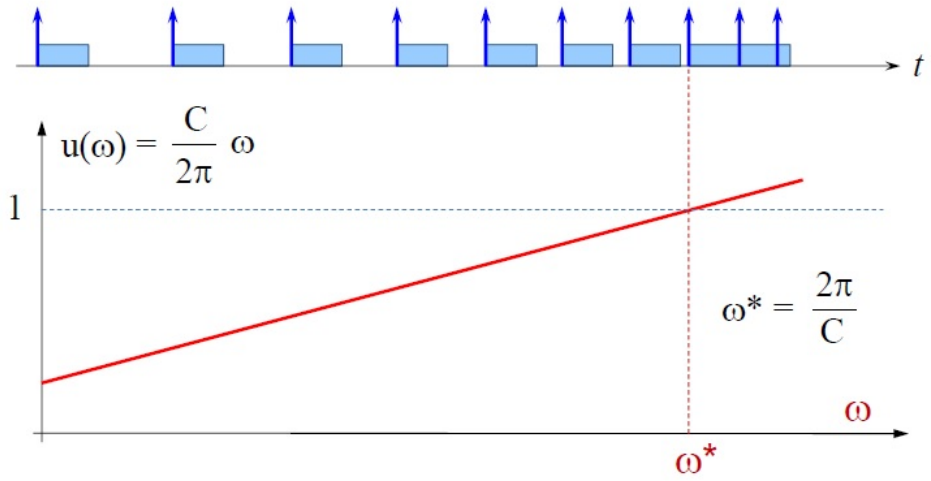

Figure 1.2: Overload

is to implement angular tasks in such a way they automatically decrease their computational requirements (by changing their functionality) for increasing speeds. To change their behavior, angular tasks are implemented as a set of execution modes, each operating within a specified range of rotation speeds. As a result, angular tasks have a variable activation rate (dependent on the engine speed) and a self-adaptive behavior implemented through a set of mode changes. For this reason, they are often referred to as adaptive variable-rate tasks (or AVR tasks) (Fig. 1.3).

More recently, the consideration of a fuel injection systems, as representatives of a possibly larger class of applications, has highlighted the limitations of the existing approaches and the need for a new type of task model and analysis. The general goal of a fuel injection system is to determine the point(s) in time and the quantity of fuel to be injected in the cylinders of an engine, relative to the position of each piston, which is in turn a function of the angular position of the crankshaft. In a reciprocating engine,

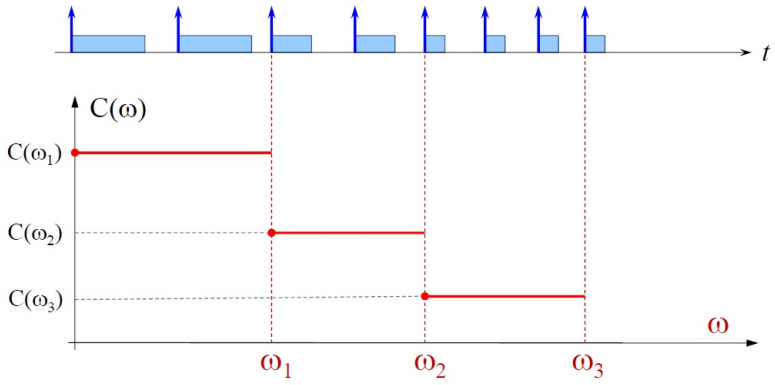

Figure 1.3: Adaptive Behaviour

the dead centre is the position of a piston in which it is farthest from, or nearest to, the crankshaft. The former is the top dead centre (TDC) while the latter is the bottom dead centre (BDC), as illustrated in FIg. 1.4. In a four-cylinder engine, the pistons are paired

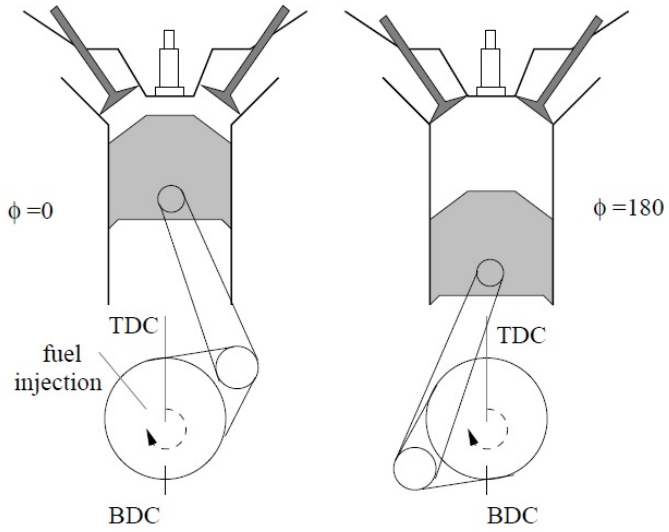

Figure 1.4: Crankshaft rotation period and motor phases

in phase opposition, so that, when two of them are in a TDC, the others are in a BDC. The TDC is the typical reference point, in the controller activities, for the functions and actions that need to take place within the rotation. These action include (among others) computing the phase (time relative to the TDC) of the injection and the quantity of fuel to be injected, but also checking whether the combustion occurred properly. Depending

on the structure of the engine control application, these functions are implemented in tasks that are activated at each TDC, that is twice every crankshaft rotation (pseudo-cycle) or even more frequently (half-TDC). Here In Fig. 1.5 is an example of ECU.

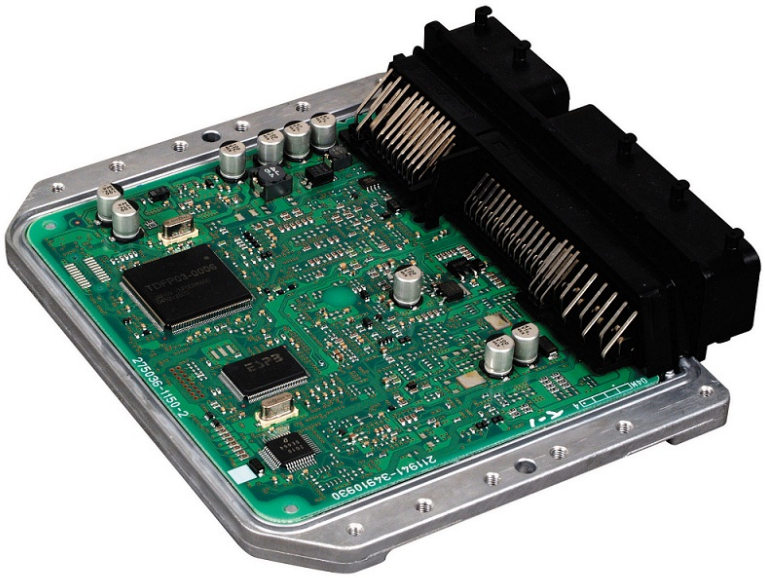

Figure 1.5: Engine Control Unit Sample

Given the peculiar characteristics of AVR tasks, the schedulability analysis of engine control applications cannot be addressed using classical real-time approaches. For instance, the elastic task model proposed by Buttazzo et al. [1], [2] is not suited to handle AVR tasks. In fact, in the elastic model, an overload condition is not handled by the task itself (by self-scaling its functionality), but through a global resource manager, which reduces the utilization of all the tasks by properly enlarging their periods. Similarly, classical mode change analysis [3], [4], [5] is not suited for engine control applications, because the activation rate of an AVR task changes continuously, thus an infinite number of modes would be required to describe all possible situations.

The problem of analyzing the real-time properties of engine control applications including AVR tasks has recently been considered in the literature by several authors, under different models and assumptions.

A task model suitable for engine control tasks with activation rates and execution times depending on the angular velocity of the engine has been proposed for the first time by Kim, Lakshmanan, and Rajkumar [6], who derived preliminary schedulability results

under simple assumptions. In particular, their analysis applies to a single rate-adaptive task with a period always smaller than the periods of the other tasks, and running at the highest priority level. In addition, they assume that all relative deadlines are equal to periods and priorities are assigned based on the Rate-Monotonic algorithm.
Pollex et al. [7] presented a sufficient schedulability analysis under fixed priorities, assuming a constant angular velocity. The analysis is formulated using continuous intervals, hence it cannot be immediately translated into a practical schedulability test, whose complexity has not been evaluated.
The dynamic behavior of AVR tasks under fixed-priority scheduling has been analyzed by Davis et al. [8], who proposed a sufficient test based on an Integer Linear Programming (ILP) formulation. Besides being only sufficient, their approach is based on a quantization of the instantaneous crankshaft rotation speed, which may introduce additional pessimism in the analysis to guarantee the safety of the test.
The exact interference of an AVR task under fixed priorities has been analyzed by Biondi et al. [9]. Here, the interference is analyzed using a search approach in the speed domain, where the complexity is contained by deriving a set of dominant speeds, which also avoid quantizing the instantaneous speed considered in the analysis.
The analysis of a mixed set of classical and AVR tasks under Earliest Deadline First (EDF) scheduling has been addressed by Buttazzo, Bini, and Buttle [10], but for AVR tasks related to independent rotation sources. They also provided a design method that allows computing the set of switching speeds at which modes have to be changed to keep the overall utilization below a desired bound. Although the results produced in the previous papers represent important milestones for the analysis of engine control systems, the task models used for the analysis are not always able to capture features that are currently adopted by the automotive industry in the implementation of AVR tasks. For example, in some work [10], tasks are considered to be linked to independent rotation sources, while in reality all the angular tasks related to engine control are linked to the same rotation speed and may be triggered at different rotation angles.
A typical engine control application includes both classical periodic tasks (with period ranging from a few milliseconds up to 100 ms) and a number of angular tasks activated every single, half, and quarter engine revolution. The assumption of independency clearly simplifies the analysis, but introduces an additional source of pessimism, considering situations that cannot actually occur when tasks are related to the same state variable. Also, all the previous papers consider mode transitions occurring at fixed speeds, while in

practice, to avoid frequent mode transitions when the engine speed is close to a switching speed, mode changes are implemented with hysteresis; therefore, the speed at which an AVR task adapts its functionality is not the same in acceleration and deceleration.
I have implemented in C++ the recently proposed AVRTask using an already available real-time scheduling simulator in Retis Laboratory (Sant' Anna): RTLib 2.0. The main contribution is to better understand how EDF and FP scheduling algorithms perform in such environment.
They have been already analyzed separately, especially in the case of fixed priority Biondi, Di Natale and Buttazzo [11], provide an exact Response-Time-Analysis for hybrid task set, but these are going to be the first general experimental results on this recent field of research.
In addition, the modules implemented in this thesis are taken into account also in a more general scenario, where the focus is not on the scheduling parameters, but on cyber-physical systems' aspects and functions.
In order to build this, more general system's simulation, also the CPU abstraction and task scheduling are needed, togheter with physical entities management and control theory applications: this is another field in which my work is involved.

Chapter 2

Theoretical models

This section introduces the notation adopted throughout the thesis and the models used for the rotation source and the computational tasks.
The rotation source considered in this paper is characterized by the following state variables:

- θ: the current rotation angle of the crankshaft;
- ω: the current angular speed of the crankshaft;
- α: the current angular acceleration of the crankshaft.

The rotation speed ω is assumed to be limited within a range $[\omega_{min}, \omega_{max}]$ and the acceleration α is assumed to be limited within a range $[\alpha-, \alpha+]$. In my simulations, the acceleration is assumed to be constant between two consecutive AVRTask activations.
In addition jerk is taken into account, limited within a range $[Jerk-, Jerk+]$ and used to update acceleration after an activation.

2.0.1 Task Model

The task set considered in this work consists of n real-time preemptive tasks $\Gamma = \{\tau_1, \tau_2, \tau_3, \tau_4 ... \tau_n\}$. Each task can be either a regular periodic task, or an AVR task, activated at specific crankshaft rotation angles. The subset of regular periodic tasks is denoted by Γ_P, whereas the subset of angular tasks is denoted by Γ_A, so that $\Gamma = \Gamma_P \bigcup \Gamma_A$ and $\Gamma_P \bigcap \Gamma_A = \emptyset$.
Whenever needed, an AVR task may also be denoted as τ^*.

Both types of tasks are characterized by a worst-case execution time (WCET) C_i, an interarrival time (or period) T_i and a relative deadline D_i.
However, while for regular periodic tasks such parameters are fixed, for angular tasks they depend on the engine rotation speed ω.
An angular task τ_i^* is characterized by an angular period Θ_i and an angular phase Φ_i, so that it is activated at the following angles:

$$\theta_i = \Phi_i + k\Theta_i, \quad for\ k = 0, 1, 2...$$

This means that the period of an AVR task is inversely proportional to the engine speed ω and can be expressed as:

$$T_i(\omega) = \Theta_i/\omega$$

An angular task τ_i^* is also characterized by a relative angular deadline Δ_i expressed as a fraction δ_i of the angular period ($\delta_i \in [0, 1]$). In the following, $\Delta_i = \delta_i\Theta_i$ represents the relative angular deadline.
In engine control, the angular phase Φ_i is relative to a reference position called Top Dead Center (TDC) corresponding to the crankshaft angle for which at least one piston is at the highest position in its cylinder.

The execution time of an angular task τ_i^* i is also a function of the rotation speed, since the task adapts its functionality to decrease its utilization at higher speeds. In the actual practice, angular tasks are implemented as a set of modes, each operating within a specified range of speeds; in addition, a hysteresis is introduced to avoid frequent mode changes when the engine speed is around to a switching speed. Therefore, the computation time of an angular task can be described by a step function consisting of M_i execution modes, where each mode $m = (1, ..., M_i)$ is defined by a computation time C_i^m and a speed range $[\omega_i^{m-}, \omega_i^{m+}]$. An example of C function is in Fig.2.1:

Note that, when considering hysteresis in mode changes, the computation time of an AVR task depends not only on the value of ω but also on the current mode m, leading to leading to two alternative mode-change behaviors, characterized by the following

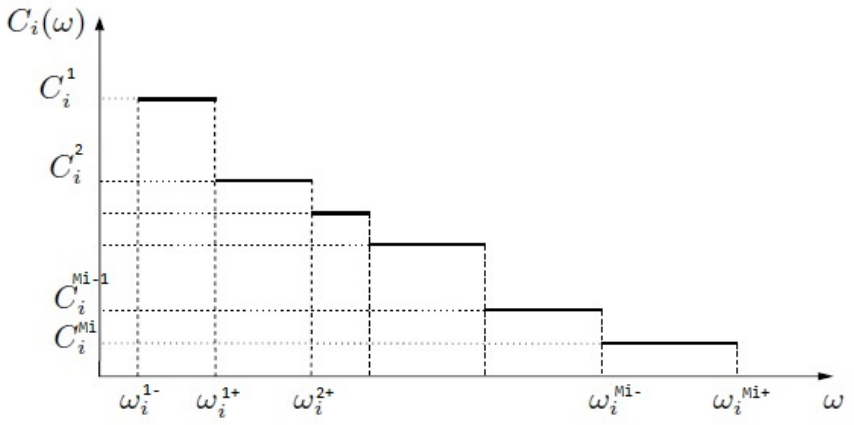

Figure 2.1: Computation time of an AVRTask as a function of the speed at the job activation

computation functions:

$$C_i^+(\omega) = C_i^m \quad \forall \omega \in (\omega_i^{(m-1)+}, \omega_i^{m+})$$

$$C_i^-(\omega) = C_i^m \quad \forall \omega \in (\omega_i^{m-}, \omega_i^{(m+1)-})$$

Similarly, the utilization of an AVR task with hysteresis (in steady-state conditions) can be expressed by the following two functions:

$$u_i^+(\omega) = \frac{C_i^+(\omega)}{T_i(\omega)} = \frac{\omega C_i^+(\omega)}{\Theta_i}$$

$$u_i^-(\omega) = \frac{C_i^-(\omega)}{T_i(\omega)} = \frac{\omega C_i^-(\omega)}{\Theta_i}$$

In the following, to simplify the notation of the analysis with no hysteresis, we use $C_i(\omega)$ and $u_i(\omega)$ to denote $C_i^+(\omega)$ and $u_i^+(\omega)$, respectively.

Let' s make an example in order to clarify the ideas:
Consider two AVR tasks τ_1^* and τ_2^*, both activated at the TDC ($\Phi 1 = \Phi_2 = 0$), with the same angular period $\Theta_1 = \Theta_2 = 2\pi$ and implicit angular deadlines ($\delta_i = 1$). Each

		C_i^m (ms)	ω_i^{m-} (rpm)	ω_i^{m+} (rpm)
τ_1	mode 1	2	500	2500
	mode 2	1	2500	6500
τ_2	mode 1	3	500	3500
	mode 2	0.5	3500	6500

Figure 2.2: Task example parameters

task implements two modes and is triggered by the same rotation source. The task parameters are reported in Fig. 2.2. Note that $\omega_1^{1-} = \omega_2^{1-}$ and $\omega_2^{2+} = \omega_2^{2+}$, since the same rotation source has clearly the same minimum and maximum speed. The implementation of such a type of tasks is typically performed as a sequence of conditional if statements, each executing a specific subset of functions. By defining an array of modes, however, an angular task can be efficiently implemented in this way:

```
//Global variables
mode[M] = {f1(), f2(), f3()};
w_plus[M] = {2000, 4000, 6000};
w_minus[M] = {500, 1500, 3500};
m = 1;
task sample_angular_task{
         w = read_rotation_speed();
         while (w > w_plus[m]) m = m+1;
         while (w < w_minus[m]) m = m-1;
}
execute(mode[m]);
```

In this implementation, function read_rotation_speed() returns the instantaneous speed ω at the task activation time (not at the execution time of the function).
Figure 2.3 graphically illustrates functions $C_i^-(\omega)$ and $C_i^+(\omega)$ for the AVR task shown in Fig. 2.2, while Figure 2.4 illustrates the steady-state utilization functions $u_i^-(\omega)$ and $u_i^+(\omega)$.

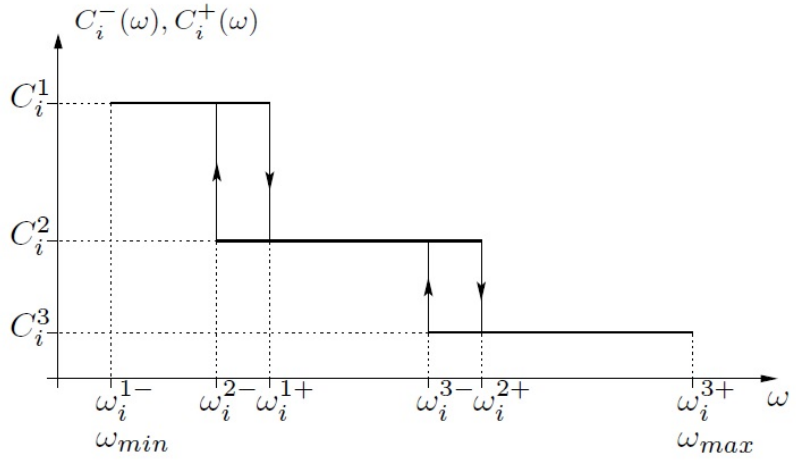

Figure 2.3: Task WCET as function of ω

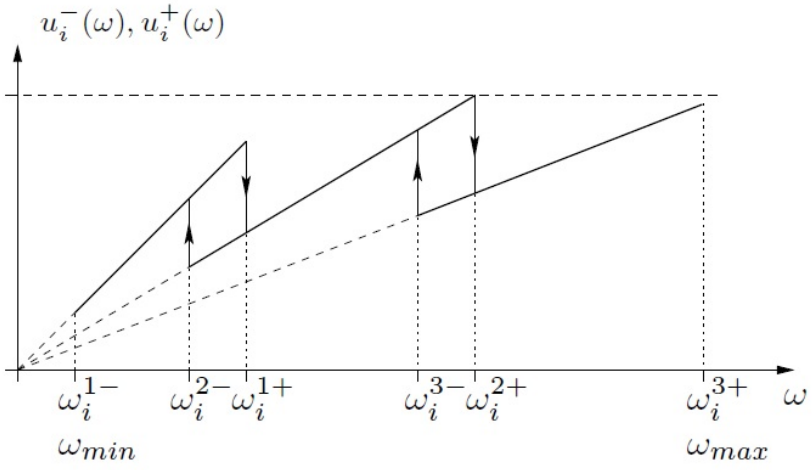

Figure 2.4: Task utilization in steady-state condition as function of ω

2.0.2 Rotation Source Model

For the purpose of analyzing the schedulability of engine control applications consisting of AVR tasks, it is crucial to characterize the relation between the task parameters and the dynamics of the engine.

Following the approach proposed by Buttazzo et al. [10], if (ω_k, α_k) is the state of the engine at time t_k, the next job release time of an AVR task can be computed (assuming

no other activations happen in this slice of time) as:

$$T_i(\omega_k, \alpha_k) = \frac{\sqrt{\omega_k^2 + 2\alpha_k \Theta_i} - \omega_k}{\alpha_k}$$

The corresponding relative deadline in the time domain can be computed by considering the value given by the max acceleration value, namely α_+

$$D_i(\omega_k, \alpha_k) = \frac{\sqrt{\omega_k^2 + 2\alpha_+ \Delta_i} - \omega_k}{\alpha_+}$$

In a similar way, the instantaneous rotation speed at the next job release can be computed (assuming there are no other activation in the meantime) as

$$\Omega_i(\omega_k, \alpha_k) = \sqrt{\omega_k^2 + 2\alpha_k \Theta_i}$$

If a job $J_{i,k}$ is released when the engine has an instantaneous speed ω_k, the interarrival time $\tilde{T}(\omega_k, \omega_{k+1})$ to the next job $J_{i,k+1}$ released with instantaneous speed ω_{k+1} (if reachable with acceleration bounds) can be like this:

$$\tilde{T}(\omega_k, \omega_{k+1}) = \frac{2\Theta_i}{\omega_k + \omega_{k+1}}$$

2.0.3 Issues of this new scenario

Dealing with variable rate tasks, it is not easy to deduce the behaviour of different schedulers, because the speed of the rotation source evolves in an unpredictable manner.
Lots of analisys studies have been performed in this case, searching for worst-case workload for AVRTask under EDF, or interference under FP.
Previous studies and scheduling theorems are no longer valid in such scenario: this is why is interesting to build up a simulator that manages AVRtasks, in order to get concrete values of scheduling parameters and performances and compare them with theoretical ongoing studies.

Chapter 3

Simulator Description

3.1 Overview

In order to build a basic scheduling simulator, the essentials software components are:

- Simulator Engine
- Kernel
- Scheduler
- Task

In order to handle AVRTasks, other three components are needed:

- AVRTask
- Rotation Source
- AVRTask Random Generator

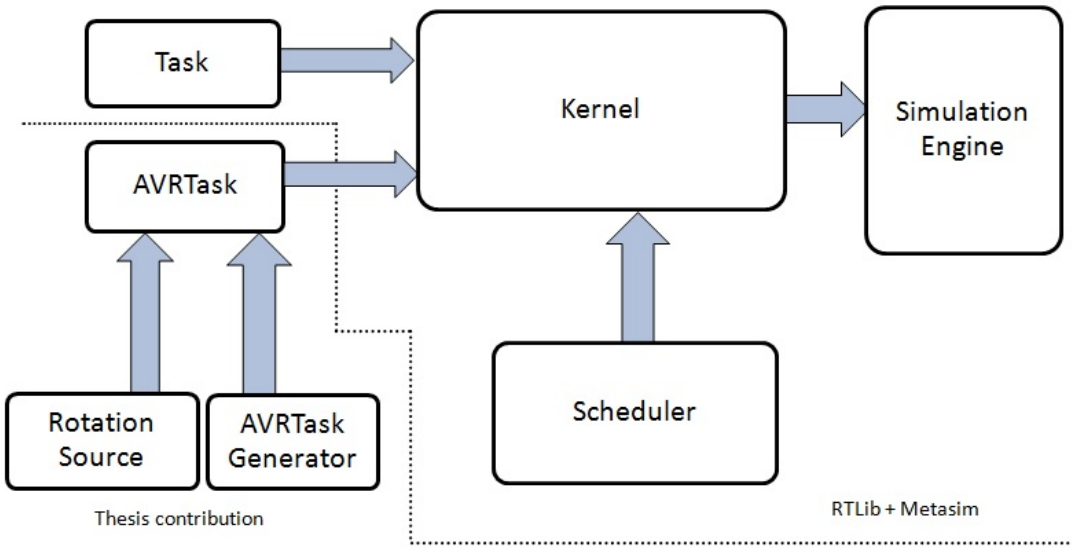

Figure 3.1: General Simulator Schema

Look at the schema in Fig. 3.1: Dealing with simulator engine, means that personalized events should be post (passed to it) with a specific time value; the kernel is the software module that has to do this work, with informations provided by the other components. Of course task parameters like instructions, period, deadline... are stored inside the corresponding software module and used by the kernel to post events like *Arrival*, *End*, *Miss*.

Schedulers' job is to maintain the tasks' queue ordered according to the related policy, updating the kernel's data structures.

Then AVRTask extends the task module in order to store the new parameters like angular period, angular deadline and mode index.

The random AVRTask generator is needed to generate random parameters for simulation purposes; finally the rotation source module implements the functions that describe the engine rotation and is responsible for AVRTask activations.

3.2 Available Libraries

Instead of program from scratch the basic blocks, two already available C++ libraries have been used:

RTLib2.0 Real-time kernel simulator based on Metasim.

Metasim2.0 Event-based discrete-time simulator.

Both of them has been developed inside Retis Laboratory as internal project in Scuola Superiore Sant' Anna. The author is Giuseppe Lipari, with the contrbution of many Phd students. This is all open source code, see here for more details:

- https://github.com/glipari/metasim2.0
- https://github.com/glipari/rtlib2.0

In this way, the work becomes to integrate the new modules to handle AVRTasks, with the already available basic ones that are provided by the libraries.

3.3 Metasim

Metasim is a very powerful framework that allows programmers to develop their own simulation entities that can post discrete events in time. In addition some useful classes are provided in order to handle random variables, collect statistical values , write traces and debug output. The user of a simulator wants to analyze the behavior of a system without actually having the system. So, he makes a model and runs it under different conditions and with different inputs, deterministic or randomly distributed. In a deterministic simulation the user is interested in analyzing the temporal evolution of the system state (trace) under certain conditions. If the input is randomly distributed, the user is interested in obtaining statistics on certain system variables, like average, variance, maximum and minimum value, confidence intervals, etc. The library has been built following the ideas of the Pattern Community. The manifesto of this community is the famous book "Design Patterns and Analysis" by E. Gamma et al. Here the list of the main classes of the framework:

- *Entities*: they are the bricks with which is possible to model a system. Every component of the system must be derived from this class. This class provides basic functionalities for initialization and provides a naming system.
- *Events*: our simulation is based on the discrete event model. Events are the basic objects for describing the temporal evolution of the system.
- *RandomVar*: the basic class to generate random variables from different distributions.
- *Simulation*: this is the main engine of the library.
- *BaseStat*: the basic class to collect statistics.
- *Trace*: the basic class to trace the behavior of a system.

The last two classes are exclusive: you may want to use only one of them, depending on the type of the simulation.

3.3.1 Entities

The base class for every simulation object. An entity has an internal status, an interface for modyfing the status, and can contain one or more events. An entity can be referred also by its name (a string of characters) using the static method find. A specific entity class should redefine the find function for doing type checking. A useful method for this class, especially when some simulation parameters have to be changed after some runs, is *newRun()*: it resets the entity status at the beginning of every run. This function is called automatically at the beginning of every run, and its job consists in initializing the entity status. It can be redefined in order to perform the desired changes. Warning: in *newRun()* is not permitted to create/destroy new entity objects.

3.3.2 Events

The basic event class. It models an event in the simulator: contains all the basic methods for handling an event. To define a new "type" of events in your system model, you need to derive a class from this, overriding the virtual *doit()* method. This class also inclides a static event queue, where all "active" events are enqueued. To insert an event in the queue, you can call the *post()* method specyfing a triggering time. Events are ordered in

the queue by triggering time. In case of two events with the same triggering time, events are ordered by priority. In fact, it is also possible to specify a priority for every event object. The priority is for the object, and not for the class! The lowest priority value is :

```
const int MetaSim::Event::_DEFAULT_PRIORITY = 8
```

while the highest one is

```
const int MetaSim::Event::_IMMEDIATE_PRIORITY = 0
```

It is not possible to post an event in the past, but is possible to post an event in the present. If the event is marked disposable, the main simulation loop will delete it after it has been processed. In practice, by setting disp to true, we are giving ownership of the object to the Simulation engine, which will destroy the object after using it.
When an event is "triggered" in the simulation, its *doit()* method is invoked. In most of the cases, the *doit()* method simply calls a method of an entity, which will be informally called "handler" of the event. In case the *doit()* method only calls the appropriate event handler, you can use the template class $GEvent< X >$ instead of deriving a new class.
An event can have a list of statistical objects. These are objects that are used to collect statistics on the system. See BaseStat for more details.
Finally, an event has a list of tracing objects. These are used to trace the evolution of the system for analysing it later. See Trace class for more details.

3.3.3 RandomVar

This class implements a random variable: some derived class provide more detailed implemetation depending on the type of random variable. The class diagram is self-explanatory.

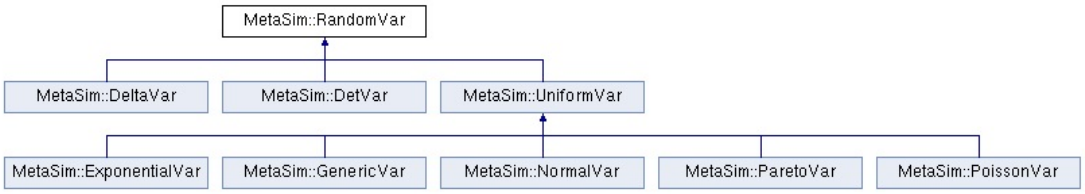

Figure 3.2: RandomVar class diagram

3.3.4 Simulation

This singleton implements the simulation engine and some debugging facilities. The main function is $run(Ticklenght, size\_t\ runs)$ that is responsible for running the simulation for one or more times. After defining all the objects in a simulation, this function should be invoked for running the simulation.This example code show how it should be used:

```
// create entities ane events, link them to each other
// create statistics and traces, attach them to the events
Simulation::run(10000, 10);   // run 10 simulations, each one for 10000 ticks
// collect and save statistics on files
```

At the beginning and at the end of each run, initialization and and finalization are called, respectively, namely the Entity::newRun() and Entity::endRun(). The random seed is not initialized at every run, but it is left as it is. Of course, it makes sense to run multiple replicas of the same simulation only if the simulation itself includes random variables, otherwise the result will always be the same. See the statistical section for more details. The *getTime()* returns the current *globalTime* in the simulation; the dbg object is used for debugging output.

3.3.5 BaseStat

This is the class that let the programmer to collect statistical values. Looking at Fig. 3.3 2 levels of abstraction are implemented, while the third is left to the programmer, depending on his needs. Level 0 (BaseStat) cannot be changed: it implements the functions for doing statistic: it is initialized with the number of experiments to be done, and has an array where the data are recorded at the end of the simulation. It has two pure virtual function, probe() and record(), so no object of this class can be instanciated. It has some function to get the final stats (like getMean and getConfInterval). Level 1 (StatMax ...) implements the function for collecting statistic; for example, the StatMax class record the maximum value during an experiment. The user can add his own classes to this level, and he must implement the record() function and the initValue() function. Level 2 implements the probe() for a single event. The user must write entirely this level, depending on the variable he needs to measure. When implementing a class of this level, the user must write the probe function, which has to call the record() function with the appropriate value.

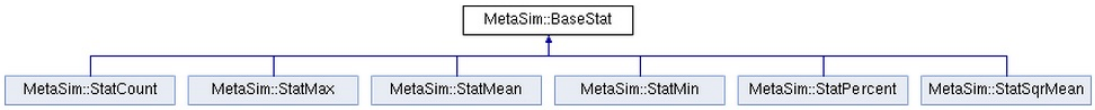

Figure 3.3: BaseStat class diagram

3.3.6 Trace

This class allows programmers to trace variables on a stream. By default it opens a binary stream. The derived class TraceASCII put the traced values into an ASCII stream.

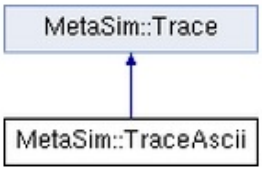

Figure 3.4: Trace class diagram

3.4 RTLib

RTLib provides software modules needed to simulate a real-time kernel, with all related stuff. It is based on Metasim, so Tasks, Schedulers, Kernels are classes derived from Entity, with all related events that are posted during the simulation. In order to get a concrete idea of how this library works, and how to use it to build new features, like AVRtaks, let's go inside implementation of some basic modules:

- *RTKernel*:The base module of real-time operating system.
- *Scheduler, TaskModel*: Modules that provide methods to handle task' s priority queues and scheduling parameters.
- *Task*: Base class that implements task functionalities.
- *Instr*: The base class for every pseudo-instruction.
- *TextTrace*: Module that let each event to be traced in a text file.

- *TaskStat*: Contains some classes in order to collect different statistical parameters related to tasks, i.e. : miss count, lateness, respons time...

There are other simulation entities provided by RTLib2.0; i am going to list them for the sake of completeness, but withouth enter in details, because they are not essential for my work.

- *CPU*
- *PollingServer*
- *SporadicServer*
- *Grub*
- *Supervisor*
- *Resource*
- *ResManager*
- *JavaTrace*

3.4.1 RTKernel

An implementation of a real-time single processor kernel. It contains :

- a ponter to one CPU
- a pointer to a Scheduler, which implements the scheduling policy;
- a pointer to a Resource Manager, which is responsable for resource access related operations and thus implements a resource allocation policy;
- the set of task handled by this kernel.

This implementation is quite general: it lets the user of this class the freedom to adopt any scheduler derived form Scheduler and a resorce manager derived from ResManager or no resorce manager at all.
Also the implementation for real-time multiprocessor kernel is available, as can be seen

from the class diagram in Fig. 3.5 . Method *dispatch()* compares currently executing task with the first in the ready queue. If they are different, it forces a context switch. The corresponding schedule and deschedule functions of the two tasks are called.
Function *onArrival()* is invoked from the task onArrival function, which in turn is invoked when a task arrival event is triggred. It inserts the task in the ready queue and calls *dispatch()*; An example pseudo-code to start a simple scheduling simulation can be like this:

```
CreateScheduler();
CreateKernel(&scheduler);
CreateTask;
kernel.addTask(&task,params);
```

Finally function *onEnd()* is invoked from the task onEnd function, which in turn is invoked when a task completes the execution of the current instance. It removes the task from the ready queue, set current executing pointer to NULL and calls *dispatch()*.

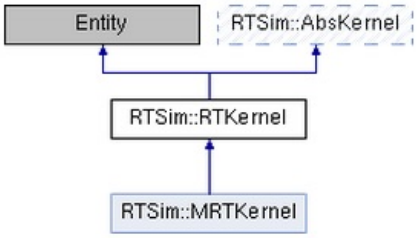

Figure 3.5: Kernel class diagram

3.4.2 Scheduler, TaskModel

Scheduler is an abstract class and cannot be instanciated.
This class models a generic real-time scheduler. It implements the Scheduler interface. Basically, this and the derived classes in Fig. 3.6 manage a priority queue in a convenient manner, and offer a clean interface toward the kernel and the resource manager.
The class keeps internally a repository of all tasks that can be scheduled by this scheduler. Every time a task is "added" to the scheduler, an appropriate *TaskModel* object is built, which contains the scheduling paramenters of the task. In this way, we clearly separate the task parameters (like period, deadline, wcet, etc.) that are contained in the task class, from the scheduling parameters that are contained in the *TaskModel* derived

classes. Implements the scheduling policy for a set of tasks. Tipically a scheduler contains a queue of task model' s instances. The responsibility of this class is to mantain the queue. Class *TaskModel* contains the scheduling parameters and a pointer to the

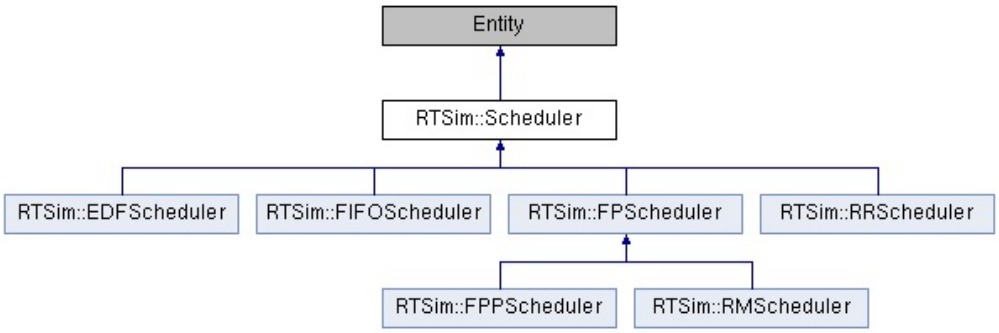

Figure 3.6: Scheduler class diagram

task. It is used by a scheduler to store the pointer to the task and the set of scheduling parameters. Each scheduler has its own task model. So the class inheritance trees of the task models and of the schedulers are similar (see Fig. 3.7).

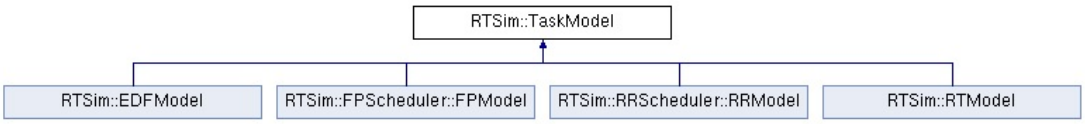

Figure 3.7: TaskModel class diagram

3.4.3 Task

This class models a cyclic task. A cyclic task is a task that is cyclically activated by a timer (for example a periodic task) or by an external event (sporadic or aperiodic task). This class models a "run-to-completion" semantic. Every activation (also called arrival), an instance of the task is executed. The task executes all the instructions in the sequence

until the last one, and then the instance is completed (task end).
At the next activation, the task starts executing a new instance, and the instruction pointer is reset to the beginning of the sequence.
When a job arrives (*onArrival()*), the corresponding deadline is set (the class Task has no deadline parameter). It adds deadline event to check deadline misses, with the possibility to abort the simulation in case of deadline miss (depending on the abort parameter in the constructor). In order to create a Task the programmer must provide two essential

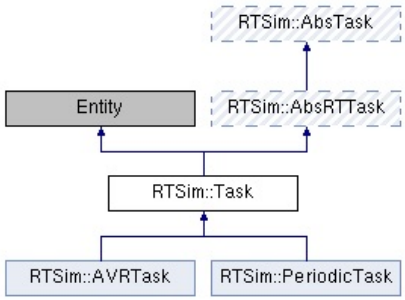

Figure 3.8: Task class diagram

parameters:

- *Interarrival-time* time between consecutive activations. Set it to NULL if you want just one activation.
- *Relative Deadline* Used to calculate the absolute deadline, when the task arrives.

Another important feature is to provide pseudo-code to the created task: function *insertCode(const string)*.
It parses and inserts instructions into this task. The input string must be a sequence of instructions separated by a semicolon (also for last instruction). The instruction' s types will be described in the related subsection.
Here is just an intuitive example of code:

```
t1.insertCode("fixed(4);wait(Res1);delay(unif(4,10));
                          signal(Res1); delay(unif(10,20));");
```

In this case, the task performs 5 instructions; the first one lasts 4 ticks; the second one is a wait on resource Res1; the third one has variable execution time, uniformely distributed between 4 and 10 ticks; the fourth one is a signal on resource Res1. Finally,

the last instruction has variable execution time uniformely distributed between 10 and 20 ticks.

3.4.4 Instruction

The base class for every pseudo instruction. Pseudo-instructions represents the code that a task executes.
An instruction is identified by an execution time (possibly random) and by a certain optional functionality. A task contains a list of instructions, that are executed in sequence.
From the Fig. 3.9 it is possible to deduce different types of pseudo-instructions implemented.

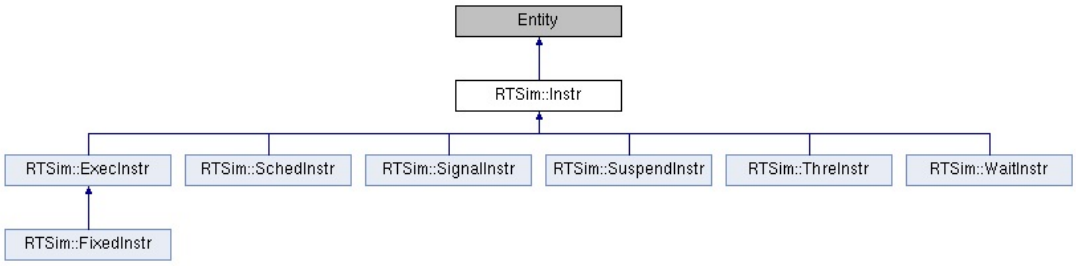

Figure 3.9: Instruction class diagram

3.4.5 TaskStat

Using statistical modules provided by Metasim i.e. StatMean, StatMax, StatMean it is possible to create custom statistics related to task. Here is the list of the task statistics implemented:

- *PreemptionStat*$< Measure >$ Computes the preemption count for each job
- *GlobalPreemptionStat* Computes the total number of preemptions.
- *FinishingTimeStat*$< Measure >$ Computes the finishing time of each job.
- *LatenessStat*$< Measure >$ Computes the lateness of each job of the task attached.
- *TardinessStat*$< Measure >$ Computes the finishing time normalized by the relative deadline.
- *UtilizationStat*$< Measure >$ Computes the utilization of each job.

- *MissPercentage* Computes the miss percentage.
- *MissCount* Computes the number of deadline misses . It can be applied to a single task or to the entire task set.
- $ConsumedPower< Measure >$
- $SavedPower< Measure >$

Some of them are created depending on the template class (Measure) that is passed to it, in order to allow programmers to collect the mean, max, min of the desired parameters within all the jobs of all the tasks that are attached to that class. Example code:

```
//average lateness of task(s) attached
LatenessStat<StatMean>
//max number of preemptions experienced by task(s) attached among all its(their) jobs
PreemptionCount<StatMax>
```

Chapter 4

Task model: AVRTask

4.1 Overview

In order to correctly implement the software module for Adaptive Varibale-Rate task it is needed to clearly understand the changes that it introduces with respect to the classical task.
A complete overview on the literature about this topic been already described in the introduction, following steps defined by research papers included in the bibliography. Now we have to define the new task model in terms of parameters, variable and implement the desired (already discussed) behaviour. Second step, but not less important, is to adapt these theoretical changes to the software modules that are already in RTLib2.0.

4.2 Theoretical model

AVRTasks run inside ECU and they are started at predetermined rotation angles. From this sentence I can immediately deduce the need of three parameters:

- *AngularPeriod*
- *AngularPhase*
- *AngularRelativeDeadline*

These parameters are fixed for each instance of taskAVR and don't change during the single simulation run.
This work allows the possibility to change them (between two simulation runs), withouth

the need to re-create the AVRTask with different parameters, but the reasons for doing this will be analyzed later.
It is easy to see that these three doubles need to be translated in the time domain, but this is left to another software module that will be analyzed in chapter 5 (Rotation Source).
For sure an integer variable is needed in order to distinguish between different execution modes depending on the current engine velocity (*mode*).
For the same reason *thresholds of angular velocities* are necessary for the correct mode transition. That is not enough, because if we want to consider hysteresis (that is different transition velocities in acceleration and deceleration), two arrays of angular velocities must be considered instead of one. The size of the velocity arrays and the maximum value (increased by 1) of the integer variable *mode* must be equal and correspond to the AVRTask 's number of modes. Last elements to be considered in order to have a general view are *Computation Times* values for each mode.
Summarizing in C++ code this first, incomplete version :

```
class AVRTask{

          double AngularPeriod;
          double AngularPhase;
          double AngularDl;
          int mode;
          vector<double> OmegaMinus;
          vector<double> OmegaPlus;
          vector<int> CompTime;
          vector<vector<Instr*>> myInstr;

}
```

4.3 Implementation

Looking at already available modules, the first one that can be used as base class from which derive AVRTask is of course *Task*. This is for sure a good choice, but some adjustments have to be made in order to get the correct behaviour.
First of all, AVRTask is not periodic in time, so the interarrival time parameter of task

class must be set to NULL.
If it is not periodic, each instance must be activated from external entities (Rotation Source), by calling the related function : *activate()*. In addition, at each activation, the mode index should be computed togheter with the temporal RelativeDeadline (both values depend on the current engine velocity). This is the final activate code:

```
void AVRTask::activate(int mode, Tick rdl) throw (ModeOutOfIndex){

            if (mode >= myInstr.size() || mode < 0)
                    throw ModeOutOfIndex("Mode Index out of range");
            BufferedDeadlines.push_back(rdl);
            BufferedModes.push_back(mode);
            arrEvt.drop();
            arrEvt.post(SIMUL.getTime());

}
```

As you can see, the mode index and the relative deadline are buffered into std::vectors: this is in order to handle overload conditions; let me describe how RTLib::Task behaves in such scenario. Suppose that a job of periodic task "Task1" is currently executing at simulation time 10 and the next job arrives exactly at time 10, while the previous one is not yet finished. RTLib manage this situation buffering the arrival time of the second job, that will be popped back when the first job finishes, posting *fakeArrivalEvent* that calls *handleArrival(PoppedArrivalTime)*.
So far, back to AVRTask, not only arrival time is need to be buffered, but also mode index and relative deadline that are essential job 's parameters. In order to pop back them in the correct order, the virtual *handleArrival()* should be redefined:

```
void AVRTask::handleArrival(Tick arr){

            setRelDline(*(BufferedDeadlines.begin()));
            BufferedDeadlines.erase(BufferedDeadlines.begin());

            mode = *(BufferedModes.begin());
            BufferedModes.erase(BufferedModes.begin());
```

```
            instrQueue.clear();

            vector<RTSim::Instr*> myCurrInstr = myInstr.at(mode);
            vector<RTSim::Instr*>::iterator j = myCurrInstr.begin();
            while (j!=myCurrInstr.end()){
                        addInstr(*j);
                        j++;
             }

             Task::handleArrival(arr);

}
```

Another problem that has to be solved is how to handle instructions.
Regular task requires the function *insertCode()* to build their instruction queue before the simulation starts: in fact Task::*newRun()* function throws an exception in the case of empty instruction queue.
AVRTask should have different instruction queues depending on the mode index; in addition it cannot build the instruction queue before the simulation starts, because it doesn' t know the execution mode until *activate()*;
This problem has been solved by creating all the possible instruction queues (one per mode) in the constructor function: *buildInstr()*, that receives as parameter a vector of string. Each string is parsed in order to create instruction instances from the instruction factory like the implementation of *insertCode()*. Then all of them are stored inside instance variable *myInstr*.
This is the constructor code, angular velocitiy values are passed in rpm and translated in rad/s;

```
AVRTask::AVRTask(double angPeriod, double angPhase, double angDl,
             vector<string> instr, vector<double> Omegaplus, vector<double>
             Omegaminus, const std::string &name) throw(WrongParameterSize)
             :Task(NULL, 0, 0, name, 1000), AngularPeriod(angPeriod),
             AngularPhase(angPhase), AngularDl(angDl){

             if (instr.size() != Omegaminus.size() ||
```

```
            instr.size() != Omegaplus.size())
                        throw WrongParameterSize();

        buildInstr(instr);

        OmegaPlus.assign(Omegaplus.begin(),Omegaplus.end());
        OmegaMinus.assign(Omegaminus.begin(), Omegaminus.end());

        std::transform(OmegaPlus.begin(), OmegaPlus.end(), OmegaPlus.begin(),
                        std::bind1st(std::multiplies<double>(), 2 * M_PI / 60));

        std::transform(OmegaMinus.begin(), OmegaMinus.end(), OmegaMinus.begin(),
                        std::bind1st(std::multiplies<double>(), 2 * M_PI / 60));

}
```

Then as can be seen above, in the *handleArrival()*, the correct instruction queue is selected, and used for that specific job.
After *handleArrival()* the flow of execution passes to the kernel entity, that insert the task in the ready queue and schedules it according to the policy established.
For simulation purpose, it is necessary to change AVRTask parameter, that are randomly generated, after some simulation runs; for this reason *changeStatus()* function has been implemented. It is very similar to the constructor, but allows to use the same entity for all the simulation runs.
Other functions that have been implemented are:

- *createInstance(string param).*
- *getWCET(int mode).*
- *∼AVRTask().*

The first one is needed by the task-factory, in order to create AVRTask instances.
An example code can be like this:

```
auto_ptr<Task> curr = genericFactory<Task>::instance().create(
            "AVRTask", vector<string>{
```

```
                        to_string(M_PI), to_string(0), to_string(M_PI_4),
                        string("fixed(20);"), string("fixed(12);"),
                        string("fixed(7);delay(1);"),
                        string("2000,4000,6000"),
                        string("500,1500,3500")
                }
);

AVRTask *t1 = (AVRTask*)curr.release();
```

The second, returns the worst-case execution time of all the instructions corresponding to the mode index passed as argument.
Last one, the destroyer, deallocates all the dynamic memory used for instructions creation and clears all vectors.

Chapter 5

Rotation Source Entity: AVRActivator

5.1 Overview

As described in the previous chapter, each job of each instance of AVRTask needs to be activated, with arguments like relative deadline and mode index.
In order to accomplish this task, and to simulate a new component, like an engine in the real world a new Metasim::Entity called *AVRActivator* has been implemented.
To be more precise, it is an entity that given acceleration and omega values, calculates the simulation time (number of Ticks) needed to rotate of an angular displacement and use this value to activate AVRTasks attached to it, correctly.

5.2 Implementation

Some implementation choices have been made, depending on the needs of usage of the implemented components. AVRTask code needs to be kept as simple as possible, in order to be reused, so some of the implementation complexity shifts to the *AVRActivator* component. In order to better understand these differences, let me describe the general algorithm performed by *AVRActivator*.
This is an iterative algorithm, that passes from an activation event to the next one, following the order imposed by the *AVRTasks*' parameters (*AngularPhase* and *AngularPeriod*) that are attached to that rotation source. So far, a personalized event: *_actEvt* is posted every time an activation should be performed.

The handler of the event (*OnActivate(Event* e))*includes all the computations of the rotation source model implementation.
This pseudo-code summarizes the flow of execution:

```
onActivate(Event* e){

        AccelerationUpdate();
        PostNextActivation(NextJob);
        Activate(CurrentJob);

}
```

Let' s analyze more in deep, the main functionalities of this software module:

- Task Management
- Physical Engine Management

5.2.1 Tasks management

First of all, it needs a list of elements like this one:

```
typedef struct elem{

        AVRTask* task;
        double ActivationAngle;

};
```

ActivationAngle values are clearly angular values that indicates the activation angle of the corresponding task.
The list:

```
list<elem*> _angularPositionToAVRTask;
```

must be ordered by increasing *ActivationAngle* values, because the *AVRActivator* work is to iteratively activate all the tasks ' jobs that are attached to it, only when the activation angle is reached.

List Management

List management consists in update the *ActivationAngle* value of the next task by adding its *AngularPeriod*; but to let the code work for even infinite time, I have to insert some kind of periodicity. The solution I have used is to define a threshold:

```
#define ACTIVATOR_PERIOD 2*M_PI
```

When the first *ActivationAngle* value in the list becomes greater than this threshold (like all the other values, because it is the lowest), the modulo with ACTIVATOR_PERIOD is calculated and used, for all the tasks.
This solution works because also the other variables: *LastActivation*, *ActivationTime* and *AngularVelocity* (see later for detailed description) are updated taking into account this particular case.

Next Activation Event creation

Togheter with list management also next *_actEvt* creation and post is performed by:

```
PostNextActivation(NextJob)
```

I have defined the *_actEvt* class with four variables:

- AVRTask* task;
- int mode;
- AVRActivator* myact;
- Tick RelDline;

The pointer to the task is need to call *AVRTask::activate(int mode, Tick RelDline)* inside the event handler.
The pointer to the activator is only for self-made event implementation issue.
The mode index and the relative deadline are computed and passed to the constructor of the *_actEvt*, that then is posted with the correct activation time.
Coming back to the implementation choiches i was talking about in the implementation section of this chapter, the mode index computation is implemented inside the *AVRActivator* instead of *AVRTask*, to let the last module easier to be used, also in other

simulation scenarios.
In order to do this, the task' s velocity threshold must be obtained by the *AVRActivator*:

```
int AVRActivator::CalculateMode(double Omega, AVRTask* task){

        vector<double> OmegaPlus = task->getOmegaPlus();
        vector<double> OmegaMinus = task->getOmegaMinus();
        int mode = 0;

        while (Omega > OmegaPlus.at(mode))
                mode++;

        while (Omega < OmegaMinus.at(mode))
                mode--;

        return mode;

}
```

Of course as all iterative methods, the initial case is implemented inside *newRun()* function, where the *postNextActivation()* is performed for the first job of the first *AVRTask*. In addition the initial values of the *ActivationAngles* is set to the *AngularPhase* of the corresponding task.

```
void AVRActivator::newRun(void){

        ElemIterator p = _angularPositionToAVRTask.begin();
        while (p != _angularPositionToAVRTask.end()){
                (*p)->ActivationAngle = (*p)->task->getAngularPhase();
                p++;
        }

        _angularPositionToAVRTask.sort(MyElemComparator());

        AlphaCurrent = UniformVar(AlphaMinus, AlphaPlus, &mygen).get();
```

```
        LastUpdate = 0;
        LastActivation = 0;

        //post the first task activation
        ElemIterator k = _angularPositionToAVRTask.begin();
        PostNextActivation(k);

}
```

The task list is created in the constructor and never changes, but task parameters can be changed by *AVRTask* interface, already described in the previous chapter.

5.2.2 Physical engine management

Physical laws

The model of a rotation source requires some physical laws to be implemented in order to update the activation times and angular velocity at each each step.
The functions that compute these values are:

```
getActivationTime(double omega, double alfa, double theta);
getAngularVelocity(double omega, double alfa, double theta);
```

The arguments required are the same for both of them:

- *double omega* the angular velocity at the current time instant t
- *double alfa* the angular acceleration that is assumed constant for the angular displacement *theta*.
- *double theta* the angular displacement the motor has to rotate, after which we want to know the time elapsed from t, or the new angular velocity.

Here the physical laws:

$$(1) ActivationTime = \frac{\sqrt{omega^2 + 2 * alpha * theta} - omega}{alpha}$$

$$(2) AngularVelocity = \sqrt{omega^2 + 2 * alpha * theta}$$

In the implementation I have made some adjustments:

- In (1) if *alpha* is zero the new formula becomes:

$$(3) ActivationTime = \frac{theta}{omega}$$

- In (2) saturation values for angular velocities are used:
  - `#define MAX_ANG_VELOCITY  6500*M_PI/30`
  - `#define MIN_ANG_VELOCITY  500*M_PI/30`

Class Variables

There are two class variables that needs to be kept between two consecutive activation:

- *LastActivation*
- *AngularVelocity*

The first one is need to define the angular displacement (*theta*) between two consecutive activations, that has to be passed to physical laws implementation:

$$theta = firstActivation - LastActivation$$

The second one also is one of the parameters to be passed to physical laws function. The *AngularVelocity* value can be changed at any time during the simulation, using the *SetOmegaZero(double newOmega)* function, where *newOmega* is in rpm. There are other class variables, related to acceleration update, that i will analyze in the last paragraph of this chapter.

PostNextActivation() Variables

Here the main variables used:

- *FirstActivation*: angular value of the next activation; the first of the ordered list.
- *NextAngularVelocity*: angular velocity at next activation instant, returned by *getAngularVelocity()*.
- *ActivationTime*: time elapsed from the current activation to the next one (in seconds), returned by *getActivationTime()*.

- *ActivationTimeTick*: time elapsed from the current activation to the next one (in Tick)
- *RelDeadline*: time elapsed from the next activation to the angular deadline expiration (in seconds), returned by *getActivationTime()*.
- *RelDeadlineTick*: time elapsed from the next activation to the angular deadline expiration (in Tick).

Of course class variables are substituted every activation with the corresponding function variable, to keep track of the status of the system, that continue evolving following the two already mentioned rules.
The only difficulty is just to set up parameters of physical function correctly and to handle periodicity.

From seconds to simulation ticks

As can be read from the previous paragraph, for each time variable, which has unit of measure in seconds, the corresponding one in simulation tick must be computed.
Physical laws return values in seconds, but this is a descrete-time, event-based simulator. A scale factor has been created, in order to set the desired granularity of the simulation time:

```
#define SIM_STEP_NS 1000000000
#define SIM_STEP_US 1000000
#define SIM_STEP_MS 1000

#ifndef SIM_STEP
#define SIM_STEP SIM_STEP_US
#endif
```

Depending on the *SIM_STEP* define statement, three values can be used: nanoseconds, microseconds, milliseconds.

Acceleration management

Looking at the activation event handler, the first function that is invoked is *UpdateAcceleration()*. Three modalities of acceleration updates are considered:

- Jerk values are taken into account: *JerkMinus* and *JerkPlus* are the boundaries. The time between two updates (*timeElapsed*) is needed to compute the delta of acceleration:

```
AlphaCurrent = UniformVar(
         AlphaCurrent + JerkMinus*timeElapsed,
         AlphaCurrent + JerkPlus*timeElapsed
)
```

  Then a saturation to *AlphaMinus* and *AlphaPlus* values is performed.

- Input values read from an external file.

- Random, uniformely distributed acceleration between *AlphaMinus AlphaPlus*.

An important issue in the first kind of update, is to set *JerkMinus*, *JerkPlus*, *AlphaMinus*, *AlphaPlus* in order to obtain velocity and acceleration realistic profiles. Now some example profiles will be showed, they have been elaborated during tuning phase: On the x-axis the simulation time in seconds (until 5s), on the y-axis rpm for velocity and rad/s^2 for acceleration.
The colored indexes in the legend chart identify different simulation runs.

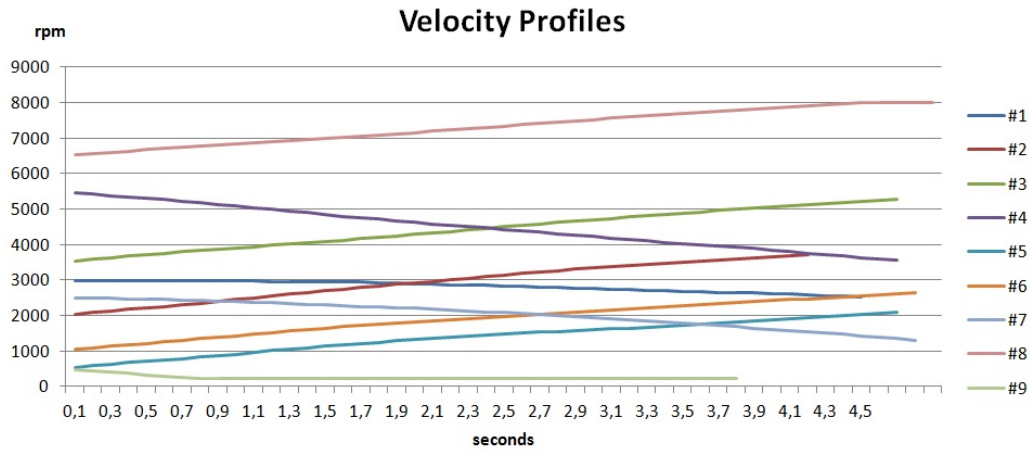

Figure 5.1: Velocity Profile with JerkMinus=-120rad/s^3, JerkPlus=120rad/s^3, AlphaMinus=-50rad/s^2, AlphaPlus=50rad/s^2

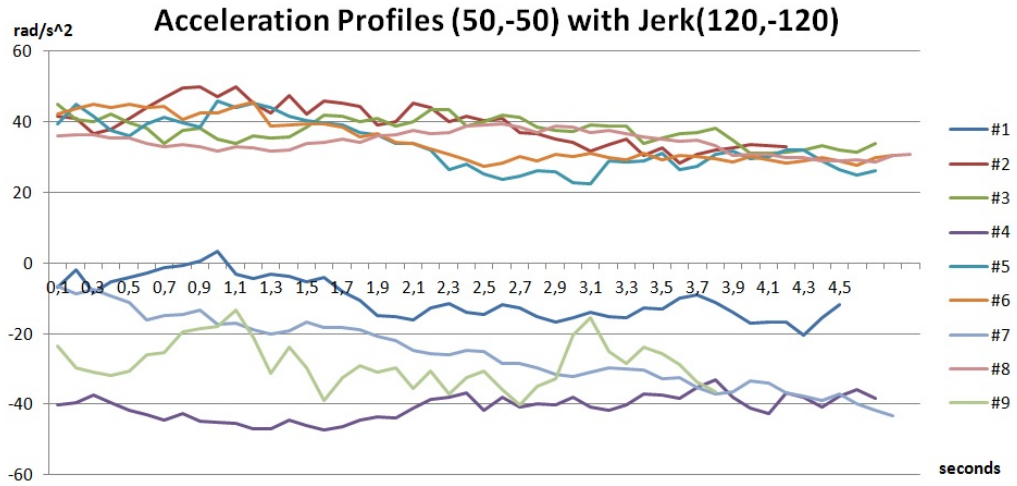

Figure 5.2: Velocity Profile with JerkMinus=-120rad/s^3, JerkPlus=120rad/s^3, AlphaMinus=-50rad/s^2, AlphaPlus=50rad/s^2

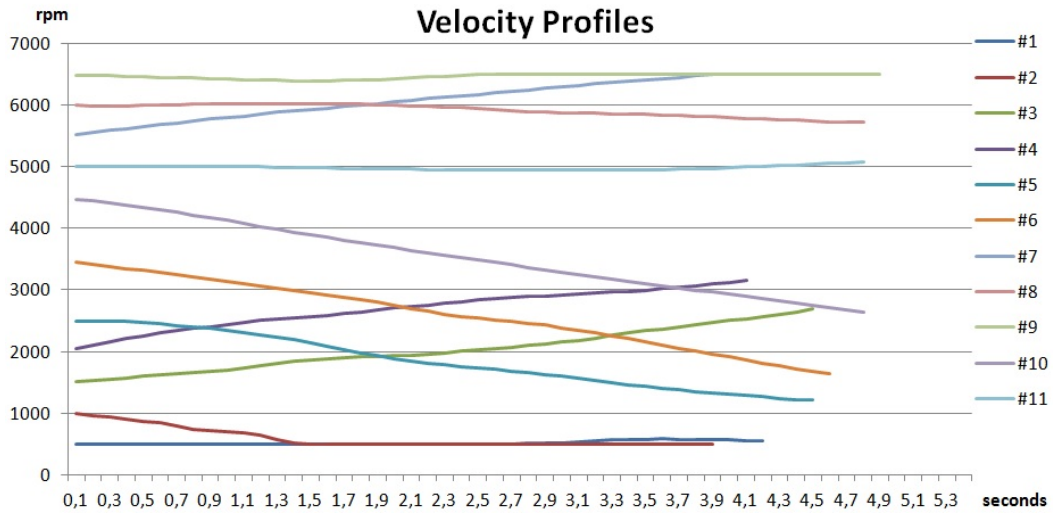

Figure 5.3: Velocity Profile with JerkMinus=-200rad/s^3, JerkPlus=200rad/s^3, AlphaMinus=-50rad/s^2, AlphaPlus=50rad/s^2

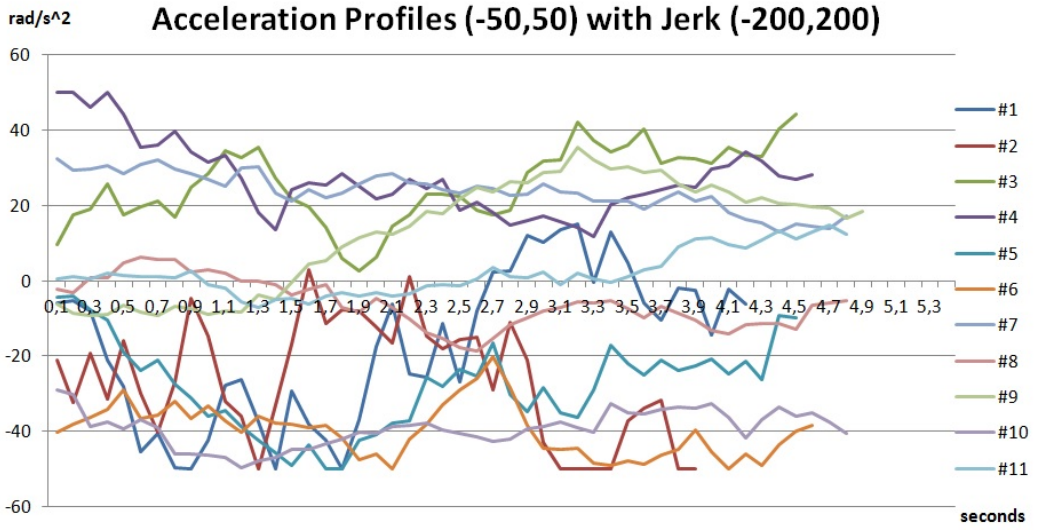

Figure 5.4: Velocity Profile with JerkMinus=-200rad/s^3, JerkPlus=200rad/s^3, AlphaMinus=-50rad/s^2, AlphaPlus=50rad/s^2

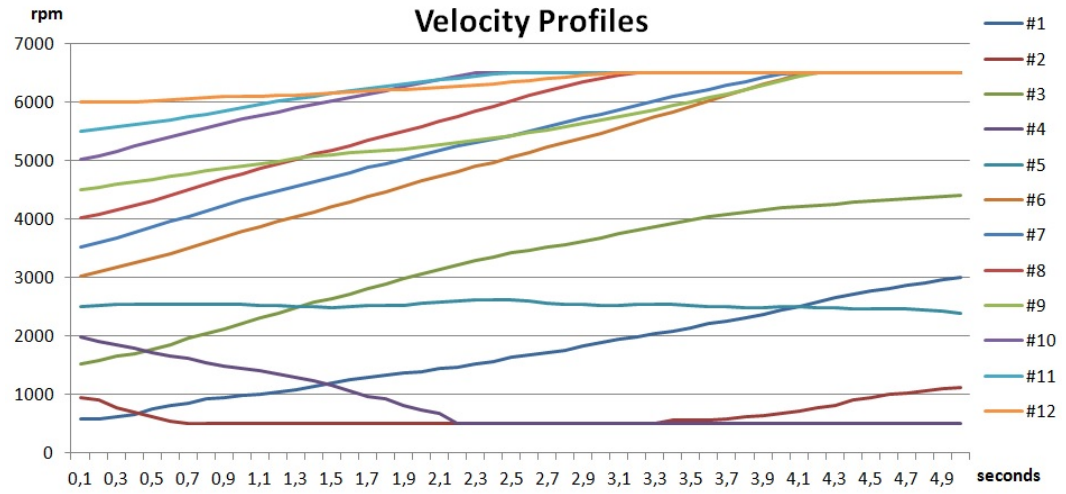

Figure 5.5: Velocity Profile with JerkMinus=-300rad/s^3, JerkPlus=300rad/s^3, AlphaMinus=-100rad/s^2, AlphaPlus=100rad/s^2

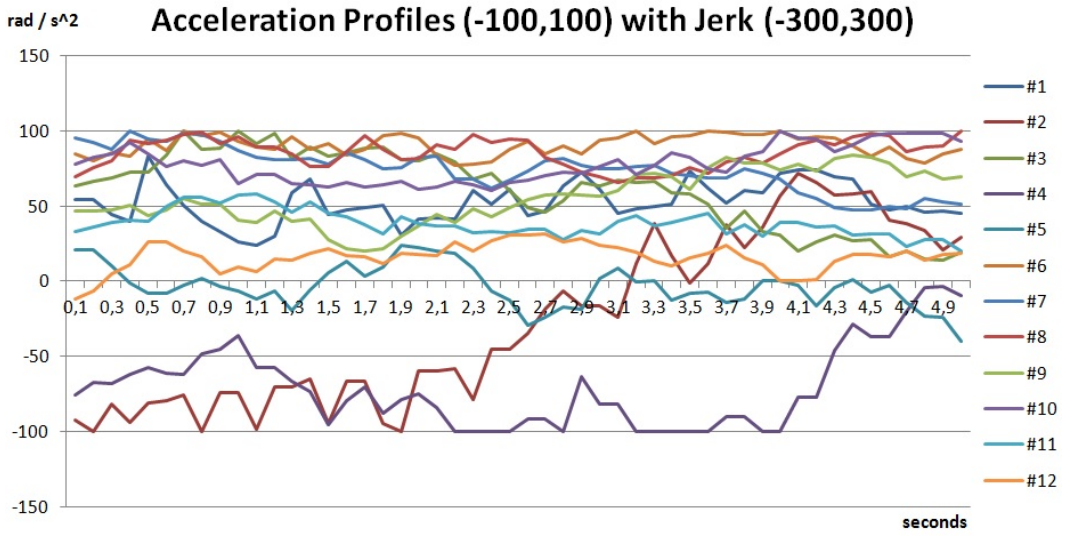

Figure 5.6: Velocity Profile with JerkMinus=-300rad/s^3, JerkPlus=300rad/s^3, AlphaMinus=-100rad/s^2, AlphaPlus=100rad/s^2

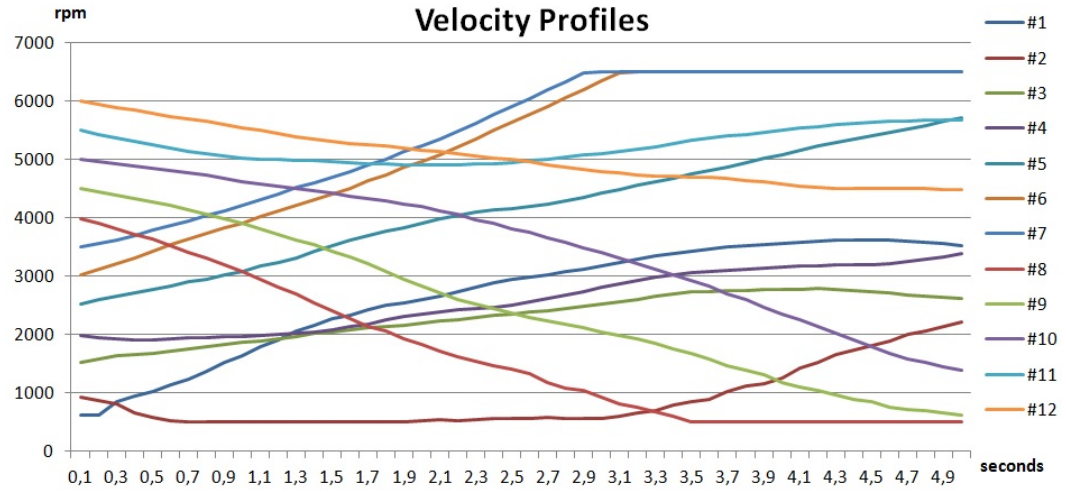

Figure 5.7: Velocity Profile with JerkMinus=-400rad/s^3, JerkPlus=400rad/s^3, AlphaMinus=-150rad/s^2, AlphaPlus=150rad/s^2

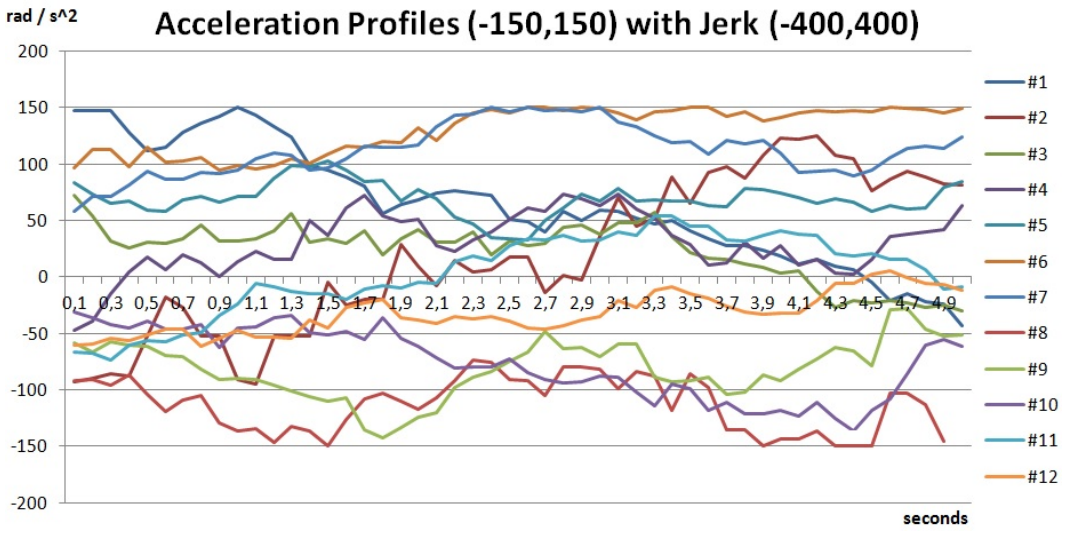

Figure 5.8: Velocity Profile with JerkMinus=-400rad/s^3, JerkPlus=400rad/s^3, AlphaMinus=-150rad/s^2, AlphaPlus=150rad/s^2

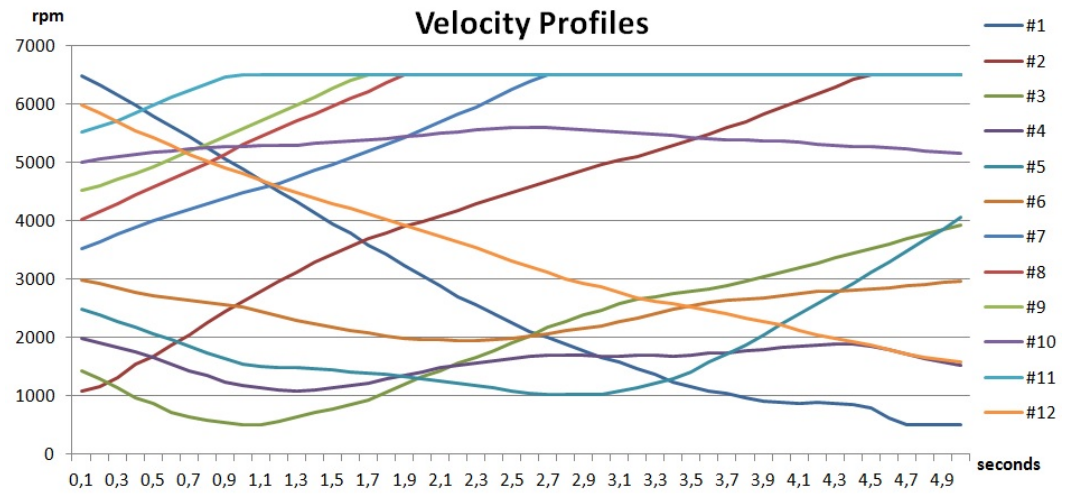

Figure 5.9: Velocity Profile with JerkMinus=-500rad/s^3, JerkPlus=500rad/s^3, AlphaMinus=-200rad/s^2, AlphaPlus=200rad/s^2

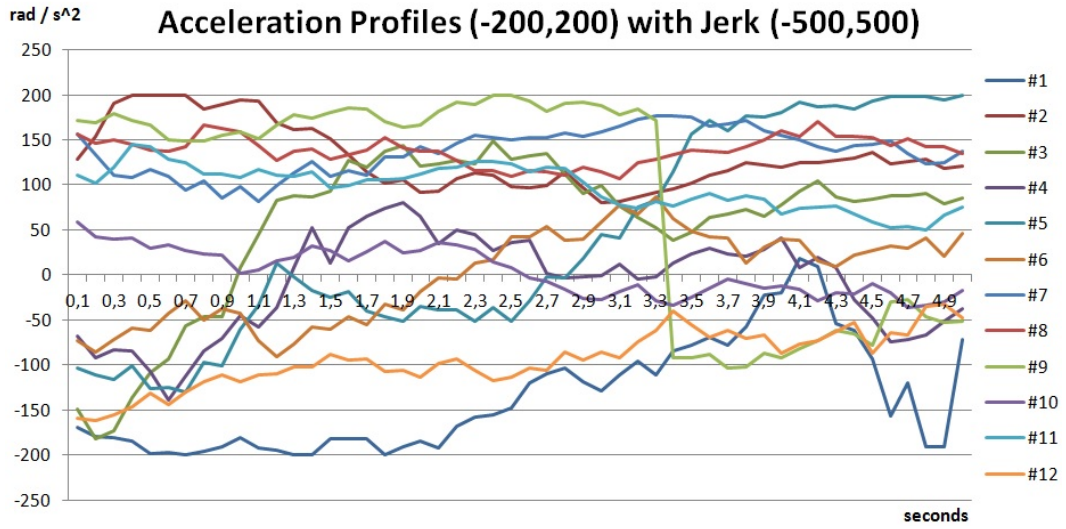

Figure 5.10: Velocity Profile with JerkMinus=-500rad/s^3, JerkPlus=500rad/s^3, AlphaMinus=-200rad/s^2, AlphaPlus=200rad/s^2

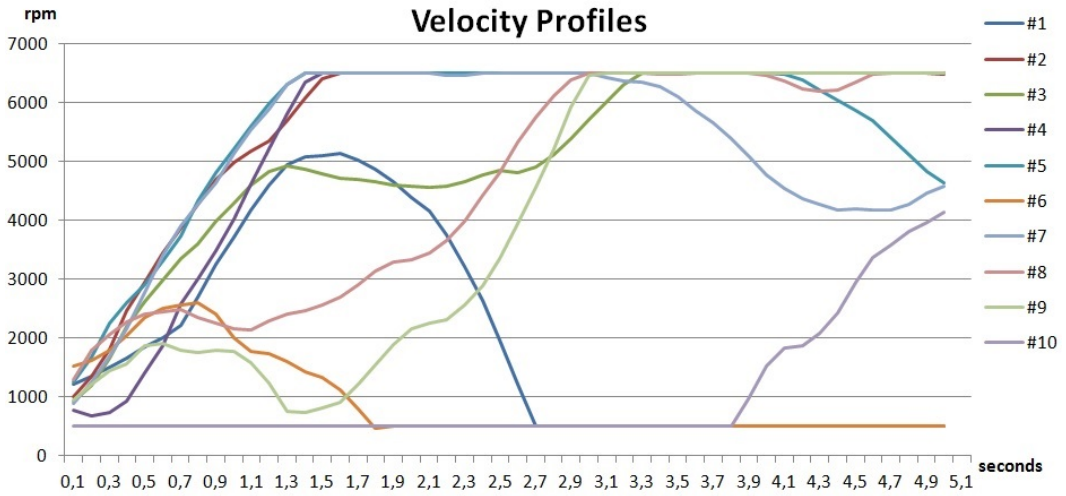

Figure 5.11: Velocity Profile with JerkMinus=-5000rad/s^3, JerkPlus=5000rad/s^3, AlphaMinus=-700rad/s^2, AlphaPlus=700rad/s^2

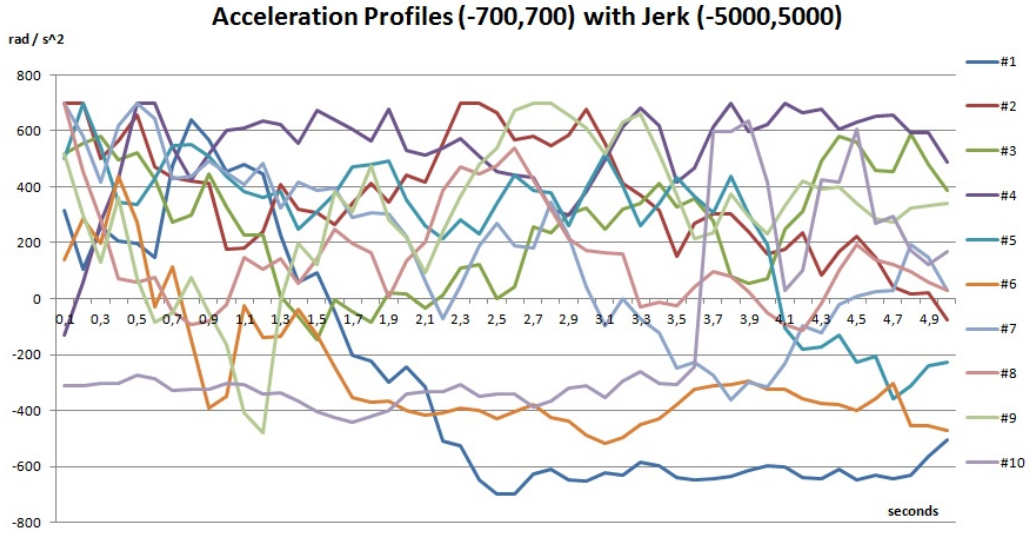

Figure 5.12: Velocity Profile with JerkMinus=-5000rad/s^3, JerkPlus=5000rad/s^3, AlphaMinus=-700rad/s^2, AlphaPlus=700rad/s^2

Chapter 6

Task Set Generator Entity: AVRGenerator

6.1 Overview

AVRTasks require some parameters in order to be instanciated; we can simply look at the class construnctor to list them:

- AngularPeriod.
- AngularPhase.
- AngularDeadline.
- Instruction' s lenghts and types.
- Angular velocities thresholds.

The goal is to randomly generate all these values by means of an entity, with some functions useful for my specific simulation purposes.

6.2 Implementation

The approach used for this entity implementation, is to maintain the last AVRTask parameters generated stored into class variables.
Also the list of all generated tasks is stored inside a class variable:

```
typedef vector<AVRTask*> AVRTaskList;
typedef vector<AVRTask*>::iterator AVRTaskIterator;

class AVRGenerator : public Entity {
        protected:
                    RandomGen mygen;
                    AVRTaskList myTasks;
                    double period, phase, rdl;
                    vector<Tick> MaxC;
                    vector<string> Instr;
                    int index;
                    vector<double> OmegaPlus; //rpm
                    vector<double> OmegaMinus; //rpm
        public:
                    ....
}
```

The random generator *mygen* is created in the class constructor and it is used for all random variables.
The general behaviour of the task generator consists in three main steps:

- Generate random AVRTask' s parameters from the generator' s parameters.
- Create AVRTask' s instances with last generated parameters.
- Update AVRTask 's instances parameters or create new instances with newly generated values.

Task list is needed to maintain a reference to all created AVRTasks; it can be returned by *getList()* function.
Here are the functions that has been implemented:

- *void updateTaskValues(int first, int max);*
- *AVRTaskList addTask(int n);*
- *void GenerateTaskParameter(AVRGeneratorParams myParams);*

- *AVRTaskList GenerateTaskSameParams(int n, int ncopies, AVRGeneratorParams myParams);*
- *AVRTaskList getList();*
- *void printTaskParam();*

Each of the created task' s instance parameter can be changed by using *updateTaskValues(int first, int max)*. This function will call *changeStatus()* of AVRTask class for each element of index between *first* and *max* inside the list; of course new values must be generated before function calling or it will be useless.
The two basic functions are *GenerateTaskParameter()* and *addTask()*: the first one generates random AVRTask parameters from generator parameters passed to it and stores them in the corresponding class variables. The second one simply creates n AVRTask' instances with stored parameters.
Depending on the simulation that has to be performed, previous functions allow the programmer to implement the desired behaviour.
For example, in this case, the decision is to generate couples of AVRTask' s instances with the same parameters (they will be used to compare two different scheduling algorithms performances); in addition the possibility to change their parameters after some runs has been taken into account.
For these reasons, *GenerateTaskSameParams* function, simply generates parameters and create *ncopies* of AVRTask instances: both this functionalities are repeated for n times.
Be careful because generating new tasks means deleting previous ones, while if I want only to change some task's values, withouth delete their instances, *updateTaskValues(int first, int max)* must be used.
Last one: *printTaskParam()* prints on screen the parameters for all tasks generated.

6.2.1 Random parameters generation

This process is implemented inside function *GenerateTaskParameter()*, starting from the modes number to the instructions generation. Here the structure that defines the generator parameters:

```
typedef struct AVRGeneratorParams{
                    int modesL;
                    int modesH;
                    double MinPeriod;
                    double MaxPeriod;
                    double MinPhase;
                    double MaxPhase;
                    double MinDl;
                    double MaxDl;
                    double minUPercentage;
                    double maxU;
                    int SpeedInterval;
                    int MinSpeedSeparation;
                    int MinSpeedPlus;
                    int MinSpeedMinus;
                    int MaxSpeedPlus;
                    int MinHyst;
                    int MaxHyst;
};
```

The first step is to define the number of modes: it is randomly generated between two integer values (*modesL*, *modesH*). Remember that AVRTask' s class deduces the maximum mode index by looking at vectors' sizes passed to the constructor (and checking that they have the same lenght).
Next parameter to be generated is Hysteresis, in the same way of the modes number, between *MinHyst* and *MaxHyst*.
The three main AVRtask' s parameters are: period, phase and relative deadline.
Period is generated as random variable uniformely distributed between *2*PI*MinPeriod* and *2*PI*MaxPeriod*.
Phase is generated as random variable uniformely distributed between *2*PI*MinPhase* and *2*PI*MaxPhase*.
Relative deadline is generated as random variable uniformely distributed between *MinDl*period* and *MaxDl*period*.
So far is possible to play with generator parameters, in order to get desired task parameters; for example if we want to set relative deadline equal to period, simply put:

```
MinDl = MaxDl = 1;
```

Antoher example: if you want to set phase equal to zero

```
MinPhase = MaxPhase = 0;
```

Otherwise just put correct fractions to the boundaries of the uniformely distributed random var.
Then the utilization factor distribution among modes is performed:

- One mode index is selected randomly and the *maxU* value is assigned to it.
- For all the other modes a random value between 0 and 1 (let' s call it *randZeroOne*) is generated and used to compute the utilization value for this mode in this way:

  ```
  double value = minUPercentage*maxU + (maxU - minUPercentage*maxU)*randZeroOne;
  ```

Angular velocity' s thresholds, in rpm, for mode changes are generated taking into account also hysteresis value: first thresholds' vector in the case of acceleration is computed, let' s call it *OmegaPlus*; then the other one in case of deceleration, namely *OmegaMinus*, is computed by subtracting hysteresis from *OmegaPlus*.
There are two fixed values, that are in the generator parameters set: the maximum speed threshold in acceleration (*MaxSpeedPlus*), and the minimum speed threshold in deceleration (*MinSpeedMinus*).
OmegaPlus elements are computed in this way:

```
int value = MinSpeedPlus + SpeedInterval * randZeroOne;
```

The vector is sorted in increasing order, then the computation is repeated until at least there is *MinSpeedSeparation* between two adjacent thresholds.
Both velocities and utilizations are needed to calculate the computation times for each mode, that are essential parameters in order to create instructions.
The instruction's queues are all composed by a single instruction of type "fixed", with computation time like this:

```
double value = period * utilization / angular_velocity;
```

Where *angular_velocity* is taken from *OmegaPlus* values, in the corresponding mode index, of course translated in rad/s.
Finally the check if computation times are decreasing with the increasing of the mode index is performed: if positive the parameters are stored and can be used to instanciate tasks, otherwise the generation process restarts until "good" values are reached.

Chapter 7

Experimental results

In order to perform lots of heavy computational simulations, that would take lot of time using a normal uniprocessor machine, a Unix-based multicore machine that is built inside Retis laboratory of Scuola Superiore Sant' Anna has been used.
The way to use it is a ssh-access to a virtual machine that utilizes 32 core of the physical machine.
Before showing the results, let' s describe how to setup simulations, in particular:

- the *main* programs that uses the software modules already presented.
- the scripts needed to distribute simulation runs among processors.

7.1 Main programs

In this section the usage of the software components already provided by RTLib2.0 and the three ones I have implemented, namely *AVRTask*, *AVRActivator*, *AVRGenerator* is shown.
Of course it is not possible to describe each main program for every simulation, so I will illustrate the general structure that, each time, can be adapted to the particular case.
Collecting statistical values requires lot of simulation to be run, in order to get meaningful results.
In this case each *simulation unit* consists of several simulation runs, each one with a different initial angular velocity for the *AVRActivator* module. In each *simulation unit* task' s parameters are left unchanged because the performances should be computed on the entire spectrum of angular velocities, in order to obtain significant values.

Launching one single *simulation unit* is not a clever idea, because we are dealing with statistical variables and the random generator will be useless in this case.
So far the code of the main program should execute a high number, let 's say at least 500, *simulation unit*, randomly generating new task' s parameters before each one.
Coming to the implementation this is the skeleton of the main program loop:

```
int main(){

        int NSims = NSIM;
        double Omega = OMEGAMIN;
        int count = 0;

        while (count < NSims){

                Omega = OMEGAMIN;

                myUpdater.SetRegenerate();

                while (Omega <= OMEGAMAX){

                        myUpdater.SetOmega(Omega);

                        SIMUL.run(SECONDS*SIM_STEP);

                        Omega = Omega + STEPOMEGA;

        }

        count++;
}
```

Here the define statments to adjust simulation parameters:

```
#define SECONDS 5
#define NSIM 1000
```

```
#define NPERTASK 4
#define NAVRTASK 1
#define ROU 0.4
#define STEPOMEGA 500
#define OMEGAMIN 500
#define OMEGAMAX 6500
#define MINPERIOD 20000
#define MAXPERIOD 100000
#define STEPPERIOD 1000
```

The main goal is to compare the two main scheduling algorithms: Earliest Deadline First and Fixed Priority when dealing with both periodic and AVR tasks.
In order to implement this behaviour in RTLib2.0, two *RTKernel* instances are needed, each one with his own *Scheduler* instance attached to it.
Of course corresponding tasks attached to the kernels should be identical, otherwise the comparison will be meaningless.
This means that, after generating parameters, two task' s instances must be created, each one attached to a different scheduler, of course for bot types: AVR and periodic.
In addition, also parameters change should be performed, between *simulation units*.
The choiche is to implement all this stuff inside a new Metasim::Entity calledl *Updater*.
The reasons of this decision are:

- use the *newRun()* method redefinition provided by Metasim in order to perform task parameters updates.
- leave the main program cycle clean and easy readable.

Looking at the main code, you can see the class *myUpdater* usage: it manages all the other simulation entities; more precisely:

- creates *AVRActivator* and gets the reference to the *AVRGenerator*.
- generates two periodic task-sets and other two AVRTask-sets with random parameters.
- attach tasks to kernels, links statistical variable to tasks and set fixed priorities.
- re-generates random parameters for tasks every *simulation unit*.

AVRTask generation is performed by *AVRGenerator* module and has been already discussed in chapter 4.
Periodic tasks generation is done by *Updater::GeneratePeriodicTasks()*; the parameters to be randomly computed are:

- periods.
- utilization factors.

From this two values, computation time also can be easily deduced.
This is the function used to generate periods:

```
Tick GetPeriod(RandomGen& mygen){
        int Kmax = (MAXPERIOD - MINPERIOD) / STEPPERIOD;
        UniformVar K(0, Kmax, &mygen);
        int kvalue = K.get();
        return MINPERIOD + kvalue*STEPPERIOD;
}
```

About utilization factors, the total utilization for both periodic and AVRTasks, let 's call it U, is passed as parameter to the main function; then two values are computed:

- $U\_AVR = ROU * U$
- $U\_PERIODIC = U$ - U_AVR

where ROU is one of the define statement, like the number of periodic taks ($NPERTASK$) and the number of AVRTasks ($NAVRTASK$).
The U_AVR and $U\_PERIODIC$ value are passed as input to the *UUnifast* algorithm implementation. This is the C++ implementation:

```
vector<double> UUnifast(int number, double MYU, RandomGen& mygen){
        vector<double> result;
        double sumU = MYU;
        double UMin = 0.001;
        double NextSumU, base, exp, temp;
        UniformVar myvar(0, 1, &mygen);
        for (int i = 0; i < number - 1; i++){
                base = myvar.get();
```

```
                exp = (double)((double)1 / (double)(number - i - 1));
                temp = pow(base, exp);
                NextSumU = sumU*temp;
                if (NextSumU>sumU - UMin)
                        NextSumU = sumU - UMin;
                if (NextSumU<(number - i - 1)*UMin)
                        NextSumU = (number - i - 1)*UMin;
                result.push_back(sumU - NextSumU);
                sumU = NextSumU;
        }
        result.push_back(sumU);
        return result;
}
```

In brief, *UUnifast* receives, for example, *U_PERIODIC* as *MYU*, the number of the periodic tasks *NPERTASK*, as *number* and randomly distribute the *U_PERIODIC* in *NPERTASK* values, with each value greater than *UMin*.
Of course the same procedure holds with *U_AVR* and *NAVRTASK*; then each single utilization value returned is set as *maxU* in the generator parameters.
While for AVRTasks, the random generation is totally performed by *AVRGenerator*, in the periodic case, the utilization factors for each task are returned and multiplied by the periods values, to get the computation time.
RTLib2.0 provides the *insertCode()* function to build the instructions, so we have just to create a string with the type of instruction and the simulation Ticks for the corresponding computation time.
An example code:

```
string k("fixed(" + to_string((int)C) + ");");
myTask->insertCode(k);
```

Of course when new parameters are generated, before inserting new instructions, previous ones must be deleted, like this way:

```
myTask->discardInstrs(true);
```

Also fixed priority management should be taken into account: in this case the priority value is an argument of *addTask()* function, from the *FPScheduler* interface.

The problem come when we want to change this value, because no *changePriority()* function is implemented; the trick used is:

- *myscheduler.discardTasks()*;
- *myscheduler.addTask(newPrio)*;

Let' s now resume the work performed by the *Updater* module:

```
class Updater : public Entity{

protected:
        vector<Task*> myEDFtasks;
        vector<Task*> myFPtasks;
        AVRGenerator* myGenerator;
        AVRActivator* myActivator;
        RTKernel* myEDFkern;
        RTKernel* myFPkern;
        RandomGen* mygen;
        double Omega;
        double U;
        double UStar;
        double UPer;
        FPScheduler* myfpsched;
        EDFScheduler* myedfsched;
        //statistical variables
public:

        Updater(...) : //initialization list
        {
                Omega = 500;
                GeneratePeriodicTasks();
                myGenerator->GenerateTaskSameParams(NAVRTASK, 2, *myParams);
                myActivator = new AVRActivator(AVRtasks, 50.0, -50.0, 200, -200);

        }
```

```
        void newRun(){

                myfpsched->discardTasks(true);
                myedfsched->discardTasks(true);
                Clist = UUnifast(NPERTASK, UPer, *mygen);
                for (int i = 0; i < NPERTASK; i++){
                        Tick myperiod = GetPeriod(*mygen);
                        //Update Task Period, Relative Deadline
                        //Instruction, Priority
                }

                myParams->maxU = UStar;
                for (int i = 0; i < NAVRTASK; i++){
                        myGenerator->GenerateTaskParameter(*myParams);
                        myGenerator->updateTaskValues(i * 2, (i + 1) * 2);
                }
                AVRTaskList myvector = myGenerator->getList();
                AVRTaskIterator p = myvector.begin();
                while (p != myvector.end()) {
                       //add AVRtask to scheduler
                       //also with priority parameter
                }
        }
        myActivator->SetOmegaZero(Omega);
}
```

The only remaining issue in the implementation of the main program, is the usage of statistical variables; of course it depends on the specific simulation goal, and I will analyze it for each experiment.
Let' s now understand how to set up the 32-core virtual machine in order to handle large scale simulations.

7.2 Scripts

In order to completely exploit the available cores of the Unix machine, some bash scripts are needed; in addition the structure of the main program should be created taking into account the goal to parallelize the executions.
Most of the executed experiments are repeated for different total utilization factor values: from 0.05 to 0.95 with step 0.05; now try to map one processor to one utilization factor. The idea is to parametrize the main execution with the utilization factor value; but this will not let to the total processors utilization: in fact only 19 processors over 32 will be used in that way.
It is possible to divide the single main execution in two equal parts: each one with half simulation units number; this process is performed until there is some free processor.
In order to do this another parameter is needed to be passed to the main program: an index (1 or 2), that says the program to divide the total number of simulation units by two.
At the end only 6 main executions will be left undivided (no index is passed to main in that case).
Last argument of the main program is the name of the experiment, in order to save them in an appropriate directory.
This script automatize the above behaviour by using variables that maintain the number of available processors (*Ncore*) and the number of experiments to be performed (*Nexp*).
The executable name should be passed to it like that:

```
./launch ExpName AVRSimulator
```

This is the launch.bash code:

```
#!/bin/bash
args=("$@")
LD_LIBRARY_PATH=./lib
export LD_LIBRARY_PATH

Uno=1
Due=2
Ncore=32
Umax=0.95
```

```
Umin=0.05
Ustep=0.05
Nexp=$(echo "( $Umax-$Umin ) / $Ustep + 1" | bc)
Expname=${args[0]}
mkdir -p ./results/$Expname/1
mkdir -p ./results/$Expname/2
for X in $(seq $Umin $Ustep $Umax)
do
        if [ $(echo $Nexp'<'$Ncore | bc -l) -eq 1 ]
        then
                nohup ./$2 $X $Expname $Uno &
                nohup ./$2 $X $Expname $Due &
                Ncore=$(($Ncore-2))
        else
                nohup ./$2 $X $Expname &
                Ncore=$(($Ncore-1))
        fi

        Nexp=$(($Nexp-1))
done
```

Nohup is used to let the processes continue after ssh exit.
Of course also collecting the final results should take into account the experiment division: the medium value is calculated and the final "result.txt" file is updated with all the values for each utilization factor.
This is also implemented in a script, namely "collectAVR.bash".
This is how final result.txt generally looks like:

```
0.95:value1_EDF:value1_FP:value2_EDF:value2_FP:...:valueN_EDF:valueN_FP
0.9 :value1_EDF:value1_FP:value2_EDF:value2_FP:...:valueN_EDF:valueN_FP
0.85:value1_EDF:value1_FP:value2_EDF:value2_FP:...:valueN_EDF:valueN_FP
0.8 :value1_EDF:value1_FP:value2_EDF:value2_FP:...:valueN_EDF:valueN_FP
0.75:value1_EDF:value1_FP:value2_EDF:value2_FP:...:valueN_EDF:valueN_FP
0.7 :value1_EDF:value1_FP:value2_EDF:value2_FP:...:valueN_EDF:valueN_FP
0.65:value1_EDF:value1_FP:value2_EDF:value2_FP:...:valueN_EDF:valueN_FP
```

```
0.6 :value1_EDF:value1_FP:value2_EDF:value2_FP:...:valueN_EDF:valueN_FP
0.55:value1_EDF:value1_FP:value2_EDF:value2_FP:...:valueN_EDF:valueN_FP
0.5 :value1_EDF:value1_FP:value2_EDF:value2_FP:...:valueN_EDF:valueN_FP
0.05:value1_EDF:value1_FP:value2_EDF:value2_FP:...:valueN_EDF:valueN_FP
0.15:value1_EDF:value1_FP:value2_EDF:value2_FP:...:valueN_EDF:valueN_FP
0.1 :value1_EDF:value1_FP:value2_EDF:value2_FP:...:valueN_EDF:valueN_FP
0.2 :value1_EDF:value1_FP:value2_EDF:value2_FP:...:valueN_EDF:valueN_FP
0.25:value1_EDF:value1_FP:value2_EDF:value2_FP:...:valueN_EDF:valueN_FP
0.3 :value1_EDF:value1_FP:value2_EDF:value2_FP:...:valueN_EDF:valueN_FP
0.35:value1_EDF:value1_FP:value2_EDF:value2_FP:...:valueN_EDF:valueN_FP
0.45:value1_EDF:value1_FP:value2_EDF:value2_FP:...:valueN_EDF:valueN_FP
0.4 :value1_EDF:value1_FP:value2_EDF:value2_FP:...:valueN_EDF:valueN_FP
```

Now let 's describe the experiments that have been performed.

7.3 Experiments

7.3.1 Schedulability

Schedulability ratio is one of the basic parameters for real-time systems, like engine control systems.
So when comparing two scheduling algorithms in such scenario, we need to check the schedulability ratio in both cases.
This means to evaluate the performance of each algorithm, in terms of number of schedulable task set (with all jobs that ends before the corresponing deadline), divided by the total number of generated task set.
RTLib2.0 provides a software module for deadline miss count: *MissCount*.
Two *MissCount* variables have been used(one for EDF and the other for FP): if the total number of deadline misses (*resEDF* or *resFP*) in the same *simulation unit* is greater than 0, the corresponding task set is not schedulable.
At the end of the entire simulation the values:

```
1 - resEDF / NSims;
1 - resFP / NSims;
```

are the schedulability ratios for each algorithm.
Dealing with AVRTasks, it is obvious to run the same taks set with different initial angular velocities (from 500rpm to 6500rpm, with step 500rpm) for the engine simulation entity: this is what I called *simulation unit*.
After each *simulation unit* ends, new task' s parameters are generated and udpated.
It is useful to establish a set of experiments, each one with some fixed parameters and focused on a specific target, in order to better exploit the huge number of variables involved.

Experiment 1: Best priority assignment

About the simulation parameters, two copies of the same task set have been considered, they are composed by:

- 4 periodic tasks
- 1 AVRTask

Periods for periodic tasks are in the range (20,50)ms with step 1ms.
Angular period is 2Π, angular phases are all 0, angular deadlines are all equal to periods.
Computation time generation has already been explained in the *Main program* section of this chapter, and ROU value for utilization factors computation is set to 0.4.
Fixed priorities are set in this way:

- for periodic tasks they are equal to periods.
- for AVRTasks three levels are considered:
  - *High (20ms).*
  - *Medium (35ms).*
  - *Low (50ms).*

A total number of 1000 *simulation units* runs for every simulation.
The maximum acceleration that can be reached is 50 rad/s^2, maximum Jerk is 200rad/s^3.

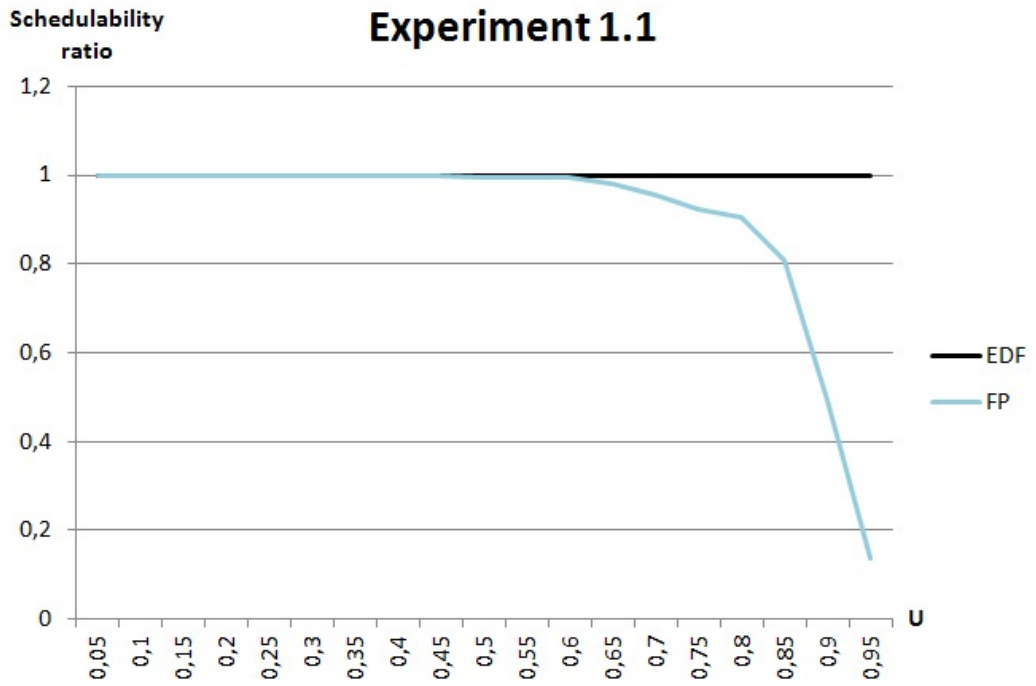

Figure 7.1: Schedulability ratio EDFvsFP, 4 periodic tasks, one AVR with highest priority

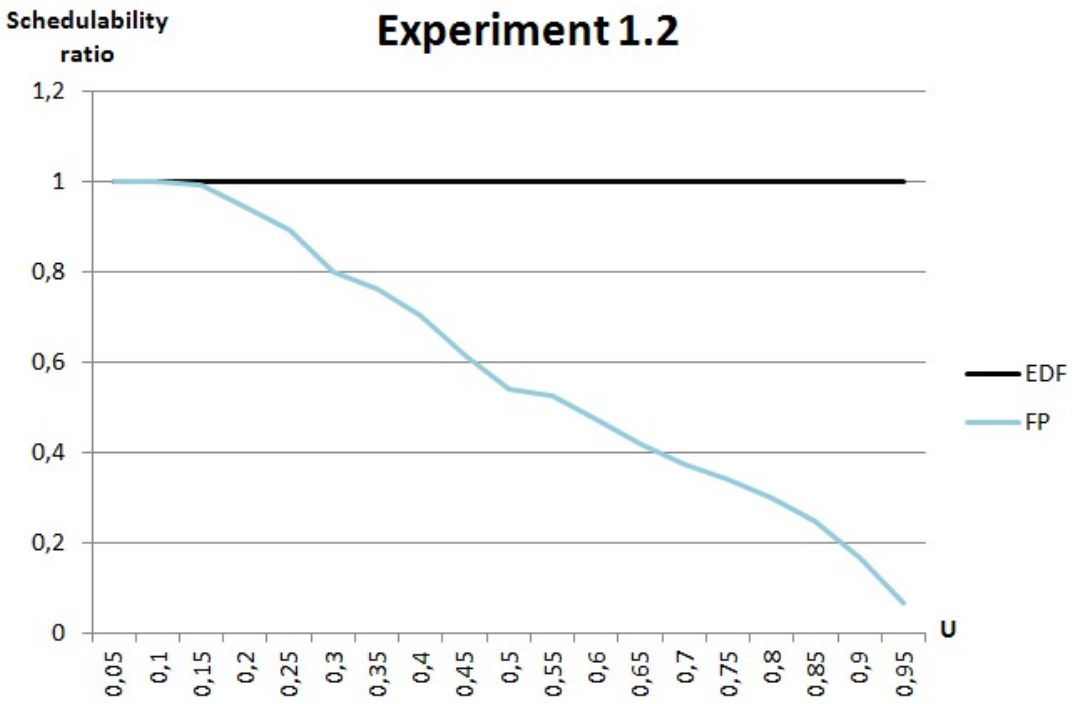

Figure 7.2: Schedulability ratio EDFvsFP, 4 periodic tasks, one AVR with medium priority

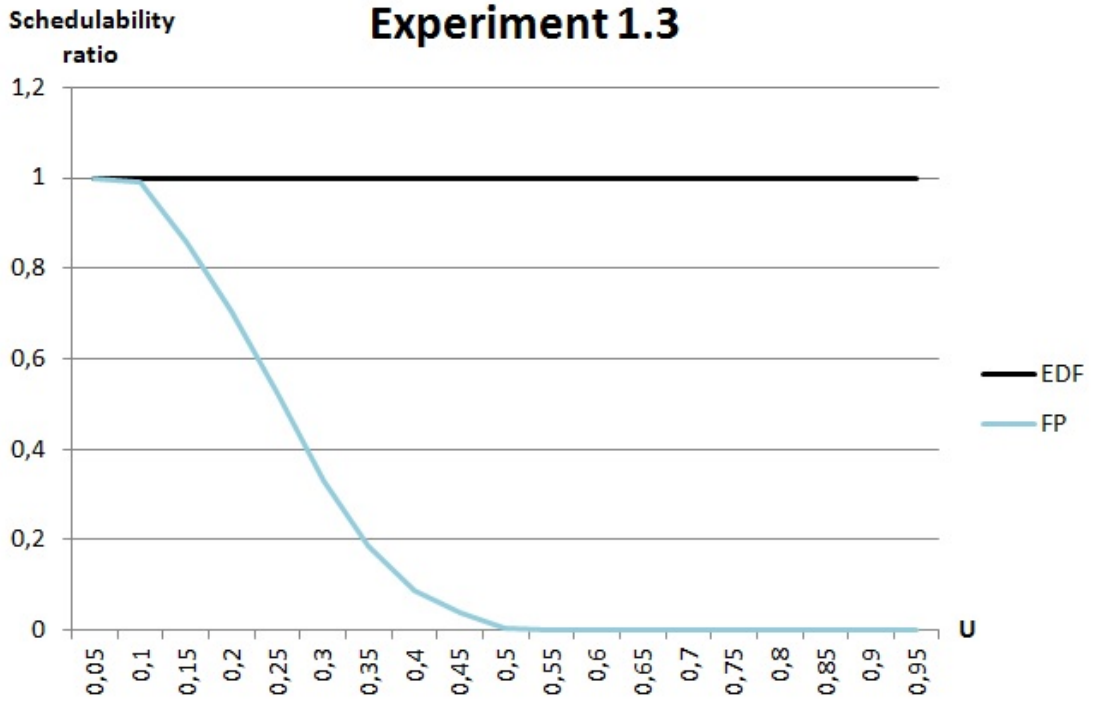

Figure 7.3: Schedulability ratio EDFvsFP, 4 periodic tasks, one AVR with lowest priority

As expected from the scheduling-theory, Earliest Deadline First outperforms Fixed Priority in therms of schedulability ratio: it schedules all task-sets withouth deadline misses for all the simulations.
However the main goal we want to achieve launching this sequence of experiments is to determine a clever priority assignment for AVRTasks, that let FP algorithm perform better, from the schedulability point of view (among the possible choiches). As evident from the graphs, in such scenario (with only three priority levels), the best priority assignment for AVRTasks is the highest one.
Of course there is no optimal priority assignment, because they are variable rate, so there will be some angular velocities for which the chosen priority is not the best one.
Looking at Fig.7.3, the AVRTaks always miss their deadlines, having the lowest priority value.

Experiment 2: Acceleration and Jerk dependency

In this experiment the goal is to understand how much maximum acceleration and jerk values affect the schedulability ratio.
About the simulation parameters, in this case I have considered two copies of the same task set, composed by:

- 7 periodic tasks
- 1 AVRTask

Periods for periodic tasks are in the range (3,100)ms with step 1ms.
Angular periods are fixed at 2Π, angular phases are all 0, angular deadlines are all equal to periods.
Running the same simulation at several speed let the period of AVRTask vary in the range (9,120)ms, depending on the current engine speed.
Computation time generation has already been explained in the *Main program* section of this chapter.
ROU value for utilization factors computation is set to 0.4.
Fixed priorities are set in this way:

- for periodic tasks they are equal to periods.
- for AVRTasks is set to about 9ms.

A total number of 1000 *simulation units* runs for every simulation.
The maximum acceleration that can be reached varies from 50 rad/s^2 to 700 rad/s^2, and maximum Jerk varies from 200rad/s^3 to 5000rad/s^3.

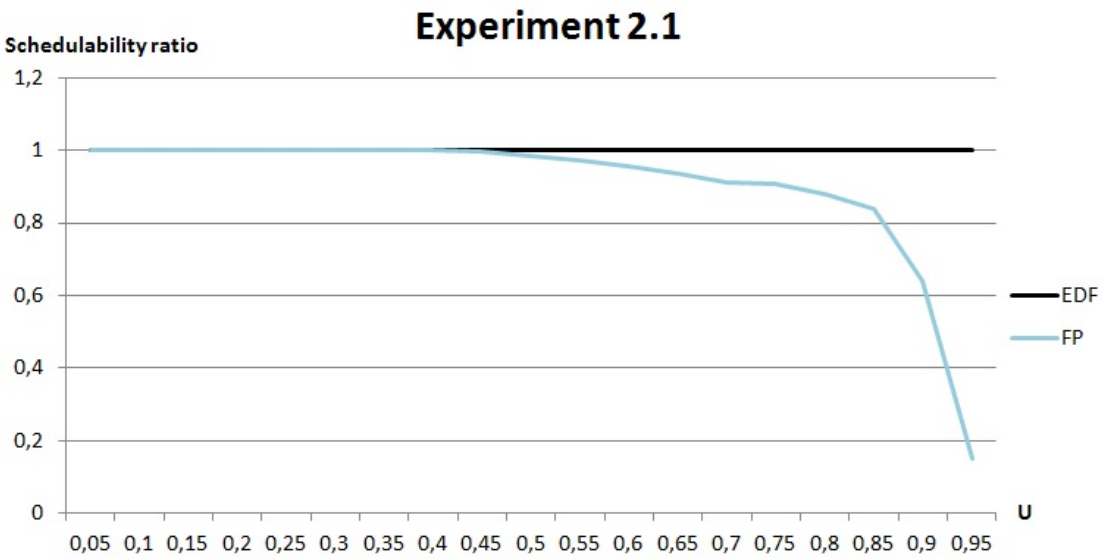

Figure 7.4: Schedulability ratio EDFvsFP, 7 periodic tasks, one AVR with JerkMinus=-120rad/s^3, JerkPlus=120rad/s^3, AlphaMinus=-50rad/s^2, AlphaPlus=50rad/s^2

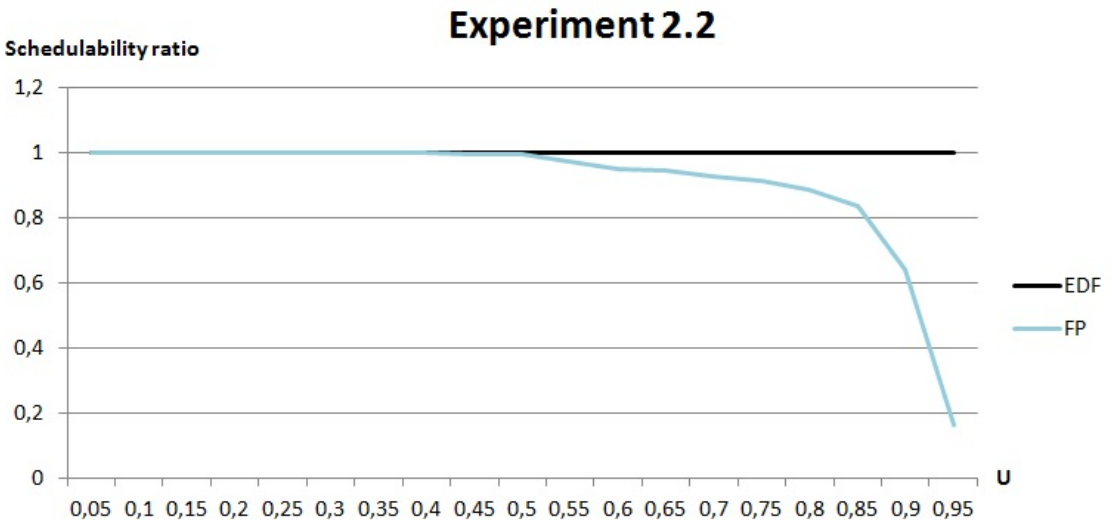

Figure 7.5: Schedulability ratio EDFvsFP, 7 periodic tasks, one AVR with JerkMinus=-300rad/s^3, JerkPlus=300rad/s^3, AlphaMinus=-300rad/s^2, AlphaPlus=100rad/s^2

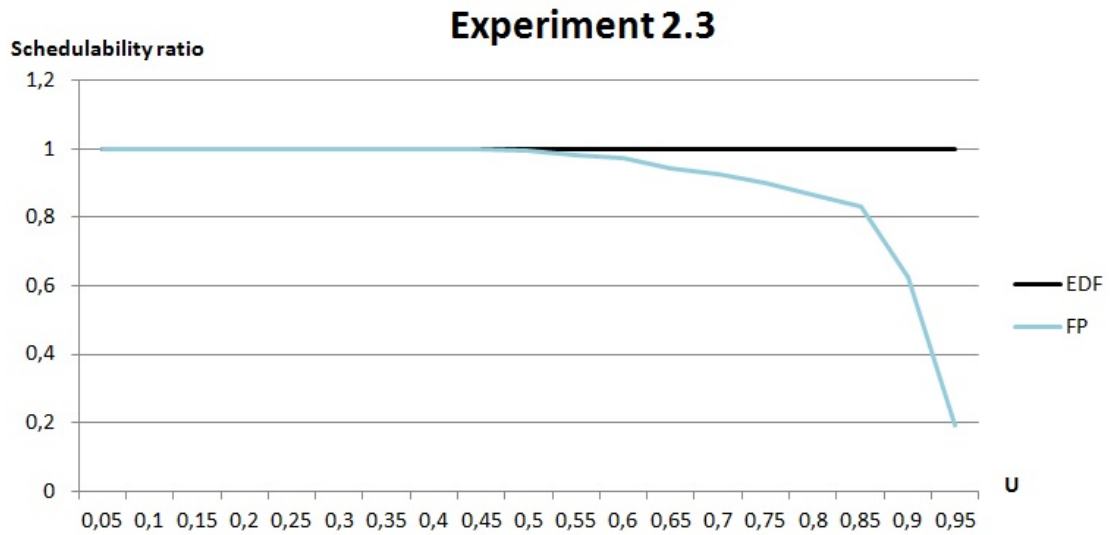

Figure 7.6: Schedulability ratio EDFvsFP, 7 periodic tasks, one AVR with JerkMinus=-400rad/s^3, JerkPlus=400rad/s^3, AlphaMinus=-150rad/s^2, AlphaPlus=150rad/s^2

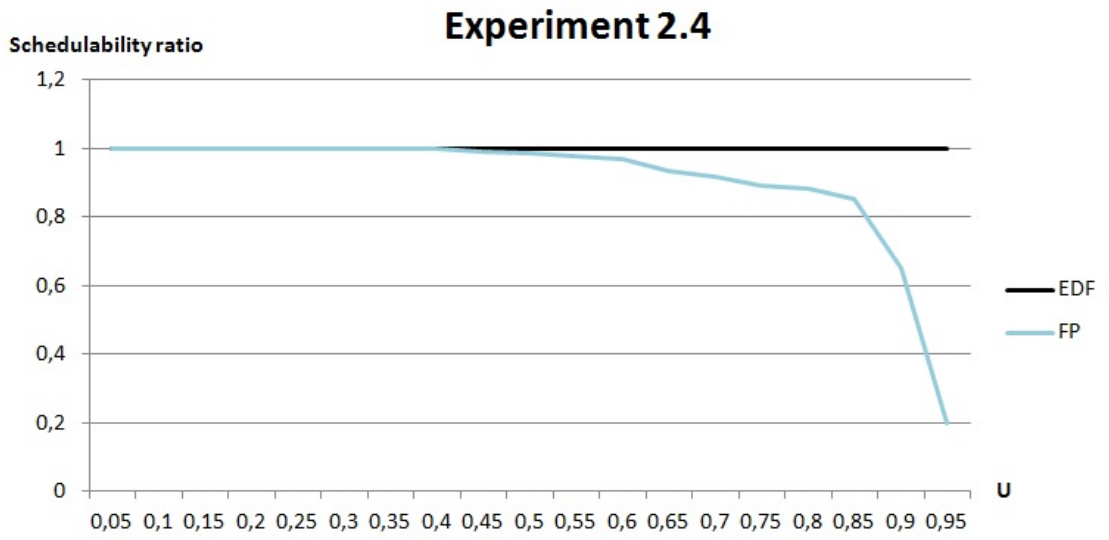

Figure 7.7: Schedulability ratio EDFvsFP, 7 periodic tasks, one AVR with JerkMinus=-500rad/s^3, JerkPlus=500rad/s^3, AlphaMinus=-200rad/s^2, AlphaPlus=200rad/s^2

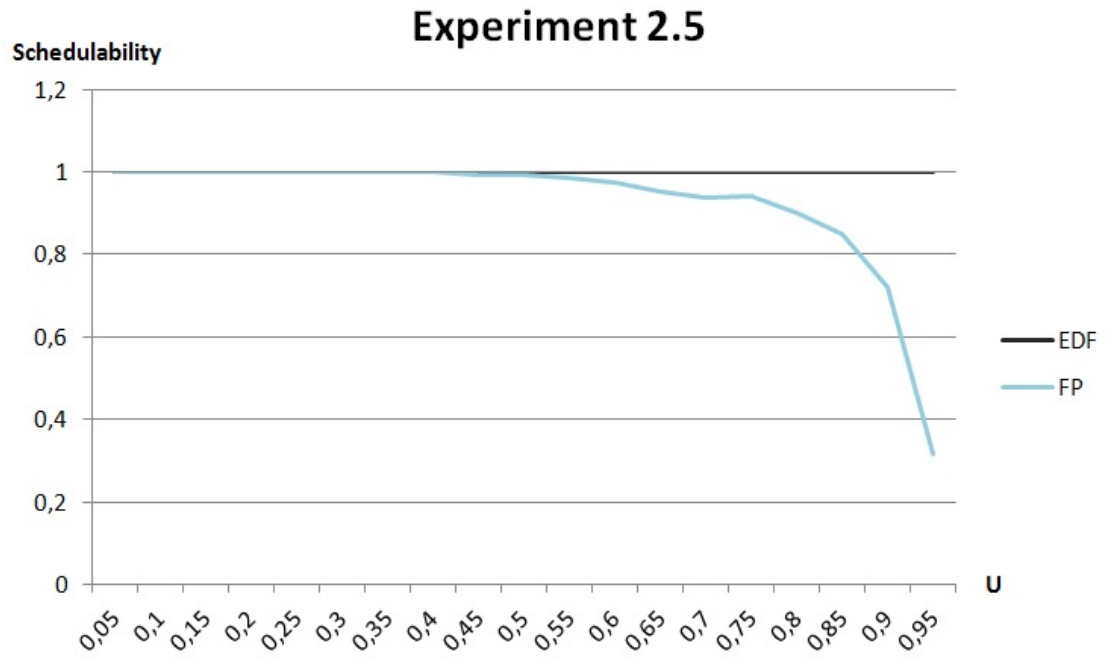

Figure 7.8: Schedulability ratio EDFvsFP, 7 periodic tasks, one AVR with JerkMinus=-5000rad/s^3, JerkPlus=5000rad/s^3, AlphaMinus=-700rad/s^2, AlphaPlus=700rad/s^2

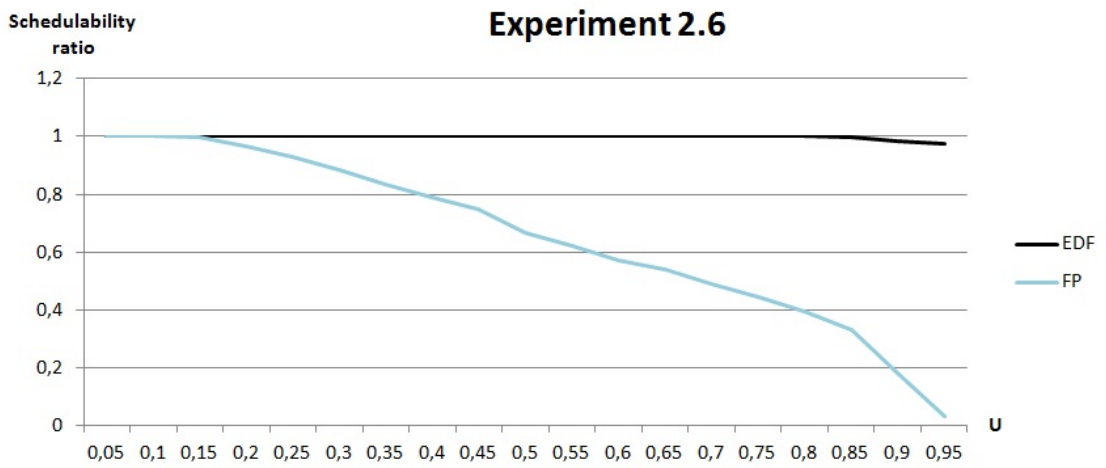

Figure 7.9: Schedulability ratio EDFvsFP, 7 periodic tasks, one AVR with JerkMinus=-10000rad/s^3, JerkPlus=10000rad/s^3, AlphaMinus=-5000rad/s^2, AlphaPlus=5000rad/s^2

Look back at acceleration and velocity profiles in chapter 5, in order to quantify how these variables change over time (Fig.5.1, ... Fig.5.11).

There are no relevant changes in the graphics, because my choice is to use realistic acceleration values that can be applied to the real world.
In order to see a lower schedulability value, an unfeasible maximum acceleration value should be set: 5000rad/s^2 and Jerk: 10000rad/s^3 (Fig.6.9); FP highly decrease the performance because of the increased number of AVRTask activations. On the other hand EDF is affected by the relative deadline reduction caused by high acceleration values.

Experiment 3: Overload conditions

Now the schedulability profile when the total utilization factor is greater than 1 will be examined.
The maximum acceleration and jerk values are (100,300), the other parameters are the same as in Experiment #2.
EDF drops to zero immediately after U=1, as expected, but always performs better than FP from schedulability point of view also in overload conditions.

Experiment 4: Number of periodic tasks dependency

Another parameter that could affect schedulability is the number of periodic tasks.
Increasing this value will not overload the system, because the total utilization is randomly distributed among tasks, using UUnifast; instead each periodic task will have shorter computation times.

Experiment 5: Mean periodic task period dependency

Look at these period values:

- (1,50) ms for periodic tasks.
- (9,120)ms for AVRTask.

They are overlapping; in addition priorities for periodic tasks are equal to periods, while for AVRTask is set to 9ms.
In this case there are few possibilities that period for periodic tasks is below 9ms and consequently that they have higher priority than the AVR one.
The goal is to understand what happens when the T_{med} (mean period value), varies from 25ms to 115ms with step 5ms; the interval lenght remains costant at 50ms.
The other simulation parameters are the same as in Experiment #2.
Increasing the mean period value, the probability that periodic tasks' periods are higher

than AVRTask' s one increase and consequently the optimal priority assignment becomes easier to be respected.

Experiment 6: ρ_U dependency

Setting a fixed value for U, in this case 0.7, how schedulability reacts to ρ_U variations? This means increasing the utilization factor for the AVRTask, and consequently it' s computation times, at the expense of periodic tasks' ones. The other simulation parameters are the same as in Experiment #2.

7.3.2 Maximum Normalized Tardiness

It is interesting to observe the behaviour of such systems in case of overload conditions ($U > 1$).
Of course the most significant parameter in this scenario is the:

$$Tardiness = max\{0, f - d/D\}$$

Where f is the finishing time, d is the absolute deadline and D is the relative deadline. Simulation parameters are the same used for the *Maximum Response Time*, except the utilization factor values range, that becomes [0.7, 1.5] with step 0.05.
In this case two tasks with different priority levels are used (the AVR one with the highest, the other one with lower priority).
The most evident result from this graph is the high value reached by EDF in the AVRTask case, while FP tardiness decrease with the increasing priority.
This means that despite from schedulability point of view EDF outperforms FP, in terms of maximum normalized tardiness EDF heavy penalizes AVRTask and higher priority(shorter periods) periodic tasks.
On the other hand FP never reaches so high values as EDF ones and seems to be more fair from tardiness point of view, reflecting the corresponding task priority.

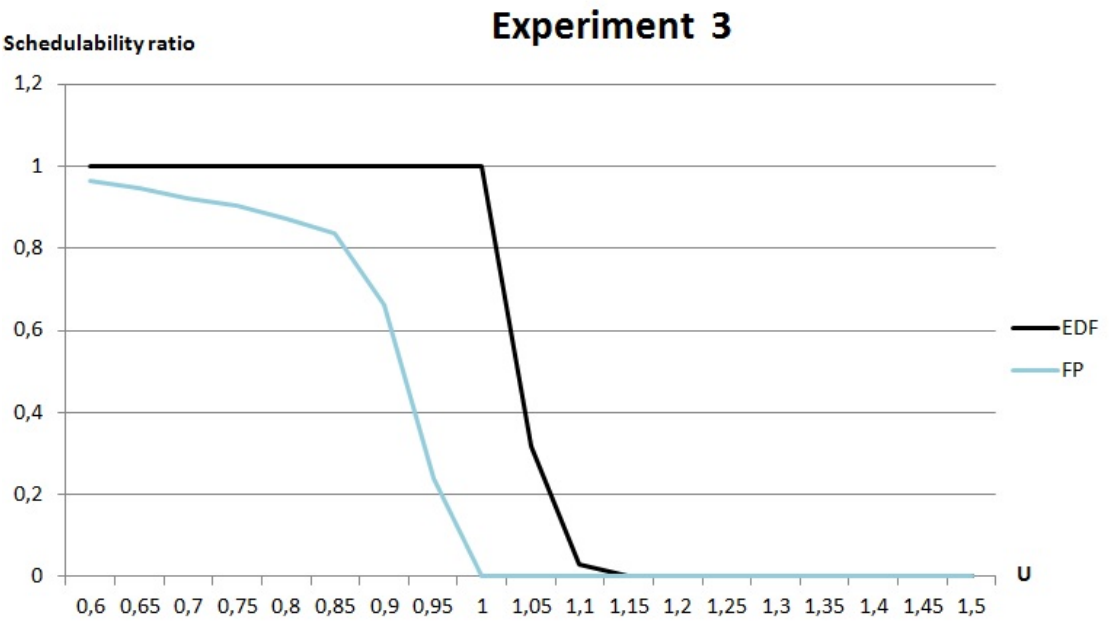

Figure 7.10: Schedulability ratio EDFvsFP, 7 periodic tasks, one AVR with JerkMinus=-300rad/s^3, JerkPlus=300rad/s^3, AlphaMinus=-100rad/s^2, AlphaPlus=100rad/s^2

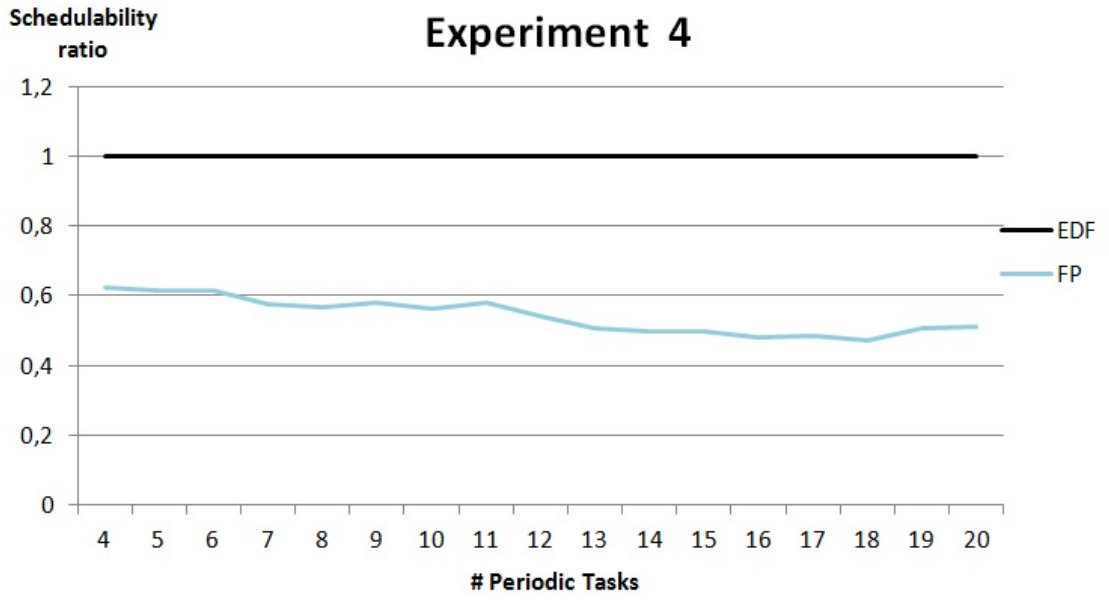

Figure 7.11: Schedulability ratio EDFvsFP, one AVR, U=0.7, JerkMinus=-300rad/s^3, JerkPlus=300rad/s^3, AlphaMinus=-100rad/s^2, AlphaPlus=100rad/s^2

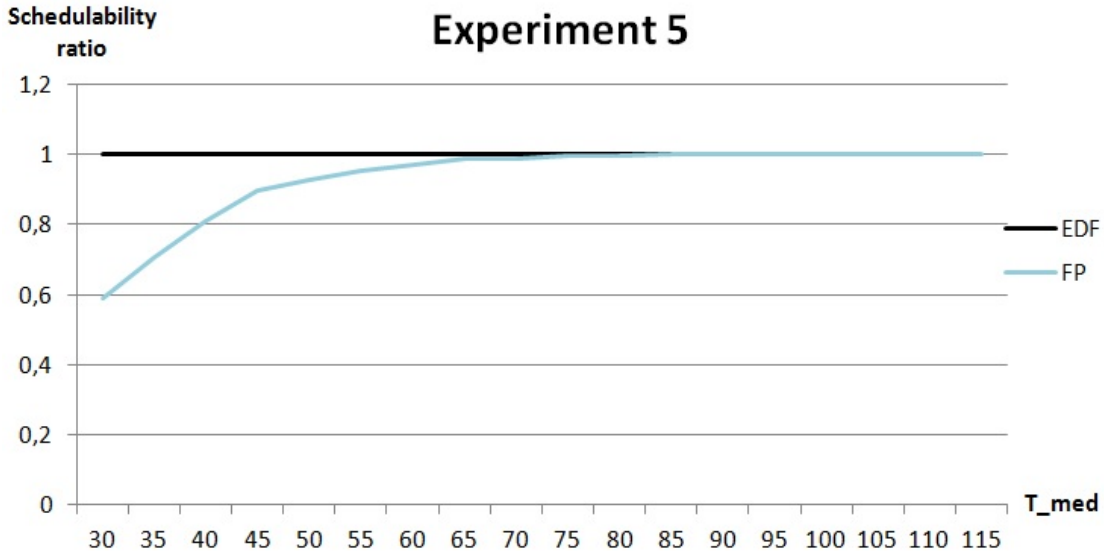

Figure 7.12: Schedulability ratio EDFvsFP, 7 periodic tasks, one AVR, U=0.7

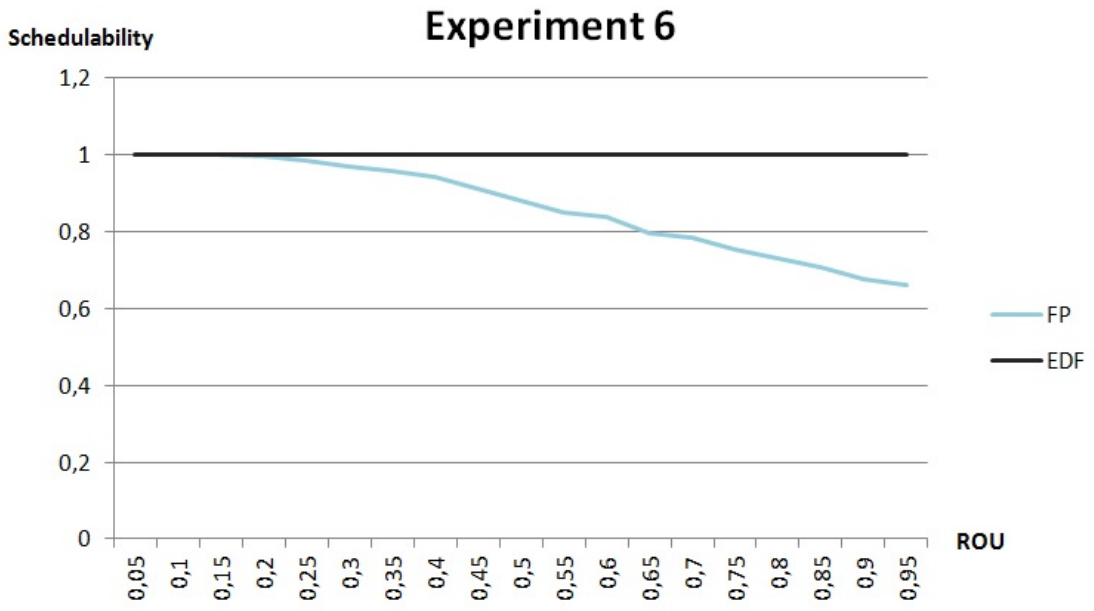

Figure 7.13: Schedulability ratio EDFvsFP, 7 periodic tasks, one AVR, U=0.7

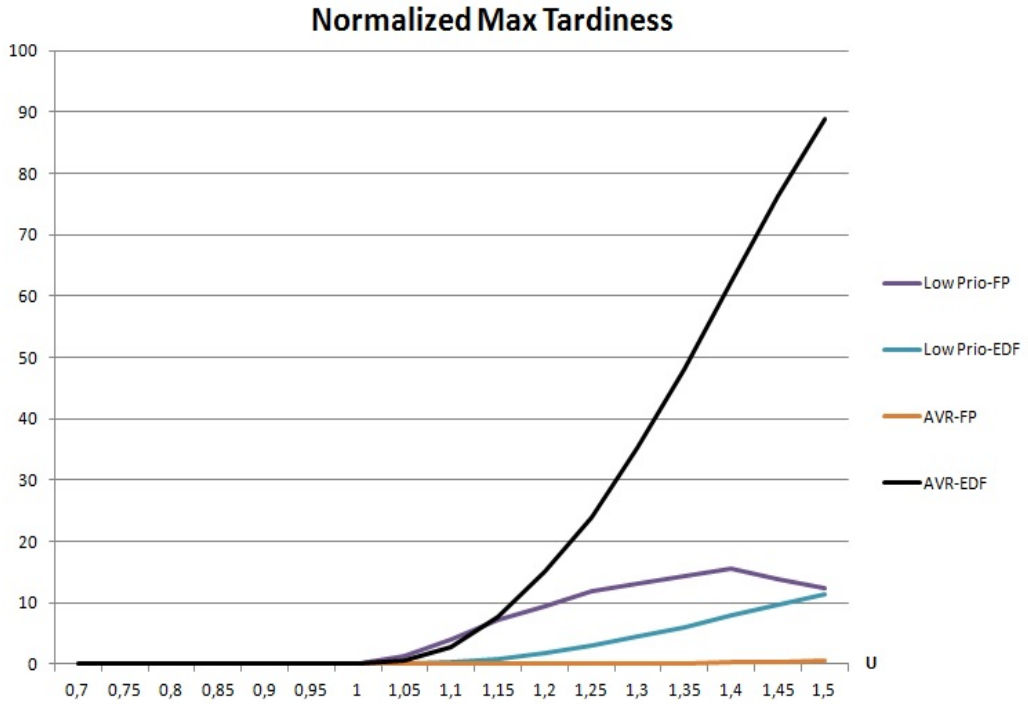

Figure 7.14: Max Tardiness EDFvsFP, AVRtask (highest priority)

Chapter 8

Conclusions

Engine control is characterized by computational activities that are triggered by specific crankshaft rotation angles and are designed to adapt their functionality based on the angular velocity of the engine.
Automotive applications include tasks that are activated with different mechanisms. Some engine control tasks are activated by a timer at a fixed rate (periodic tasks), whereas other tasks are linked to the rotation of the crankshaft and are activated at specific rotation angles (angular tasks). The activation rate of such angular tasks is therefore proportional to the engine speed: the higher the engine speed, the higher the activation rate.

As a result, angular tasks have a variable activation rate (dependent on the engine speed) and a self-adaptive behavior implemented through a set of mode changes. For this reason, they are often referred to as adaptive variable-rate tasks (or AVR tasks). Several models have been presented in literature in order to analyze the schedulability of such tasks (see References). Following Buttazzo, Di Natale, Biondi and others' work from Retis Laboratory, the C++ based scheduling simulator for AVR taks has been implemented, using already available software modules in *RTLib2.0*.

One of the most important feature that has been focused is the comparison between Earliest Deadline First and Fixed Priority scheduling algorithms (see [12]), in this new scenario. The main problem that has been addressed is the schedulability one: deadline misses may lead the entire system in bad performance state or even to failure, depending on the missing task functionality.

The results obtained are quite general, random values for tasks' parameters are generated each time, but the values range are chosen taking into account the application field involved.
As expected, there are no shocking results, in the sense that theoretical studies are in line with simulation' s statistical values returned.
Several experiments have been carried out, with the aim to indentify the most relevant parameters, that affect schedulability performances.

The first important result is that fixed priority scheduing is not the best choice in this environment, considering all the possible interarrival time's values of AVRTask, there are for sure some engine speed for which the priority assignment is far from being optimal, causing the schedulability value to decrease.
On the other hand EDF is indipendent from the variable period values, being influenced only by relative deadlines that are computed taking into account the current angular velocity of the engine.
The only case that brings EDF schedulability a little under 1.0 is when using the not realistic acceleration value $5000 rad/s^2$.
However in the real world, only few operating systems provide the EDF scheduling algorithms (e.g., Erika Enterprise [13], which is also OSEK-certified for automotive systems, and Linux, using the SCHED DEADLINE scheduling class [14]).
The best priority assignment, even if not optimal, is to set the AVRTask priority as the minimum interarrival time: $AngularPeriod/MaxAngularVelocity$; the other experiments are carried out by assuming this priority choice.
The total number of periodic tasks don't affect the schedulability ratio, of course, because the total utilization is randomly divided among them.
Increasing the utilization factor for AVRTask, while the total utilization is kept constant, let the schedulability ratio decrease, only in the case of FP.

Another experiment shows that with the increase of the mean period value for periodic tasks, the FP schedulability ratio grows until 1.0, because the probability that the AVRTask period is the lowest increase.
About behaviours in overload conditions, of course schedulability drop to 0 in both cases immediately after utilization value equal to 1; maximum normalized tardiness results are interesting: AVRTask under EDF can end more then 60 times the relative deadline after

miss. FP seems more fair from this point of view: tardiness values are not so high like the EDF case and they decrease with the increasing priority of course.
Some of the simulation results of this thesis have been used for a research paper with Buttazzo and Biondi, namely "Feasibility Analysis of Engine Control Tasks under EDF Scheduling", submitted to "Euro Micro Conference on Real Time Systems".

Bibliography

[1] G. Buttazzo, L. Abeni, and G. Lipari, "Elastic task model for adaptive rate control," in IEEE Real Time System Symposium, Madrid, Spain, December 1998.

[2] G. Buttazzo, G. Lipari, M. Caccamo, and L. Abeni, "Elastic scheduling for flexible workload management," IEEE Transactions on Computers vol. 51, no. 3, pp. 289–302, March 2002.

[3] L. Sha, R. Rajkumar, J. P. Lehoczky, and K. Ramamritham, "Mode change protocols for priority-driven preemptive scheduling," Real-Time Systems, vol. 1, no. 3, pp. 243–264, December 1989.

[4] J. Real and A. Crespo, "Mode change protocols for real-time systems: A survey and a new proposal," Real-Time Systems, vol. 26, no. 2, pp. 161–197, March 2004.

[5] N. Stoimenov, S. Perathoner, and L. Thiele, "Reliable mode changes in real-time systems with fixed priority or EDF scheduling," in Proceedings of the Design, Automation and Test Conference in Europe (DATE 2009), Nice, France, April 20-24, 2009.

[6] J. Kim, K. Lakshmanan, and R. Rajkumar, "Rhythmic tasks: A new task model with continually varying periods for cyber-physical systems," in Proc. of the Third IEEE/ACM Int. Conference on Cyber-Physical Systems (ICCPS 2012), Beijing, China, April 17-19, 2012, pp. 28–38.

[7] V. Pollex, T. Feld, F. Slomka, U. Margull, R. Mader, and G. Wirrer, "Sufficient real-time analysis for an engine control unit with constant angular velocities,"

in Proc. of the Design, Automation and Test Conference in Europe, Grenoble, France, March 18-22, 2013.

[8] R. I. Davis, T. Feld, V. Pollex, and F. Slomka, "Schedulability tests for tasks with variable rate-dependent behaviour under fixed priority scheduling," in Proc. 20th IEEE Real-Time and Embedded Technology and Applications Symposium, Berlin, Germany, April 15-17, 2014.

[9] A. Biondi, A. Melani, M. Marinoni, M. D. Natale, and G. Buttazzo, "Exact interference of adaptive variable-rate tasks under fixed-priority scheduling," in Proceedings of the 26th Euromicro Conference on Real- Time Systems (ECRTS 2014), Madrid, Spain, July 8-11, 2014.

[10] G. Buttazzo, E. Bini, and D. Buttle, "Rate-adaptive tasks: Model, analysis, and design issues," in Proc. of the Int. Conference on Design, Automation and Test in Europe, Dresden, Germany, March 24-28, 2014.

[11] A. Biondi, M. Di Natale, G. Buttazzo, "Response-Time Analysis for Real-Time Tasks in Engine Control Applications".

[12] G.Buttazzo, "Rate Monotonic vs. EDF: "Judgment Day", in Real-Time Systems,vol. 29, pp. 5–26, 2005.

[13] "Erika enterprise: an OSEK compliant real-time kernel." [Online].
Available: http://erika.tuxfamily.org/drupal/

[14] D. Faggioli, F. Checconi, M. Trimarchi, and C. Scordino, "An edf scheduling class for the linux kernel," in Proc. of the 11th Real-Time Linux Workshop (RTLWS), Dresden, Germany, September 28-30, 2009.