YARTISS: A Generic, Modular and Energy-Aware Scheduling Simulator for Real-Time Multiprocessor Systems

Younès Chandarli, Manar Qamhieh, Frédéric Fauberteau, Damien Masson

YARTISS: A Generic, Modular and Energy-Aware Scheduling Simulator for Real-Time Multiprocessor Systems

Younès Chandarli, Manar Qamhieh, Frédéric Fauberteau and Damien Masson

October 20, 2014

Abstract

In this report, we present a free software written in Java, *YARTISS*, which is a real-time multiprocessor scheduling simulator. It is aimed at comparing user-customized algorithms with ones from the literature on real-time scheduling. This simulator is designed as an easy-to-use modular tool in which new modules can be added without the need to decompress, edit nor recompile existing parts. It can simulate the execution of a large number of concurrent periodic independent tasksets on multiprocessor platforms and generate clear visual results of the scheduling process (both schedules and tunable metrics presentations). Other task models are already implemented in the simulator, like graph tasks with precedence constraints and it is easily extensible to other task models. Moreover, *YARTISS* can simulate tasksets in which energy consumption is a scheduling parameter in the same manner as Worst Case Execution Time (WCET).

1 Introduction

The real-time scheduling problem has been studied by many researchers since decades, and a lot of algorithms and approaches have been proposed over the years to optimize the scheduling of real-time tasks on single and multiple processor systems. In order to check whether a set of tasks respects its temporal constraints when a scheduling algorithm is used or to evaluate the efficiency of a new approach, software simulation against other algorithms is considered as a valid comparison technique and it is commonly used in the evaluation of real-time systems.

Unfortunately, there is no standard simulation tool approved by the real-time community and the existing tools are usually hard to be extended due to various reasons such as code complexity, software copyrights or poor documentation. As a result, most of the researchers tend to create their own simulation tools. This situation raises some concerns. On one hand, the presented results are hard to be validated without careful examination of the used simulation tool. So these results might be biased toward the proposed approach either by adapting the generation of tasks or by biased implementation against the compared algorithms. On another hand, out-of-date simulation tools or the lack of good documentation can incite researchers to avoid the existing tools and create new ones, which leads to repetitive algorithms' implementations specially the common ones (e.g. *Earliest-Deadline First* (EDF), *Rate-Monotonic* (RM) and *Deadline-Monotonic* (DM)).

Besides the time and the effort spent by researchers while developing new simulation tools, the process of implementing comparable algorithms can be sometimes complicated and time-consuming. If a standard simulation platform succeeds to emerge, one can re-use already-implemented policies from literature and compare them with new policies without the need to understand their specifications and particularities. Finally, the simulation protocols can be standardized, and easily describable by the use of such a reference tool.

In this report, we introduce *YARTISS*, a new simulation tool for real-time multiprocessor systems, which provides various functions to simulate the scheduling process of real-time tasksets and their temporal behavior when a scheduling policy is used. The main feature of *YARTISS* is its genericity, by which we aim to overcome the previously mentioned problems. Its architecture is designed in a way to allow new users to add their own policies and algorithms and extend the simulator easily, with no need to modify the core of the simulator or even understand how it is built or works. *YARTISS* is an extension of previous simulator projects [13, 5] designed earlier by our research team. We learned a lot from these attempts and we included this experience in the development of *YARTISS*.

YARTISS is written in Java programming language, which is a popular object-oriented language that offers valuable attributes regarding code portability. In order to ensure independence between the different features of the simulator and to reduce the possibility of massive failures among them, we used modern programming paradigms like module-oriented programming and Java Unit tests (JUnit). We tried to develop *YARTISS* keeping in mind that in order for a simulator to become a reference tool, it should have the following properties: 1) the software must be available under an open source license which gives any researcher the freedom to analyze, verify or modify its implementation without the permission of the copyright holder; 2) the *Application Programming Interface* (API) of the software must be well-documented and the developer who wants to add or modify an algorithm should be able to do so easily with no need to read the entire source code in order to understand its behavior ; 3) each part of the simulator (its core, the task generator, the result analyzer, ...) must be independent from the other parts, and easily replaceable by an external module ; and 4) the simulator has to be easy to use in a way that a non-developer researcher can be able to use it easily. Due to its genericness and modularity, we hope that *YARTISS* makes a valuable contribution to the long process of developing a standard simulation tool recognized by the real-time scheduling research community.

The structure of this report is as follows: we review related works and examples of real-time simulators in Section 2. Section 3 contains our motivation and it shows the development history of *YARTISS*. Then in Section 4 we present the various functionalities of our simulation tool. The architecture design of the simulator is described in Section 5. We present a case study in Section 6 in order to demonstrate the extensibility of the tool. Information about the available versions of *YARTISS* and its download and install instructions are provided in Section 7. Finally, we discuss our future work in Section 8 and Section 9 brings a conclusion to this report.

2 Related Works

In this section we outline the existing works related to simulation and visualizing of the scheduling of real-time tasks while identifying their strengths and shortcomings. There are a lot of tools to test and visualize the temporal and execution behavior of real-

time systems, and they are divided mainly into two categories: the execution analyzer frameworks and the simulation software. Regarding the execution analyzers, one can refer to RESCH [10] which is a loadable real-time scheduler framework for linux. Also, Grasp[8] which is a tool set of tracing and measuring the scheduling of real-time tasks. Furthermore, LitmusRT [2] is a real-time extension to linux kernel for multiprocessor systems and it supports a set of scheduling modules and synchronization supports.

Among open-source simulation tools, we start by referring to MAST [7] which is a modeling and analysis suite for real-time applications that is developed in 2000. MAST is an event-driven scheduling simulator that permits modeling of distributed real-time systems and offers a set of tools to *e.g.* test their feasibility or perform sensitive analysis. Another known simulator is Cheddar [20, 19] which is developed in 2004 and it handles the scheduling of real-time tasks on multiprocessor systems. It provides many implementations of scheduling, partitioning and analysis of algorithms, and it comes with a friendly *Graphical User Interface* (GUI). Unfortunately, no API documentation is available to help with the implementation of new algorithms and to facilitate its extensibility. Moreover, Cheddar is written in Ada programming language [16] which is used mainly in embedded systems and it has strong features such as modularity mechanisms and parallel processing. Ada is often the language of choice for large systems that require real-time processing, but in general, it is not a common language among developers. We believe that the choice of Ada as the base of Cheddar reduces the potential contributions to the software from average developers and researchers.

Finally, STORM [21] and FORTAS [4] are tools which, as *YARTISS*, are written in Java. In 2009, STORM is released and it is described as a simulation tool for Real time multiprocessor scheduling. It has modular architectures (both software and hardware) which simulate the scheduling of tasksets on multiprocessor systems based on the rules of a chosen scheduling policy. The specifications of the simulation parameters and the scheduling policies are modifiable using an XML file. However, the simulator tool lacks a detailed documentation and description of the available scheduling methods and the features of the software. On the other hand, FORTAS is a real-time simulation and analysis framework which targets uniprocessor and multiprocessor systems. It is developed mainly to facilitate the comparison process between the different scheduling algorithms, and it includes features such as task generators and computation of results of each tested and compared scheduling algorithm. FORTAS represents valuable contributions in the effort towards providing an open and modular tool. Unfortunately, it seems to suffer from the following issues: its development is not open to other developers for now, we can only download *.class* files, no documentation is yet provided and it seems that no new version has been released to public since its presentation in [4].

During the development of *YARTISS*, we learned from those existing tools and we included some of their features in addition to others of our own. Our aim is to provide a simulation tool for multiprocessor systems that is easily extendable by fellow researchers and developers, and it can be used to compare, simulate and visualize the scheduling of real-time tasks on multiprocessor systems.

3 Development History of *YARTISS*

YARTISS is the fourth simulation tool developed by our team during the last few years. In this section we will present each one of these tools, their contributions, challenges and limitations. Each one of these tools had been built for a specific purpose and consumed

a relatively long period of time to be developed. We used the experience from the earlier simulators in the design of *YARTISS*. This is done by summing and optimizing their functionalities while avoiding their limitations.

Our first try in writing a real-time system simulator was called RTSS [13] and it was developed between 2005 and 2008. RTSS was initially developed to test some scheduling algorithms on uniprocessor systems in order to handle temporal fault tolerance, such as preemptive fixed priority, EDF and D^{OVER} algorithms citedover. Later, it was extended in order to test aperiodic tasks with task server of type Polling and Deferrable Servers and their handling algorithms [14, 15]. RTSS suffered from some problems. For example, lots of modifications had been made in a hurry with certain assumptions on the behavior of existing classes without proper documentation. Also, RTSS was not easily extendable and a modification to a certain part of the simulator could result in errors in another completely different part. Moreover, although the tool was initially programmed in Java, it began to rely more and more on bash scripts which are used mainly to launch the simulation process and to transform outputs into human readable files.

Based on RTSS, a second simulation tool was developed between 2008 and 2011. It was called RTMSim [5] and it targeted the scheduling simulation of multiprocessor platforms [6]. In the meanwhile, RTSS had become such complicated and unmaintainable that we had to start it over rather than extend its functionalities to multiprocessor systems. The general key ideas of RTSS were kept for RTMSim. Unfortunately, the validated parts of RTSS, which were of no interest at the time, were not re-implemented in RTMSim and thus they were lost (as the implementation of D^{OVER} [11] scheduling algorithm, for example).

A third try was made in early 2011. RTSS v2 [13] was developed as a rebuild of RTSS in the aim of including energy consuming tasks and used for energy-harvesting systems as in [3]. Unfortunately, even if RTSS v2 is more usable today than the first version of RTSS, it still suffers from the same problems of the original tool, which were the poor documentation, the lack of modularity and usability features which can be used to simulate and exploit results of large scale simulations. Moreover, it seemed difficult to extend it to simulate multiprocessor platforms.

So we came to the development of a new software: *YARTISS*. From the start, we aimed to produce a tool where the task models, the number of processors and scheduling behavior such as energy consumption models can be easily added and modified. Another important point is the usability of the user interface to produce human readable traces of the simulation process of scheduling algorithms. Our goal was to develop a simulation tool which is able to produce evaluations as well as to debug various algorithms including the energy-related algorithms. When we wanted to use *YARTISS* for another purpose, namely the simulation of scheduling dependent real-time tasks of the Directed Acyclic Graph model (see [18]), this was done without any problems which validates the extensibility of our simulator . This will be explained as a case study in more details in Section 6.

4 Functionalities

There are two main functionalities in our simulation tool, the first is the simulation of the execution and temporal behavior of a taskset scheduling using a specific scheduling policy. The second functionality is the large-scale comparison of several scheduling policies in different scenarios. Both functionalities require a third feature which is the random taskset generation.

In this section we explain all functionalities of *YARTISS* in details while showing their specifications and various characteristics regarding the problem of scheduling real-time systems.

4.1 Single Taskset Simulation

This functionality concerns with the simulation of a taskset execution on multiprocessor systems with a specific scheduling policy. The used tasksets can be loaded into the simulator either through the GUI by using a file or entering the parameters manually, or by using taskset generators. We can parameterize the desired simulation and run it easily by the click of a button, then several views are proposed. The simulation parameters are the taskset, the number of processors, the scheduling algorithm and the energy profile.

4.1.1 Task Models

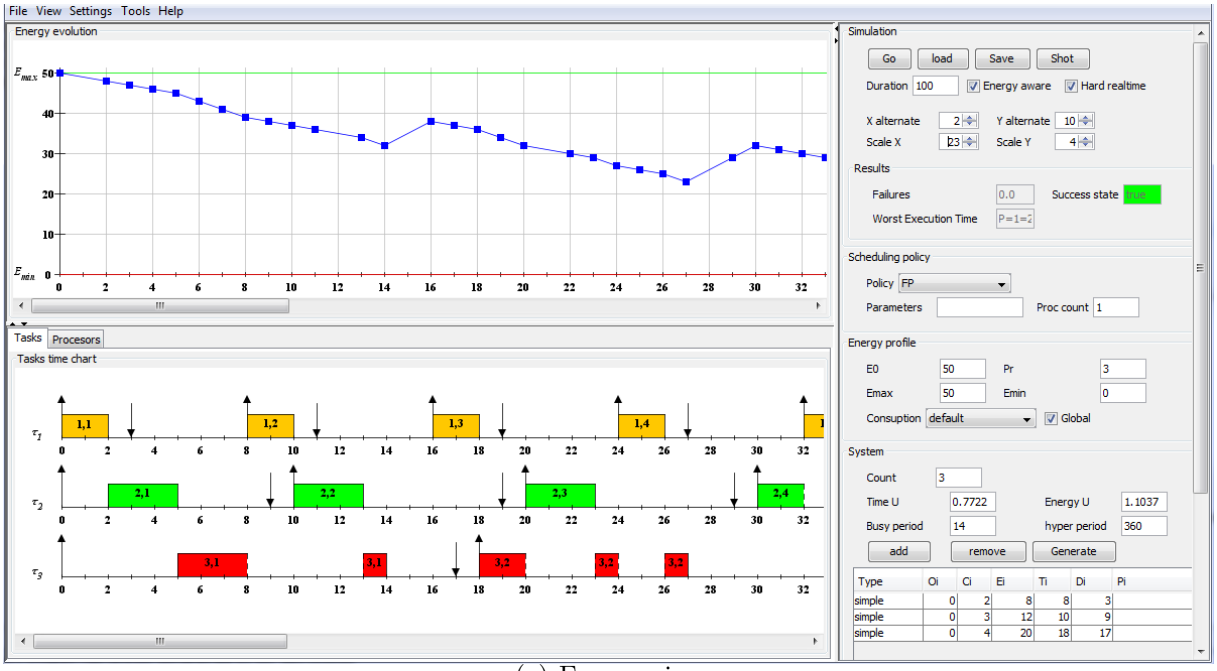
YARTISS offers an open architecture that greatly facilitates the integration of different task models. The current version proposes two models, the first one is the Liu and Layland task model with energy related parameters. All tasks are considered independent and each task τ_i is characterized by its WCET C_i, its worst case energy consumption E_i, its period T_i and its deadline D_i. The second model is the Directed Acyclic Graph (DAG) task model which is a common real-time dependent task model for multiprocessor systems. It is used to implement systems consisting of number of subtasks with dependencies to control their execution flow. In this model, a graph task G_i is a collection of real-time subtasks $\{\tau_{i,1}, \tau_{i,2}, ...\tau_{i,q}\}$ that share the same deadline D_i and period T_i of the DAG, and they differ in their own WCET $C_{i,j}$. The directed edges between the tasks of the graph determine their precedence constraints, and since each task in the graph might have more than one successor and predecessor, concurrent execution can be generated.

These two models are the common models of independent and dependent real-time tasks, and they can have different characteristics regarding deadlines (implicit, constrained), and regarding periods (periodic, sporadic). We show in Section 6 a case study to demonstrate how to easily integrate new task models into *YARTISS*.

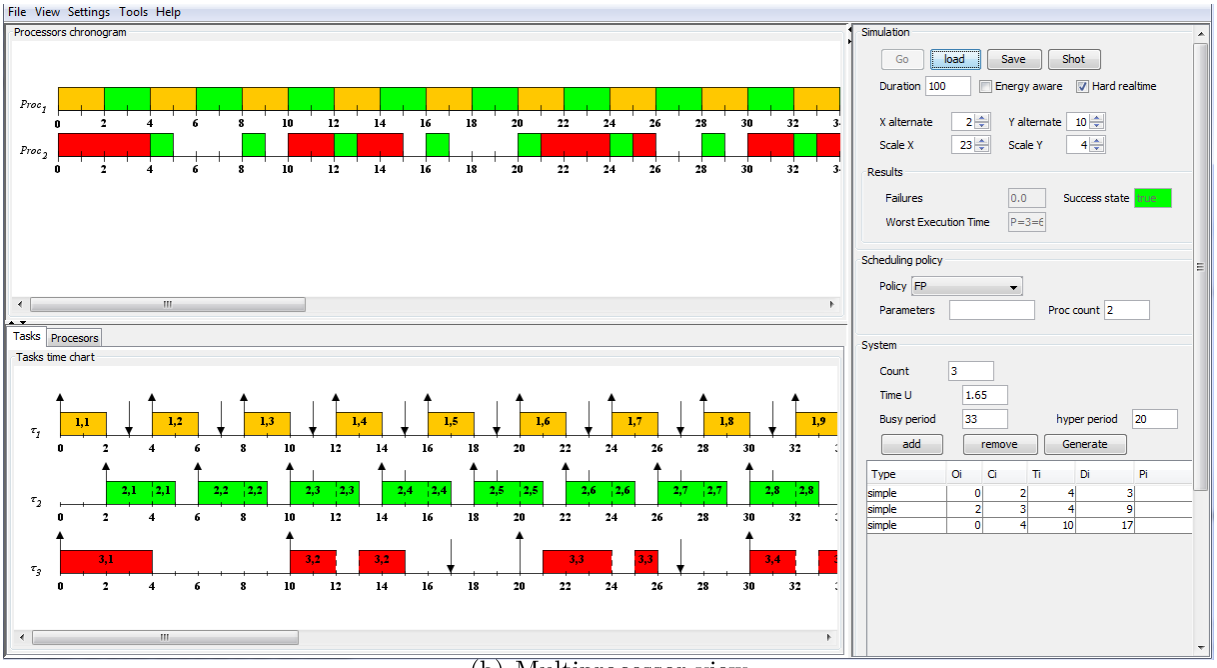
4.1.2 Uniprocessor / Multiprocessor

Our simulation tool supports multiprocessor scheduling. According to this, the number and the type of processors is passed as a parameter of the simulation. Taskset generators are developed to be compatible with the multiprocessor scheduling and they can have a total system utilization greater than 1. Initially we considered a single type of system platform which consists of identical unit-speed processors. This model is extended in *YARTISS* to include heterogeneous systems in which the processors are different in their execution rate.

Using the simulator, one can implement and test his own multiprocessor algorithms and partitioning policies. Also, we considered a global scheduling of tasksets on multiprocessor systems. It is defined as the scheduling in which the execution of a job can be interrupted by higher priority jobs and the execution can be resumed on a different processor. By default, some multiprocessor scheduling algorithms were implemented in *YARTISS* on multiprocessor systems like EDF and FP.



(a) Energy view



(b) Multiprocessor view

Figure 1: Energy and Multiprocessor Simulation views

4.1.3 Energy Profile

Unlike many other simulators, *YARTISS* permits us to simulate the production and the consumption of energy in real-time systems. It permits the user to model an energy harvester, such as a battery or a capacitor with limited or unlimited capacity. A renewable energy source can be modeled also using a charging function. It is possible as well for the user to implement and use customized energy profiles. Figure 1 shows a screenshot of the GUI of *YARTISS*. It shows the energy level and its consumption during the scheduling process. The energy view which is used to print the energy level is among many views provided by *YARTISS* to show other metrics, such as the system slack time for example.

Energy Source Model: We have implemented an energy source profile that models a renewable energy source represented by a battery with limited capacity and a linear charging function. Other models can be added by the user by implementing the profile interface and injecting it into the engine of the simulator. This process is not complicated and it can be done without the need to modify other packages in the simulator.

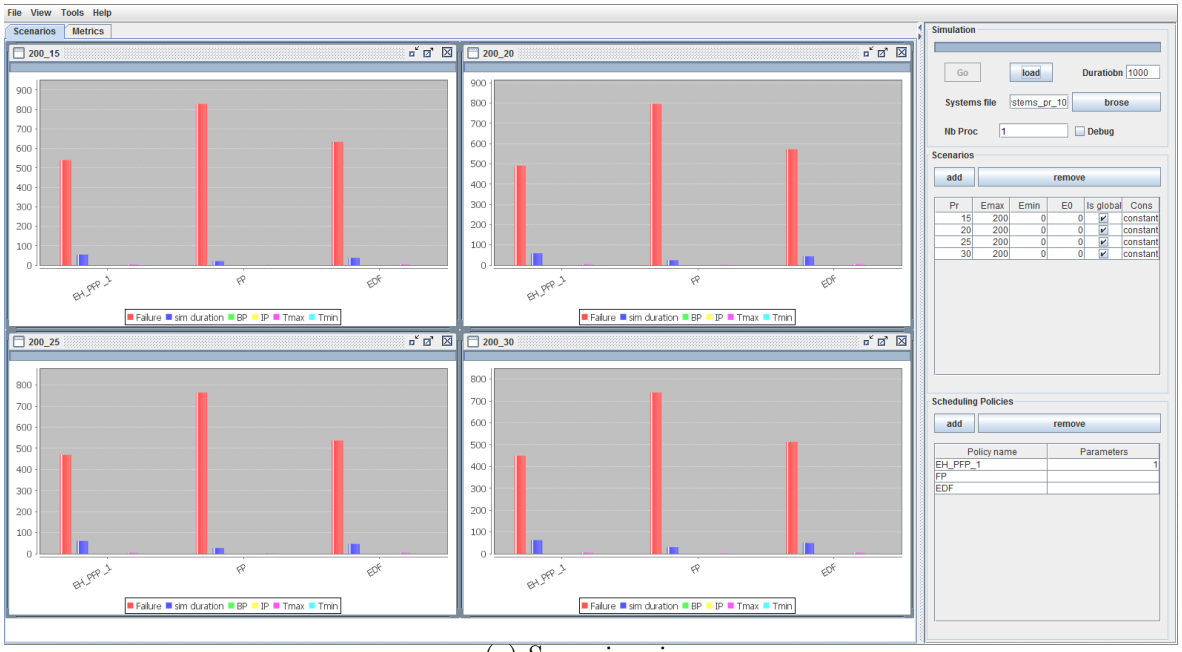
Consumption Model: Sometimes, it is important to note that the energy consumption of a task must be modeled independently from its WCET as in the case of [9]. Therefore, our simulator provides the ability to define a consumption profile for each task of the system or to choose one global profile applied to all tasks. A consumption model is represented by a function and it must be able to provide the amount of energy consumed within a time interval during the tasks execution i.e. the integral of the consumption function. Implemented models so far are: *Linear consumption* (not realistic but permits establishing some interesting preliminary conclusions) and *Early instantaneous consumption* where all the energy cost of a task is consumed as soon as a task is scheduled. This later model is assumed to represent the worst case scenario.

4.1.4 Scheduling Policy

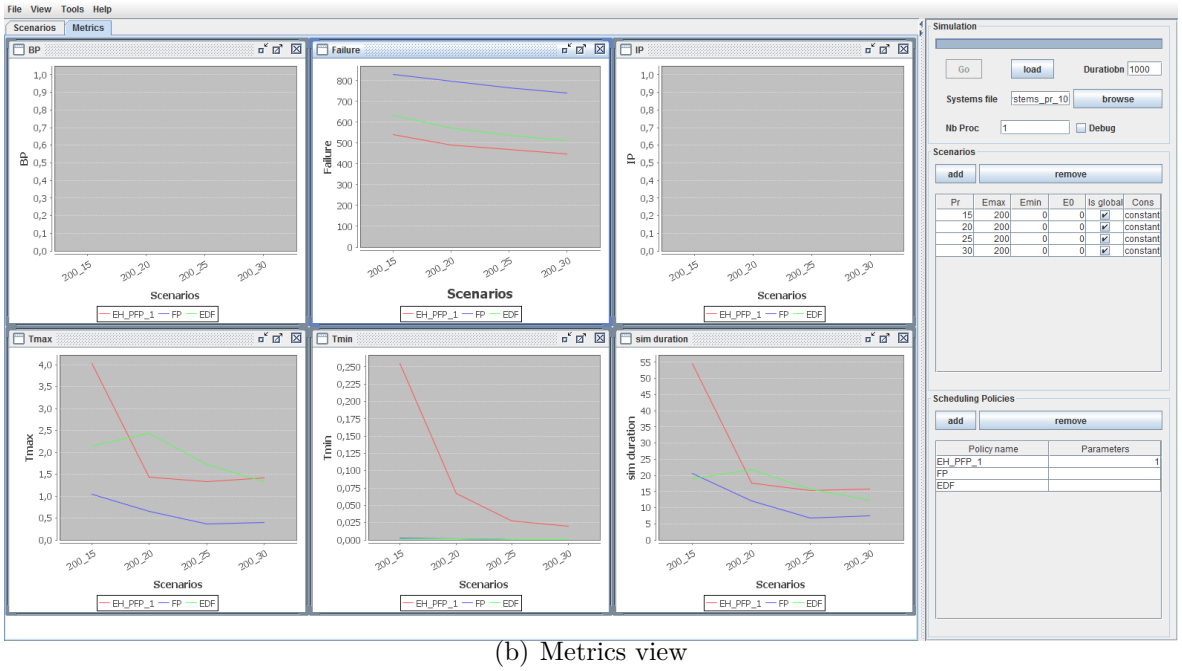
The main purpose of the simulator is to test scheduling algorithms by comparing them to show their performances and efficiency. Much attention has been focused on the design of this part of *YARTISS* to make it as generic as possible so that users can add, override and inject their own scheduling policies easily. There are currently twenty algorithms implemented in *YARTISS* with different priority assignment techniques such as fixed-job, fixed-task and dynamic priority. It also includes classical algorithms such as RM, DM, EDF, LLF and heuristics for the energy constrained scheduling problem. Users can add new scheduling policies easily, and later in this document, we will demonstrate in details how to integrate a new policy independently from the rest of the modules of the simulator.

4.2 Run Large Scale Simulations

A major utility of *YARTISS* is the large scale comparison of several algorithms or scheduling policies. It is similar to the single taskset simulation but it is done on a large dataset with different scenarios. The comparison results are based on statistics such as the number of schedulability failures or deadline misses, the system's lifetime, the amount of time spent at maximum and minimum energy levels and the average duration of idle and busy periods. Multiple simulations can run simultaneously due to the use of the multi-threading concept supported by Java. As a result, the duration of simulations is greatly reduced through the parallelism of the used hardware.



(a) Scenarios view



(b) Metrics view

Figure 2: Concurrent large scale simulations: histogram and curves views

4.3 Taskset Generation

Performing large-scale tests requires a large dataset of tasks. For the simulation results to be credible, the used tasksets should be randomly generated and varied sufficiently. The simulator provides the ability to choose a generator according to the desired scenarios and algorithms. The current version includes by default a generator based on the UUniFast-Discard algorithm [1] coupled with a hyper-period limitation technique [12] adapted to energy constraints. This algorithm generates tasksets by dividing among them the CPU utilization ($U = \sum \frac{C_i}{T_i}$) and the energy utilization ($U_e = \sum \frac{E_i}{T_i \times Pr}$ where Pr is the recharging function) chosen by the user. The original version was not energy aware, and we had to adapt it to produce feasible systems regarding time and energy. The idea behind the algorithm is to distribute the system's utilization on the tasks of the system. When we add the energy cost of tasks to the system, we end up with two parameters to vary and two conditions to satisfy. The algorithm in its current version distributes U and U_e uniformly on the tasks then it finds the 2-tuple (C_i, E_i) which satisfies all the conditions namely U_i, U_e and energy consumption constraints. The operation is repeated several times until the desired 2-tuple approaches the imposed conditions. Finally, the algorithm returns as a result a time and potentially[1] energy feasible system. The user can use the default generator as he can write and integrate his own customized generator into *YARTISS*. The addition of the dependent taskset generator is described in details in Section 6.

Listing 1: An example of XML code represents a default taskset

```
<?xml version="1.0" encoding="UTF-8"?>
<dataset>
  <policy name="PFP_ASAP" nbParams="0"/>
  <simulation endValue="100" nbProc="1"/>
  <energyProfile E0="0" Emax="50" Emin="0" pr="3"/>
  <tasks nbTasks="2" type="Fixed Priority">
    <task deadline="3" firstRelease="0" period="8" priority="1" type="
        simple" wcee="10" wcet="2"/>
    <task deadline="9" firstRelease="0" period="10" priority="2" type="
        simple" wcee="3" wcet="3"/>
  </tasks>
</dataset>
```

4.4 Graphical User Interface (GUI)

To facilitate the use of the simulator by a large number of users, we provide our tool with a GUI to support the above-mentioned features in an interactive and intuitive way. After the simulation of a single system with an energy profile and a scheduling policy, the user can follow and analyze the scheduling process using three different views: a time chart, a processor view and the energy curve which monitors the energy levels of the system (as mentioned before, other data can be monitored and print on this view). In order to run simulations and get the comparison results of scheduling policies, the simulator offers a controlling view that allows the user to select the used scheduling policy, the energy scenario and to start running the simulation process. Thus, the user can see the simulation results as one graph per scenario or per comparison criterion. Then, the simulator can obtain results on a large number of randomly generated tasksets in order to evaluate a

[1] To the best of our knowledge, there is no necessary and sufficient feasibility test until now that takes into account energy constraints.

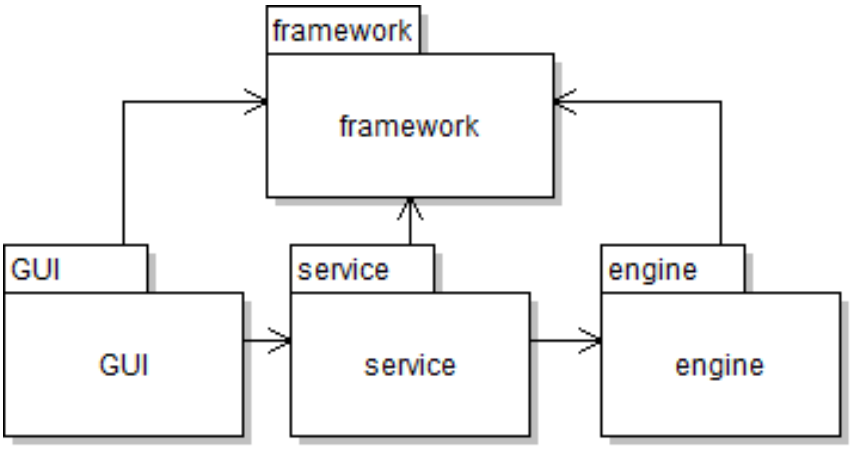


Figure 3: Modules connexion UML Diagram

scheduling policy, and easily explore the properties of a new algorithm. This view offers also a debugging view in which the user can analyze the results of comparison and can optionally display the time chart of each system and for each scheduling policy. This helps in analyzing and debugging scheduling policies. It can be also useful to find counter examples and to isolate degenerate cases. For example, in the case of energy scheduling, no optimal algorithm exists yet. In order to test empirically whether a new algorithm is optimal or not, a simple approach consists in simulating the scheduling of all possibilities of tasksets and ask the simulator to filter cases where the new scheduling algorithm fails whereas other heuristics succeed. If no such cases exist, then the scheduling algorithm is deemed optimal.

5 Architecture

Usually, the common concern of real-time simulation tools is the possibility to use the simulator in different contexts. In many cases, extending a simulator to include customized modules needs a lot of modifications and refactoring of the code. Careless modifications can lead to incompatibility with the original modules of the tool, or worst, this may lead to a different or wrong simulation results. Furthermore, most of the time, researchers do not like to spend much time or effort to understand and preserve an existing tool, and finally, the tool becomes difficult to maintain. In our team, we experienced this issue previously, and we noticed the importance of the design and the architecture of the tools for their extendibility. The more generic and flexible the software is, the less time and effort we spend to integrate new customized modules. For this reason and in order to meet the requirements mentioned in Section 3, we decided to merge the old versions of our simulator and provide a new tool with enhanced architecture and design. We have ensured that the design is as generic and open as possible by applying the appropriate design patterns and programming paradigms.

The simulation tool is divided into four main modules each with a specific responsibility: the engine (core) module of the simulator which contains all the classes necessary to simulate the scheduling of real-time systems, the service module responsible of handling the transparent interactions between the engine and the presentation, a module for GUI and finally a framework module which contains useful tools and classes necessary for the

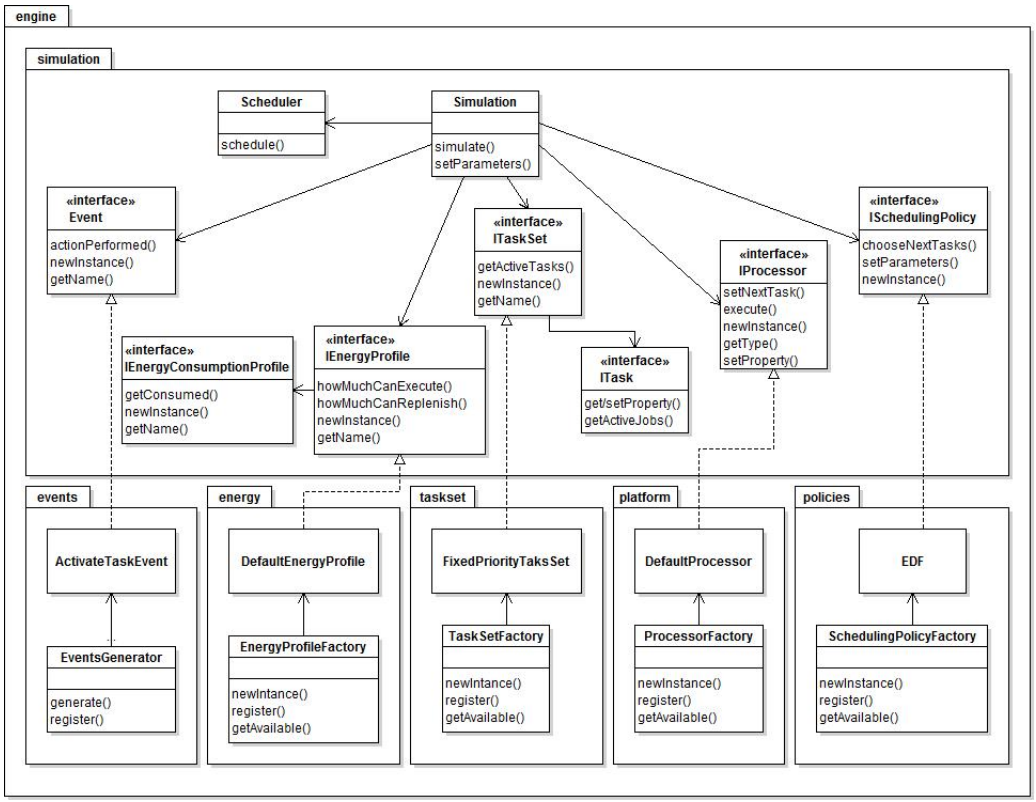


Figure 4: The engine module UML diagram

application.

This module separation follows the classical *Model-View-Controller* (MVC) design pattern (see Figure 3) that allows a secure isolation of the business part of the application from its presentation and thus it allows the engine module to be more generic and easily usable by other applications such as an API. Furthermore, splitting the code into several modules limits the communication between the different modules to specific classes or interfaces by respecting the inversion of control design pattern (IoC). The main advantage of IoC is to prevent inter-dependency between modules and facilitate their maintenance.

5.1 Engine Module

The aim of a real-time simulator is to imitate the execution behavior of systems while respecting the hypotheses. Real time systems are usually composed of a platform (i.e., processors, memories and caches), an energy profile (i.e., supply, consumption, heating, etc), a real-time taskset and a scheduling policy. Many types of these components are studied in research works and are simulated and compared. The architecture of the simulator must take into account the need to integrate new customized components, and then be used for the comparison process using the same simulator.

In *YARTISS*, the concept of each component is defined with an interface that describes its responsibility, input and output as follows:

Taskset:

Firstly, a task is defined mainly by the interface `ITask` that describes, with getter and setter properties, the most common parameters of real-time tasks. It is characterized namely by a deadline, a period, a WCET and energy consumption. Other parameters

can be added based on the desired model by inheriting classes from the basic interface. Also, the `ITask` interface describes the behavior of tasks and their jobs' activation and management. As a result, several task models can be implemented from the same interface with different parameters, such as periodic and sporadic.

Secondly, a taskset is defined in *YARTISS* as a collection of sorted tasks based on the priority assignment. This aims to provide a sorted subset of active tasks at any time to be used in the scheduling. It can be seen as a container of tasks that manages the dependencies between the tasks (as in the case of DAG tasks) and provides a task ordering according to the specifications of the user (fixed job or task priority and dynamic priority). The combination of the inherited classes of tasks and tasksets defines the taskset model to be simulated with its priority assignment. The user can implement a customized behavior of tasks and tasksets by extending these interfaces and add his own properties by using the generic available getters and setters.

The current release of *YARTISS* contains by default the following taskset models: Liu and Layland model (i.e., periodic fixed-priority tasksets), the dependent tasks of the DAG model and the energy-harvesting task model.

Scheduling policy:

the responsibility of a scheduling policy is to merge the partitioning of the task execution on available processors. *YARTISS* defines this behavior with the interface `ISchedulingPolicy`. It has a main method called `chooseNextTasks()` that selects the highest priority tasks at time t, to execute on the processors of the system. This method receives the necessary components for simulation such as the state of the processors, the taskset and the energy profile. The scheduling policy can use external and generic parameters to calibrate its behavior with the method `setParameters()`. A scheduling algorithm is usually adapted to a specific taskset model. For this reason, we allowed the scheduling policies to create an adapted type of taskset model with the method `createTaskset()`. By outsourcing the scheduling decision from the core of the `Simulation` class, we are able to implement several scheduling algorithms for different taskset models, which simplifies the comparison process. We have already implemented around twenty scheduling policies in *YARTISS* for different taskset models, including fixed-priority Rate-Monotonic, Deadline-Monotonic, EDF, LLF, etc. Also, there are algorithms for energy-harvesting systems and DAG taskset model.

Energy profile:

The energy profile of a system includes the profiles of energy production and consumption models. The interface `IEnergyProfile` describes how the system is supplied with energy and how it manages its consumption. It offers a set of methods that demonstrates the energy source (battery for instance), its limits and its energy levels within a time interval. The method `howLongCanExecute()` calculates the maximum time at which a task can execute w.r.t. the available energy and the used consumption profile. This can be used to simulate different energy sources, different rechargeable batteries with different environment energy sources. The second part of the energy profile is the energy consumption model. *YARTISS* proposes two modes, the global mode in which all the tasks have the same energy consumption model, and the non-global mode which allows the tasks to consume energy based on to their own consumption model. According to this, the simulator can be general and specific at the same time. The interface `IEnergyConsumptionProfile` describes how task consumes energy with the method `getConsumed()`. This method gives the amount of energy to consume for a given execution time. Until now, we implemented one energy profile and one energy consumption model which are used to manage energy

for energy-harvesting systems. In other words, a battery replenished continuously with a fixed rate with a global constant consumption. It is possible to use this profile for traditional real-time models that do not include energy constraints, by setting the energy consumption parameter to 0 and the energy production parameter to ∞. Furthermore, other energy profiles can be added easily to *YARTISS* such as the solar energy source or the Gaussian consumption curve.

Execution platform:

A system platform consists mainly of a number of processors on which the tasks execute. The interface `IProcessor` defines the common behavior of processors. A task is allocated to a processor within a time interval and the method `execute()` performs the execution of the tasks within this interval by updating the state of the tasks. The current release of *YARTISS* deals only with one default type of processors which are the identical unit-speed processors. However, it is possible to add new types of processors with different speed and, consequently, the method `execute()` has to be modified according to the new characteristics. An example of the different type of processors is the processors with Dynamic Voltage and Frequency Scaling (DVFS) capabilities.

5.1.1 The Simulation Process

The aim of this tool is to simulate the scheduling of a system according to the parameters and the assumptions of the user, mainly the taskset, the platform, the energy profile and the scheduling policy. According to the scheduler, the scheduling decisions are taken based on specific event, such as the activation and finish time of a job, its deadline and a preemption event. Respectively, a time-triggered simulator will lead to run a slow simulation especially if the used scheduling policy has heavy overheads.

YARTISS is an event-triggered simulator and at each event, the scheduler is awaken to take a scheduling decision. A scheduling decision consists of assigning priorities to the active jobs according to the scheduling policy, and allocate them to processors based on the state of the system (energy state for example). The simulator uses a basic set of events that guaranties a correct execution of the system such as job request, activation and finish time, deadline check, etc. These events are natively implemented in *YARTISS*. The simulation starts generating the events responsible for requesting the first job of each task. Then other events are generated based on the execution behavior of the jobs. For example, when a requested job meets its deadline, the corresponding deadline event will be generated and request event for the next job as well. The events are generated based on a temporal and priority ordering. For each event, the scheduling policy is called to select the highest priority jobs to execute on the available processors. Then, the selected jobs execute while respecting the type of processors and the energy profile. This mechanism is implemented in class `Simulation`.

The interface `Event` describes the role of an event. The current release of *YARTISS* provides the necessary events to schedule a real-time system. However, the user is allowed to add his own events by implementing their customized behavior. To do so, the user should extend the `Event` interface and register the new event with the method `register()` from class `EventGenerator`. Furthermore, to generate an event, the user should use the method `generate()` of class `EventGenerator`. Then the event will be processed with the desired behavior which is implemented by the user.

Metrics & Statistics: The aim of a simulation is not restricted to checking the feasibility of a given system. It is important also to compute some metrics and statistics of the performed simulation in order to analyze its performance. We considered this

functionality in *YARTISS* and we integrated a generic mechanism dedicated to compute statistics. It is possible to compute a metric at the beginning or the end of the simulation, or even during the processing of the events. Also it can be aggregated with other values when several simulations are performed. In the simulator, a metric is represented by the interface `IStatisticCriterion` which has a method for each possible calculation time (i.e., beginning, end, event processing). Several metrics are implemented natively and, as always, the user can add customized metrics to the simulator by implementing the interface. At the beginning of each simulation, an instance of each active metric is created. Then, at each metric calculation time, the corresponding method of the metric is called. At the end of the simulation we can get the final value of each metric using the method `getValue()`. In the case of several simulations, we can also aggregate the values of the same metric on all of the simulations in order to compute the maximum/minimum values and the average value, etc. The aggregation function is implemented by the metric class and it can provide several values: maximum, minimum and average values, percentage value, etc. By doing so, the simulation class computes several statistics independently from their implementation. Among the implemented metrics available in *YARTISS*, we can cite: deadline miss counter, average busy and idle periods, average overheads.

5.1.2 Inversion of Control and Dependency Injection

Based on the above-described components, we notice that the simulator deals only with interfaces and never directly with implemented classes. This allows the simulator to be generic regarding the parts that are varied mostly, such as the scheduling policy, the taskset model, the platform, the energy profile and the statistics metrics. This implantation respects the design pattern Inversion of Control (*IoC*) and the Dependency Injection that stipulates that the objects composing the application must be weakly coupled and dependencies are linked only at run-time.

The *IoC* design pattern is a style of software construction where reusable code controls the execution of problem-specific code. Its main advantage is that the reusable code is developed independently from the problem-specific code, which often results in a single integrated application. The *IoC* design guideline is useful for the following purposes:

- the execution of a a certain task is independent from its implementation,
- every part of the system focuses on its design purpose,
- parts have no assumptions about the constraints and the limitations of the other parts,
- replacing or modifying a part in the system does not affect the rest of the parts.

Sometimes, the IoC is referred to as the "Hollywood Principle: *Don't call us, we'll call you*, because program logic runs against abstractions such as callbacks".

In the case of *YARTISS*, the simulation code is a reusable part and the different implementations of scheduling policies, processors, tasksets or metrics are done on specific parts that can be replaced and developed independently. This is what makes *YARTISS* a generic real-time simulator that can be used easily by other researchers.

5.2 Service Module

The engine module contains all the code necessary to build and run a simulation. Furthermore, it can be used alone or imported as an API to be integrated into another software. However, we were interested in providing a complete simulation tool and not only an API. So we implemented more classes which allow an intermediate user to build and run simulations easily. This is done using the Service module which contains the functionalities described above in Section 4. From the design point of view, the service module represents the Controller and the Model parts of the famous design pattern Model-View-Controller (MVC) or the View-Model part of the Model-View-View-Model (MVVM) design pattern. It is a software brick of higher level that uses the engine module. The service module provides the classes that set the replaceable parts of a simulation (e.g., the scheduling policy, the taskset, the platform, metrics, etc), run the simulation and save the results. It contains mainly three public classes, each represents a certain functionality:

- The `TimeLineViewModel` class provides the necessary parameters to simulate a task and it allows the user to run the simulation and get the results.
- The `BenchmarkViewModel` class allows the user to set quickly the description of a large scale simulation where several scheduling policies are compared using chosen statistic metrics in different scenarios of energy profiles. This class helps the user to build such a simulation without knowing the implementation of all parts. The description is passed using a String object and all created generic objects are done using the Dependency Injection paradigm. This class saves results which are readable by the chart plotter tool *gnuplot*.
- The `TaskSetsGenViewModel` class builds a custom taskset generator in the same way.

These classes are packed in an independent module so as to be built easily using a higher level software with a friendly user interface (such as a web server or a graphical or textual interface).

5.3 Framework Module

This is a toolbox module that contains generic classes and functions that facilitate simulations. It is completely independent from the other modules and it can be reused by other projects easily. This module is considered as a small API that provides a simple and small abstraction of the the Java concurrency API and some generic classes for the MVC design pattern :

Concurrency: this feature allows the user to run several Java runnables simultaneously with the possibility of collecting results. It implements the *producer/consumer* design pattern to get the final results. It aims to accelerate the execution by taking advantage of the concurrency API of Java. Furthermore, running several computations in parallel can be done directly by using Java threads. However, getting and computing the final results of simulation is difficult due to the concurrency of the threads. For this reason, we adopted the *producer/consumer* design pattern so as to the threads perform the required computations and produce the results in parallel. On the other hand, a single consumer is allowed to get and aggregate the results with a *thread safe mode* by using a blocking queue. The service module uses this functionality to run large scale

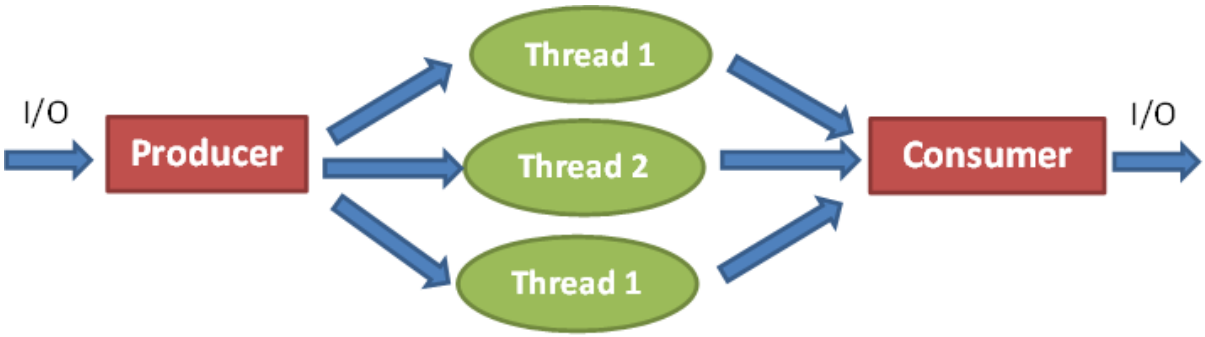


Figure 5: Producer/Consumer design pattern

simulations in parallel to collect aggregated results in order to present them through the user interface. It can also use the taskset generator.

Model-View-Controller (MVC): This part provides a simple framework for building MVC applications. It is not as sophisticated as commercial frameworks, but it is sufficient for the simulator needs. Mainly, it contains classes that use the Java reflection capabilities to handle getter and setter properties. This enhances the generic aspect of the architecture and keeps a weak coupling between objects and modules.

5.4 View Module or GUI

This module is responsible for building and running a graphical user interface that facilitates the use of the different features and functionalities of the simulator that are mentioned in section 4. It represents the highest level of the application and it proposes some independent graphical components that can be used to build a customized graphical interface (components to visualize the time chart of a simulation, to visualize the energy or the battery evolution curve). The views provided by this module are linked to the other models and controllers of the service model only at run-time. This leads to a generic module and the reusability of the proposed code.

6 Case Study

As described above, the main feature of *YARTISS* is its genericness. This means that its functionality can be extended easily by researchers to include new customized modules and scheduling policies to the simulating tool without the need to understand the core of the system.

In this section we demonstrate the generality feature of *YARTISS* by adding a new task model called the Directed Acyclic Graph (DAG). This model is an example of dependent real-time tasks with precedence constraints. Then we describe how to add implemented simulation policies and schedulability tests for DAG tasksets on a system of identical processors. This case study demonstrates various functionalities in *YARTISS* such as the task generator, the addition of new scheduling policy and new scheduling test and the integration of dependent tasks within the GUI of *YARTISS*.

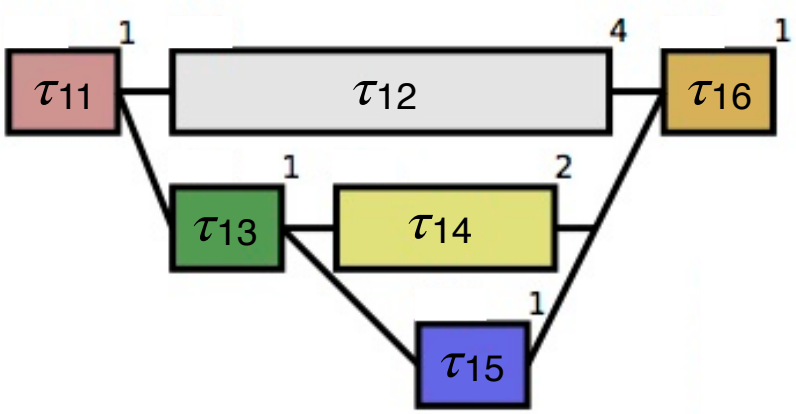


Figure 6: An example of a DAG task consists of six subtasks.

6.1 Task model: Directed Acyclic Graphs

The Directed Acyclic Graph (DAG) is a model from the intra-task parallelism family, in which a real-time task is composed of dependent subtasks that can execute in parallel based on their precedence constraints. A real-time parallel DAG task is characterized by a set of single-threaded subtasks, a deadline and a period. The subtasks of a DAG task are dependent subtasks and their execution flow is represented by the structure of the graph. Each subtask is characterized by its WCET. The relations between the subtasks are directed from a source subtask to a sink task and the structure of the DAG shows the dependencies between the subtasks. A directed relation from subtask u to subtask v identifies the precedence relation between the subtasks. This means that subtask u is a predecessor of subtask v, and subtask v cannot start its execution until subtask u completes its own. For any given two subtasks in a DAG task, there is only one directed relation only. An acyclic graph means that the cycles in the execution flow of subtasks are prohibited, such as subtask u $\rightarrow$ subtask v $\rightarrow$ ... $\rightarrow$ subtask u.

Figure 6 shows an example of a DAG task consists of six subtasks. Each box in the figure represents a subtask and the lines between the subtasks are the directed relations between the subtasks. The execution flow of the subtasks is shown in the figure.

The DAG tasks are an example of intra-task parallelism, in which the parallelism in the task comes from the dependencies between the various threads in the task. It is a general model which can be applied to many other parallel task models such as the multi-threaded segment and the fork-join model.

6.2 Taskset Generator

The DAG task model requires special treatment when it comes to the process of the task generation, because it is a dependent task in which subtasks execute in parallel on the different processors based on their precedence constraints. According to this, the traditional independent sequential task model cannot be used directly to describe the dependent tasks and it requires extra parameters to describe their dependencies.

We can notice that a dependent task has the same temporal parameters as an independent task, such as its WCET, its period and its deadline. However, the difference is in their execution behavior. For a given DAG task, the execution of a subtask is affected by execution behavior of its predecessor subtasks such as their activation time or their

completion time. As a result, we had to create a new class in *YARTISS* to represent dependent tasks which we called the `Subtask` class in the `schedulable` package. This class extends the `PeriodicTask` class in order to use the available functionalities of the default task model provided in *YARTISS*. However, each dependent subtask will have extra parameters which are mainly information about its predecessors (the parent subtasks) and its successors (the children subtasks). Using these two sets, the dependencies between the subtasks of any DAG task are represented by their parents and their children. More functions are implemented in the class in order to manipulate the dependencies and calculate extra parameters for each subtask such as their local offsets and deadlines which are necessary to the scheduling of DAG tasks as described in [17]. Another class called `GraphTask` had to be added for the DAG model to contain the global information about each DAG task, such as the total execution time of a DAG, its utilization, its density and the critical path calculations.

In order to launch a simulation process or to perform large-scale tests for the DAG tasks, it is necessary to provide a number of tasksets generated with specific characteristics, such as a specific utilization and deadline constraints. This is implemented in *YARTISS* for the DAG tasks using two methods:

- a new generator class is added in *YARTISS* to implement a random generator for the DAG tasks which is called `GraphTaskGenerator` class. The generator extends the default task generator which uses the UUniFast-Discard algorithm to split the system's utilization among a specific number of DAG tasks. Extra functions are added to the graph generator so as to randomly create the subtasks, assign them a WCET and generate their directed relations. These functions guarantee a random structure of the DAGs and their acyclic relations. The result of the class is a set of DAG tasksets of generic type inherited from ITask, which can be used directly in the simulation process or can be saved into an XML file using the `XMLTaskSetWriter` class,

- XML source file is passed to the `TasksetFactory` factory class. This approach permits the use of the same DAG tasksets, contained within the XML file, for multiple simulations and scheduling tests. However, the integration of the DAG tasks (or dependent tasks in general) in this method requires modifying the `SchedulableFactory` class in *YARTISS* so as to handle dependencies. The `XMLTaskSetReader` class parses the XML file containing the DAG tasksets and the simulation parameters into ITask objects which can be used by *YARTISS* directly. In this file, each task is denoted as a collection of subtasks and their dependencies are stored in the parents and children attributes of the subtask element. An example of an XML file containing a DAG taskset is shown in Listing 2.

Listing 2: An example of an XML file for describing Directed Acyclic Graph Tasks

```
1 <?xml version="1.0" encoding="UTF-8"?>
2   ...
3   <tasks nbTasks="1" type="Fixed Priority">
4     <task deadline="10" firstRelease="0" nbSubtasks="2" period="10"
          priority="1" type="graph" wcee="0" wcet="5">
5       <subtask children="1" deadline="10" firstRelease="0" index="0"
            localDeadline="7" nbProc="1" parents="" period="10" priority="1"
            type="subtask" wcet="2"/>
```

```
6       <subtask children="-1" deadline="10" firstRelease="2" index="1"
            localDeadline="10" nbProc="1" parents="0" period="10" priority="1
            " type="subtask" wcet="3"/>
7     </task>
8   </tasks>
```

6.3 Adding a new Scheduling Policy

The scheduling policies for the DAG tasks share mainly the same features as the default independent tasks. The dependencies between the subtasks require to be managed and considered by the scheduling policy, since a subtask job is activated dynamically when all of its predecessor subtasks complete their execution in the interval between the activation and the deadline of its DAG task. For the DAG model, we implemented two scheduling policies: The Least Laxity First (LLF) from the dynamic priority scheduling and the Global Earliest Deadline First (GEDF) from the fixed job priority scheduling.

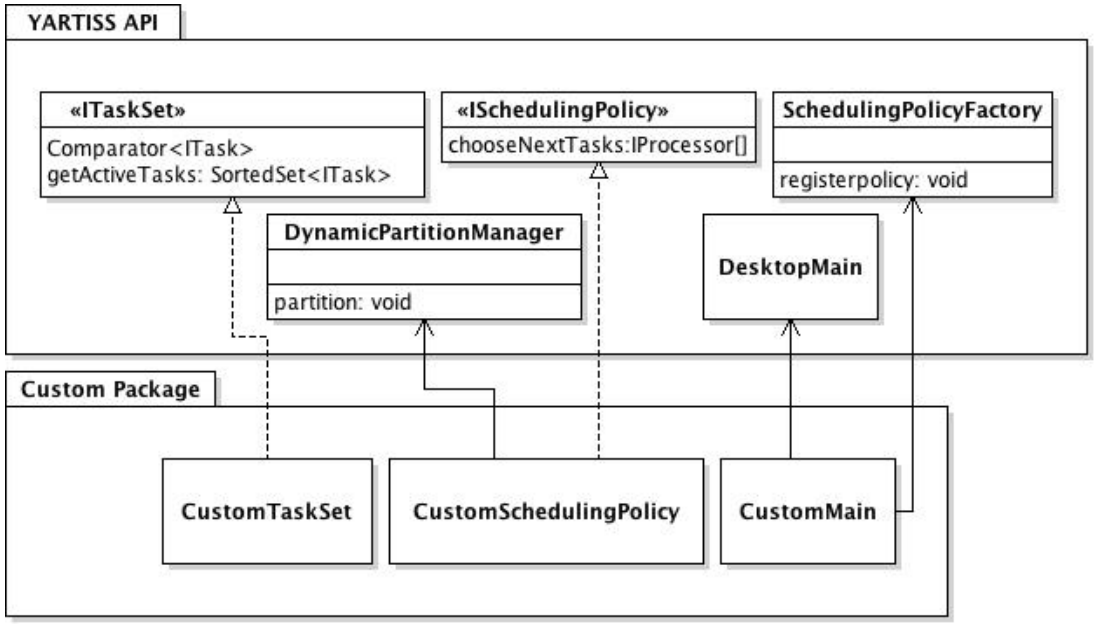


Figure 7: A UML diagram describes the addition of a new scheduling policy.

The addition of new scheduling policies to *YARTISS* follows the UML scheme described in Figure 7. As described earlier, the scheduling policy depends mainly on the corresponding taskset class. We created a taskset class for each scheduling policy which contains the necessary functions to sort the tasksets based on the priority assignment required by the policy. In the case of the GEDF scheduling policy, we added a new class called the `GlobalEDFTaskset` which extends the `ITaskSet` interface. The functions necessary for assigning priorities have to be implemented such as the `Comparator<ITask>` function and the `getActiveTasks()` function, which will assign higher priorities to the active jobs with the earliest absolute deadlines.

Then, another class for the scheduling policy called `GlobalEDFGraphPolicy` is added, which extends the `ISchedulingPolicy` interface. In this class, the main function to be implemented is the `chooseNextTask()` function and the sorted taskset from the corresponding `TaskSet` class is its input. Its main functionality is to choose the m highest-priority active jobs to be executed in the system, where m is the number of processors.

Then the chosen jobs are assigned to processors with a desired scheduling characteristic, such as global or partitioned scheduling (according to the authorization of migration and preemptions between jobs). This is done using a partition manager which is by default a global scheduler. The newly-created scheduling policy should be registered in the simulator by the use of the `register()` function from the `SchedulingPolicyFactory` class. The call of this function should be done in the customized main class created by the user or by using the GUI of *YARTISS*.

Finally, the GUI of *YARTISS* supports the dependent tasks in general and the DAG tasks in particular. The simulation process and the scheduling of DAG taskset is shown within a specific simulation interval based on the chosen scheduling policy. The simulation can be launched by using the `DesktopMain` class which is the entry class of *YARTISS* and the DAG tasksets are passed through an XML file. The same process can be applied on LLF scheduling policy which is described in Listing 3.

Listing 3: How to add a scheduling policy

```
public class MainDemoSP {
  public static void main(String[] args) {
    SchedulingPolicyFactory.registerPolicy(new LLF());
    DesktopMain main  = new DesktopMain();
    main.setVisible(true);
  }
}

class LLF extends
  AbstractMultiProcSchedulingPolicy{
  @Override
  public String getPolicyName() {
    return "LLF";
  }
  @Override
  public ITaskSet createTaskSet() {
    return new AbstractTaskSet(new Comparator<ITask>() {
      @Override  public int compare(ITask t1, ITask t2) {
        long laxity1  = t1.getDeadline() - t1.getRemainingCost();
        long laxity2 = t2.getDeadline() - t2.getRemainingCost();
        int cmp =  (int) (laxity1 - laxity2);
        if(cmp==0)
          return (int) (t1.getPriority() - t2.getPriority());
        return cmp;
      }
    }) {

  @Override
  public SortedSet<ITask> getActiveTasks(long date) {
        SortedSet<ITask> activeTasks = new TreeSet<ITask>(comparator);
        for (ITask t : this)
          if (t.isActive())
            activeTasks.add(t);
        return activeTasks;
      }
    };
  }

  @Override
  public Processor[] chooseNextTasks(
      Processor[] processors,  ITaskSet taskSet,
```

```
42        IEnergyProfile energyProfile, long date,
43        EventGenerator evGen) {
44      int i=0;
45      for (ITask task : taskSet.getActiveTasks(date)) {
46        if(i<processors.length){
47          long hlcet = energyProfile.howLongCanExecute(task);
48          if (hlcet <= 0) {
49            evGen.generateEvent("energy_failure", task, date, null);
50            processors[i].setNextTask(null);
51          }
52          else {
53            evGen.generateEvent("check_energy_state", task, date + 1, null);
54            processors[i].setNextTask(task);
55          }
56        }
57        i++;
58      }
59      for(;i<processors.length; i++){
60        processors[i].setNextTask(null);
61      }
62      return processors;
63    }
64
65    @Override
66    public ISchedulingPolicy newInstance() {
67      return new LLF();
68    }
69  }
```

6.4 Adding a new schedulability test

Usually, a schedulability test of a task model consists of a scheduling condition that determines the schedulability of the given taskset. The test can be either exact, sufficient or necessary based on its schedulability certitude level. Adding a new scheduling test to *YARTISS* does not require simulating the scheduling of the tasks in the systems and the results of the test are usually a text file or a curve that shows whether each taskset is schedulable or not. The scheduling tests can be used as an indication of the performance of different algorithms and also for the comparison process between them.

In the case of DAG tasks, we implemented a number of scheduling tests to compare the performance of our own approaches with other tests proposed in the literature of real-time systems. For each test, we create a new class for the desired test that extends the `IFeasibilityTest` interface from the `simulation` package in *YARTISS*. The test class should implement two main functions; `isFeasible(...)` ,which contains the test functionalilty and it returns a boolean object as a result, and `checkFeasiblity(...)` which returns a `FeasibilityResult` enum object as a result. Both functions can be applied either of a task or a taskset level.

Multiple schedulability tests use usually the same tasksets as entries in order for the results to be reliable. Hence, the input DAG tasksets are passed using either the DAG generator directly or mainly by using the dataset files which contain the parameters of the DAG tasksets (either using a text or an xml file). There are pre-coded parser classes written in *YARTISS* for parsing text and xml files. However, the format of the input files should be adapted to the parsers as in the XML example shown in Listings 1 and 2. The

output results are saved to text files using a writer class already available in *YARTISS*.

6.5 Adding an Energy Profile

The same methodology can be applied if we want to add a new energy consumption profile to *YARTISS*. Listing 4 shows the Java code needed to use an alogarithmic consumption profile as an external module.

Listing 4: How to add a new energy profile

```
1  public class MainDemo {
2    public static void main(String[] args) {
3      SchedulingPolicyFactory.registerPolicy(new LLF());
4      ConsumptionProfileFactory.registerConsumptionProfile(new LogConsumption
           ());
5      DesktopMain main  = new DesktopMain();
6      main.setVisible(true);
7    }
8  }
9
10 class LogConsumption implements
11   IEnergyConsumptionProfile {
12   @Override
13   public String getName() {return "log";}
14   @Override
15   public List<Double> getParameters() {return null;}
16   @Override
17   public void setParameters(List<Double> params) {}
18
19   @Override
20   public long getConsumed(long wcet, long wcee,
21           long remainingTimeCost,  long duration) {
22     double a = wcet - romainingTimeCost;
23     double b = a + duration;
24     if(b > wcet) b = wcet;
25     if( (b-a) <= 0 )return 0;
26     long result = (long) Math.log(b/a);
27     if(result > wcee )result = wcee;
28     return result;
29   }
30
31   @Override
32   public IEnergyConsumptionProfile cloneProfile() {
33     return new LogConsumption();
34   }
35 }
```

7 Distribution

The project is available from the GForge collaborative development environment hosted at `https://svnigm.univ-mlv.fr/projects/YARTISS/`. This environment provides a subversion repository allowing anonymous checkouts, documentation hosting, RSS feeds subscriptions, and public forums. A web page dedicated to *YARTISS* is also available at `http://YARTISS.univ-mlv.fr`. In addition to a general presentation of the tool, it pro-

poses a demo applet version which allows interested readers to try *YARTISS* directly from their web browser and an application form to allow anybody to share external modules.

8 Future Works

The actual release offers many important and expandable features but the simulator is still under development. Some parts of the project have been made in a hurry which has prevented them to be as clean as they could. Improvements are planned to address such weaknesses, like we have done with energy profiles and scheduling policies. Some other future work are planned:

1. *YARTISS* is also a teaching tool used in our university, so in order to open it to other real-time systems problematics, we plan to implement resource sharing protocols such as memory, caches and processor synchronization issues.

2. the use of XML format mostly in all inputs and outputs in order to be able to reuse other external tool functionalities, this feature must be generalized to the simulation results in order to allow their visualization with an external tools (*e.g.* GRASP[8]),

3. an additional work is needed to implement other kinds of platforms, like processors with Dynamic Voltage and Frequency Scaling (DVFS) capabilities, in order to be compliant with most recent works in this research area.

4. we want to provide a command line user interface to allow the use of our simulator without the graphical environment to permit its use inside automated scripts and/or through a distant machine. This should be done easily because of the adopted architecture and responsibilities separation,

5. finally, concerning the graphical part, we study the possibility to develop a generic graphical component based on XML that can be able to show and edit all taskset models

9 Conclusion

In this report we presented *YARTISS*, a real-time multiprocessor scheduling simulator. A consequent effort has been made to make it as extensible as possible. To justify the need for an open and generic tool, we presented the history of *YARTISS* development. Then we briefly presented existing simulation tools. We have described the three main functionalities of *YARTISS*: 1) simulate a taskset on one or several processors while monitoring the system energy consumption, 2) concurrently simulate a large number of tasksets and present the results in a user friendly way that permits us to isolate interesting cases, and 3) randomly generate a large number of tasksets. Then, in order to demonstrate the modularity and extensibility of our tool, we presented its architecture and a case study that shows how to add functionalities, in most cases without the need to modify the project archive. Finally we gave the instructions to test *YARTISS* and presented some improvement features we will implement. We hope that this software can become a first step toward a widely adopted simulation tool through the real-time scheduling community.

References

[1] E. Bini and G. C. Buttazzo. Measuring the Performance of Schedulability Tests. *Real-Time Systems*, 30(1-2):129–154, May 2005.

[2] J. M. Calandrino, H. Leontyev, A. Block, U. C. Devi, and J. H. Anderson. Litmus-rt: A testbed for empirically comparing real-time multiprocessor schedulers. In *27th IEEE International Real-Time Systems Symposium*.

[3] M. Chetto, D. Masson, and S. Midonnet. Fixed priority Scheduling strategies for Ambient Energy-Harvesting embedded systems. In *The International Conference on Green Computing and Communications*, pages 50–55, 2011.

[4] P. Courbin and L. George. FORTAS: Framework fOr Real-Time Analysis and Simulation. In *2nd International Workshop on Analysis Tools and Methodologies for Embedded and Real-time Systems*.

[5] F. Fauberteau. RTMSIM. `http://rtmsim.triaxx.org/`.

[6] F. Fauberteau, S. Midonnet, and L. George. Laxity-Based Restricted-Migration Scheduling.

[7] M. G. Harbour, J. J. G. García, J. C. P. Gutiérrez, and J. M. D. Moyano. MAST: Modeling and analysis suite for real time applications. In *13th Euromicro Conference on Real-Time Systems*.

[8] M. Holenderski, M. v. d. Heuvel, R. Bril, and J. Lukkien. Grasp: Tracing, visualizing and measuring the behavior of real-time systems. In *1st International Workshop on Analysis Tools and Methodologies for Embedded and Real-time Systems*.

[9] R. Jayaseelan, T. Mitra, and X. Li. Estimating the Worst-Case Energy Consumption of Embedded Software.

[10] S. K. Kato, R. R. Rajkumar, and Y. Ishikawa. A Loadable Real-Time Scheduler Suite for Multicore Platforms. Technical report, 2009.

[11] G. Koren and D. Shasha. D^{over}: An Optimal On-Line Scheduling Algorithm for Overloaded Uniprocessor Real-Time Systems.

[12] C. Macq and J. Goossens. Limitation of the hyper-period in real-time periodic task set generation. In *9th International Conference on RTS Embedded System*, 2001.

[13] D. Masson. RTSS v1 and v2. `https://svnigm.univ-mlv.fr/projects/rtsimulator/`.

[14] D. Masson and S. Midonnet. Userland Approximate Slack Stealer with Low Time Complexity.

[15] D. Masson and S. Midonnet. Handling non-periodic events in real-time java systems. In M. T. Higuera-Toledano and A. J. Wellings, editors, *Distributed, Embedded and Real-time Java Systems*, pages 45–77. Springer US, 2012.

[16] F. S. McCormick, John W. and J. Hugues. *Building Parallel, Embedded, and Real-Time Applications with Ada*. Cambridge University Press, 2011.

[17] M. Qamhieh, F. Fauberteau, L. George, and S. Midonnet. Global EDF Scheduling of Directed Acyclic Graphs on Multiprocessor Systems. In *21st International Conference on Real-Time Networks and Systems*, pages 287–297, 2013.

[18] M. Qamhieh, S. Midonnet, and L. George.

[19] F. Singhoff, J. Legrand, L. Nana, and L. Marcé. Cheddar: a Flexible Real Time Scheduling Framework. In *The Special Interest Group on Ada*.

[20] F. Singhoff, A. Plantec, P. Dissaux, and J. Legrand. Investigating the usability of real-time scheduling theory with the cheddar project. *Real-Time Systems*, 43(3):259–295, 2009.

[21] R. Urunuela, A.-M. Déplanche, and Y. Trinquet. STORM: a Simulation Tool for Real-time Multiprocessor Scheduling Evaluation. *Workshop of GDR SOC SIP*, page 1, 2009.