# ARTISST: An extensible framework for the simulation of real-time systems

David Decotigny, Isabelle Puaut
{ddecotig,puaut}@irisa.fr
Tel: 02 99 84 {25 13, 73 10}
Fax: 02 99 84 25 29
IRISA, Campus de Beaulieu, 35042 Rennes Cédex, France

**Abstract**

This paper presents ARTISST (**A**RTISST is a **R**eal-**Ti**me **S**ystem **S**imulation **T**ool), an event-driven simulation framework for real-time systems. Contrary to most existing real-time systems simulators, it allows to simulate complex systems made of tasks performing arbitrary computations and exhibiting a complex and realistic pattern for their arrival law, synchronization relations, and execution time. The simulator actually focuses on a time-accurate simulation, for it allows, among other things, to take the operating system (including its scheduler) costs into account. Furthermore, thanks to its modular and extensible architecture, the simulator is not dedicated to a particular Operating System API, but is fully customizable instead.

**Keywords:** Real-time, Operating System, Scheduler, Discrete events simulator, extensibility.

# 1 Introduction

## 1.1 Context

This work considers *real-time* computing systems, in which the correctness of the computations not only depends upon the logical correctness of the computation but also upon the time at which the result is produced[1].

For the *hard* real-time systems in which missing a deadline leads to catastrophic consequences, the most important kind of evaluation of the system is the *feasibility test*. It consists in establishing, before any actual execution, whether all the time constraints will be met or not when the system operates. This assumes that the system designer did provide at specification time the necessary material for characterizing the timing properties of both the system activities and its environment. This implies a rather static design of the system and a static modeling of its environment.

Another (complementary) approach for evaluating a real-time system consists in comparing the behaviors of multiple implementations of the same application functionalities on various operating systems (especially schedulers) according to a set of metrics, such as for instance the guarantee ratio and the release jitter. This kind of evaluation, be it conducted on real or simulated executions of the system, allows for the evaluation of time constrained systems that may be less statically determined, and which may evolve in a less statically characterized environment.

In this paper, we focus on the latter kind of evaluation. This implies that we address real-time systems for which either the environment or the system may not be fully characterized in the time domain (*eg.* evolves dynamically), thus making it difficult if not impossible to be statically analyzed.

## 1.2 Motivations for the design of a real-time simulation framework

In this context, where all the temporal behaviors are not fully characterized, evaluating the system during a real or simulated execution is expected to be both more achievable and less pessimistic than a static analysis of the system, albeit less safe.

However, this kind of evaluation implies to be able to gather sufficient runtime information regarding the system being studied. Since the information of interest for some targeted metrics (release jitter, slack time, ...) is related to time, it is very sensitive to instrumentation, which can bias the whole system timing behavior (*probe effect*). Whereas real execution is *de facto* subject to such probe effects, simulated execution is immune to them, since the instrumentation routines are part of the simulator rather than part of the simulated system. This pleads in favor of the evaluation through simulated execution.

An other advantage for evaluating a real-time system through simulated execution is the *playback* feature, which consists in studying the system behavior under the exact same execution conditions, with successively varying internal characteristics (scheduler's costs, scheduling policy...).

Because our primary goal was to evaluate many schedulers, these are the two reasons why we moved towards the evaluation through simulated execution. However, resorting to evaluation through simulated execution is of interest only if it is realistic. This means that both the whole system and its environment should be modeled as close as possible to their corresponding real counterparts. Considering that the existing simulators for real-time systems we know don't fulfill these requirements, we decided to implement our own generic, customizable and extensible real-time systems simulation framework: ARTISST.

## 1.3 ARTISST characteristics

ARTISST (for **A**RTISST is a **R**eal-**Ti**me **S**ystem **S**imulation **T**ool) is a modular framework dedicated to the accurate simulation of real-time systems. The figure 1 presents the various entities that ARTISST is able to simulate: an *application* executing on top of a *real-time operating system* (RTOS), interacting with an external *environment* by way of *events*.

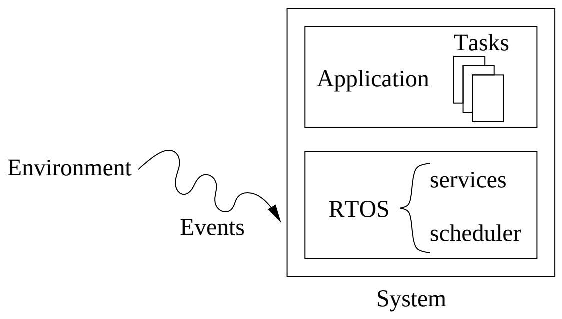


Figure 1: Model of the simulated world

The primary goal of ARTISST is to provide an architecture and a set of tools for evaluating the behavior of real-time systems:

1. Made of a set of tasks making user-defined computations characterized by an arbitrarily complex pattern for their arrival law (not only synthetic workload), execution time and synchronization constraints;

2. Constructed on top of user-defined RTOS services and scheduler;

3. Interacting with an arbitrary and possibly dynamically changing environment;

4. Managing the simulated system according to a global simulated real-time scale;

5. Distinguishing between global real and (systemwide) system time clocks, thus allowing for the simulation of distributed systems subject to clock desynchronization for example;

6. Taking the simulated RTOS costs (including those of the scheduler) into account;

7. Generating instrumentation traces according to the simulated real-time scale suited to post-simulation evaluation of the system.

Among these properties, the main contributions of ARTISST are *i)* its ability to be tailored to a large panel of RTOS schedulers and services; *ii)* its ability to take the RTOS costs into account; *iii)* the fact that any operation exhibiting a complex CPU simulated-time consumption pattern can be simulated. This is all achieved thanks to the use of a highly modular and extensible object-oriented framework, which, as a side effect, favors software reuse.

Example utilizations of ARTISST can be: comparing the performance of various schedulers managing the same application under the same environment influence; or simulating distributed systems made of multiple system simulators connected through a simulated network, in the presence of system clock drift and network delays/faults.

### 1.4 Paper organization

This paper is organized as follows. In section 2, we present the main features of the ARTISST framework. In section 3, we present the extensible task model that the ARTISST framework is able to simulate. In section 4, we detail the architecture of the ARTISST framework, and discuss its extensibility. In section 5, we present the system simulator core engine, responsible for the simulation of the studied system. Section 6 discusses the related works in the field of simulation for evaluating real-time systems. Finally, section 7 gives our conclusions and discusses future work.

## 2 Overview of ARTISST

ARTISST exhibits a set of characteristics (listed hereafter) that enable the simulated system to have a behavior as accurate and as close as possible to a real-world system, even closer than with a synthetic workload based simulation, and which can take the RTOS costs into account.

### 2.1 Extensibility of the real-time system simulator

A simulated system in ARTISST is made of tasks and the underlying RTOS. The tasks are associated with a task model informing the underlying RTOS about their timing characteristics. One strong property of ARTISST is that it constrains the simulated system neither regarding the scheduler, nor the RTOS services it offers, nor the task model it depends on. For that purpose, the ARTISST approach for the design of the simulated system relies on an object-oriented customizable and extensible task model, RTOS services API, and scheduler API.

### 2.2 Support for fine-grained CPU consumption simulation in both the application and the RTOS

In order to allow complex systems to be simulated, we chose to impose the least restrictions on the programming language for the simulated tasks' computations and the RTOS services. Actually, the tasks and the RTOS can be written in plain C, or in any other language defining the appropriate wrappers to the simulator API (C++ available).

Furthermore, the system simulator comes with a central simulation primitive: `hold_cpu(delay)`. It is responsible for simulating CPU occupation for the given `delay` duration. One important feature of ARTISST is that this primitive may be embedded anywhere in the application code or even in the simulated RTOS code.

### 2.3 Modularity

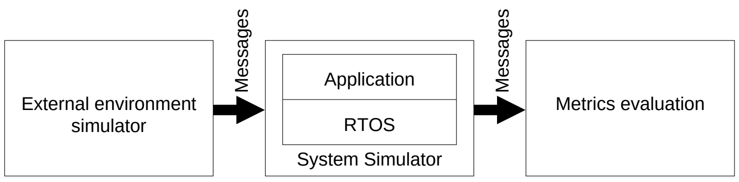


Figure 2: Basic architecture for centralized systems evaluation

Actually, as shown on figure 2, the system simulator only constitutes a (big) part of the ARTISST framework. More precisely, an ARTISST simulator is made of an user-defined interconnection of *modules* “plugged” into one another, the system simulator being one of these modules.

The interconnection of modules takes the form of *messages* exchanges. These messages are essentially related to system instrumentation: they reflect the occurrence of simulated hardware interruptions, the beginning/completion of processing for these interruptions, and the creation/preemption/completion of tasks’ execution. These messages, together with the timestamp according to the simulated real-time that characterizes them, form the basis of the fine-grained instrumentation of the system behavior.

ARTISST comes with a set of default *input* modules that enable to deliver messages for simulating the external environment according to probabilistic distributions, or according to a previously generated message trace. The designer is encouraged to compose multiple such modules, or even to connect his own modules for modeling the environment.

Given these *input* messages, the *system simulator module* in ARTISST delivers them to the simulated RTOS, and is in charge of generating the *output* messages related to the RTOS and application in reaction to them. The designer is encouraged to customize the application or the simulated RTOS that form this module, and even to generate his own instrumentation messages.

Finally, ARTISST comes with a set of default *output* modules that are in charge of processing the instrumentation messages generated by the system simulator, using either statistical analysis or graphical representation, or through trace recording. The designer is encouraged to compose multiple such modules, or even to connect his own modules for evaluating his own metrics.

## 3 Task Model & Task Execution

The system simulator in ARTISST identifies a simulated *task* in the system as being a sequential flow of control which is executing on the simulated CPU, or which is waiting for execution on the CPU (*ie* preempted or blocked). Such a task is an instance of a *stream of task executions*, which is characterized off-line by the *task model*. On-line, each task is associated with its on-line *status* informations managed by the simulated RTOS.

The figure 3 gives an illustration for both the task model and the task status: the shaded areas figure a single stream of task executions, made of multiple task instances being preempted by other tasks (not shown). The task model characterizing the stream gives some information about the arrival law and the execution time of the task instances. Each task instance is contained in a dotted box on the figure, and its execution is associated with its execution status.

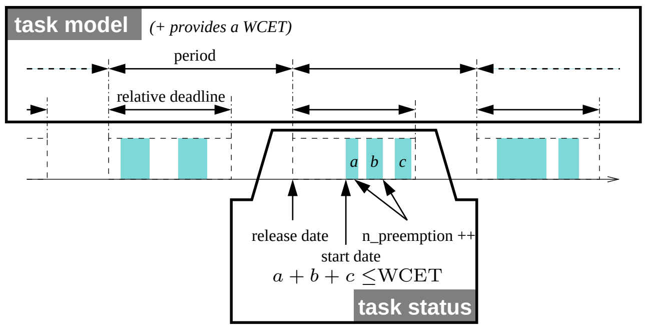


Figure 3: Task model & status example for a periodic task preempted by other tasks (not shown) in the system

In the following, we present the basic task model (3.1) and status (3.2). We show why and how they can be extended (3.3). Then we give details on what kind of computation each task is able to make (3.4).

## 3.1 Basic Task Model

A task model in ARTISST is a data structure that defines the properties for a stream of tasks' executions.

Since a wide range of schedulers use a recurrent set of attributes, we chose to gather these common attributes in the *basic task model*, which may be extended if not suited to the user's needs for the simulation. This basic task model contains:

- Tasks' operational definition

  **The maximum number of concurrent executing tasks.** It specifies how many tasks associated with a task model may be concurrently executing in the system simulator.

  **The entry point for the task.** It is a reference to the function that identifies where a task associated with the task model has to start its execution.

- Tasks' temporal behavior qualification

  **The temporal characteristics of the tasks' executions.** It defines the execution times (worst and/or average and/or best) and the relative deadline of each task belonging to the task model. Some or all of these attributes may be undefined.

**The arrival law of the tasks.** It qualifies the way tasks are activated on the system: periodically, sporadically, aperiodically. The first two arrival laws are associated with a parameter that describes it: the period and the minimum inter-arrival time respectively.

For example, on the figure 3, the task model would specify that the tasks are periodic with a specified period, that each task has a specified deadline relative to its arrival date, and that each task's execution is characterized by a specified worst case execution time (*WCET*).

## 3.2 Basic Task Execution Status

While the task model defines the abstract task attributes common to all task executions of a stream of tasks' executions, a *task* is a concrete instance of such a task model. A basic *task execution status* (or shortly: *task status*) is associated with each simulated task. It is a data structure that may be extended by the designer if it doesn't suit his needs, and that is made of:

**A thread of execution.** It maps to a stack and a program counter on the host for the simulation.

**The task model.** It is the reference to the task model the task conforms to.

**Timing status for the task instance.** It consists in the *arrival date* for the task, and in the *CPU time* the task consumed up to now, according to the real and system time scales (defined in 4.2 and 5.3.4 respectively).

**Performance monitoring.** It consists in counters for events that are occurring while the task is executing: number of preemptions due to resource synchronization, number of (simulated) hardware interruptions.

All this information is managed by the core of the system simulator, and is made available to any entity in the system simulator, such as the scheduler, the extensions to the basic OS, the interrupt handlers, or even the simulated tasks.

## 3.3 Customization & extensibility

Both the task model and the task execution status are essentially aimed at servicing the simulated OS and the scheduler. The designer of the simulated RTOS is not urged to use any of the default basic attributes previously enumerated: he can simply ignore them; this wouldn't cause any trouble with the system simulator module.

Moreover, those two basic data structures can be extended if they don't fit the needs of a special OS API or scheduler: the designer can add to them the new attributes that are needed by the OS extensions or the scheduler she/he is implementing. This can be easily achieved using classical object-oriented inheritance.

## 3.4 Task execution

Once a task model is defined and serves as the description for a stream of tasks' executions, and once the attributes for describing the status of each task of this stream are defined, the tasks may actually be simulated.

### 3.4.1 Task computations

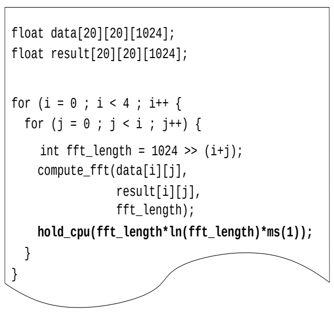
Starting from the entry point defined in the task's model, the computations the task makes can use all the facilities and constructs of the programming language (C, C++, or whatever language that can define wrappers to the simulator API). For instance, a task may choose to call any function in any library on the simulation host, or spawn other ARTISST tasks.

### 3.4.2 CPU occupation simulation: `hold_cpu()`

Any computation done inside the task, no matter how complex it is, actually takes absolutely no time with respect to the simulated real-time. So, there is a need for a facility in charge of simulating time advance.

In ARTISST, this is achieved by calling the `hold_cpu()` function which is part of the ARTISST API. Its purpose is to simulate CPU occupation inside a task.

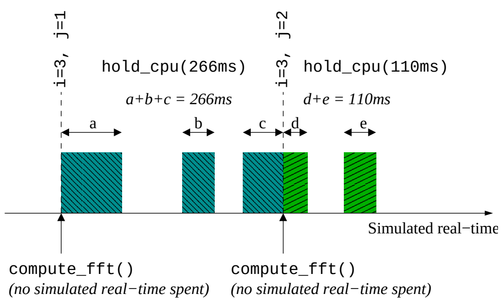
The fundamental property of an `hold_cpu(delay)` call is that its effective duration with respect to the simulated real-time exactly equals `delay`, **independently of any preemption** due to resource synchronization or interrupt handling. The figure 4 illustrates this property: the computation of the deepest iteration when $i = 3$ and $j = 2$ really consumes 266ms on CPU with respect to the simulated real-time, though it is being preempted 2 times; *ie* calling `hold_cpu(266ms)` ensures that $a + b + c = 266$ms.



```
float data[20][20][1024];
float result[20][20][1024];

for (i = 0 ; i < 4 ; i++ {
  for (j = 0 ; j < i ; j++) {
    int fft_length = 1024 >> (i+j);
    compute_fft(data[i][j],
                result[i][j],
                fft_length);
    hold_cpu(fft_length*ln(fft_length)*ms(1));
  }
}
```

(a) Sample code for a task



(b) CPU time consumption with respect to the simulated real-time in the presence of preemptions

Figure 4: hold_cpu(): CPU time consumption simulation

The programmer may add as many `hold_cpu()` as he wants inside his code, independently of the external environment influence (through the hardware interrupts), and independently of any other task that may preempt it in the system (through resource synchronization). This allows to make fine-grained simulations, since one can for example embed `hold_cpu()` calls inside any program loop (as the sample code in figure 4(a) shows).

Furthermore, as we shall see in section 5.6, in addition to being available from inside the tasks of the simulated application, such `hold_cpu()` function calls may also be embedded into the simulated OS and scheduler code, resulting in the fine-grained simulation of the costs induced by the simulated RTOS.

# 4 Overall software architecture

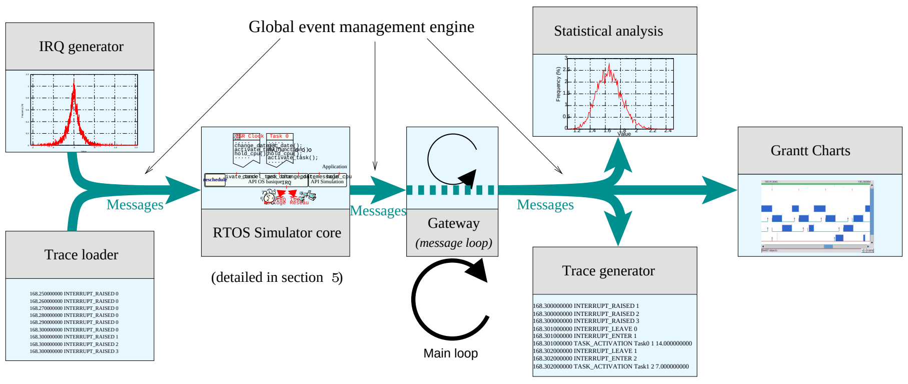
Here, we describe the way the overall ARTISST simulation framework is structured: we define the notions of modules and messages (4.1), present the way the simulated real-time is managed (4.2), and give some details on the event management engine (4.3). We then give an overview of the default ARTISST modules available (4.4).

## 4.1 ARTISST messages and modules

Being a discrete event simulation framework, ARTISST is based on event management, and relies on the concept of event propagation between *modules* that generate, process and react to them. This leads to a modular architecture close to those found in many other discrete event simulators (such as ADEVS[2]).

### 4.1.1 Modules

A module is an object in charge of reacting, generating or propagating the simulation events.



```
168.250000000 INTERRUPT_RAISED 0
168.260000000 INTERRUPT_RAISED 0
168.270000000 INTERRUPT_RAISED 0
168.280000000 INTERRUPT_RAISED 0
168.290000000 INTERRUPT_RAISED 0
168.300000000 INTERRUPT_RAISED 0
168.300000000 INTERRUPT_RAISED 1
168.300000000 INTERRUPT_RAISED 2
168.300000000 INTERRUPT_RAISED 3
```

```
168.300000000 INTERRUPT_RAISED 1
168.300000000 INTERRUPT_RAISED 2
168.300000000 INTERRUPT_RAISED 3
168.301000000 INTERRUPT_LEAVE 0
168.301000000 INTERRUPT_ENTER 1
168.301000000 TASK_ACTIVATION Task0 1 14.000000000
168.302000000 INTERRUPT_LEAVE 1
168.302000000 INTERRUPT_ENTER 2
168.302000000 TASK_ACTIVATION Task1 2 7.000000000
```

Figure 5: Modular structure in ARTISST

The modules in ARTISST are plugged onto one another: the designer is responsible for building such a *circuit* of modules. A simulation circuit may look like the one in figure 5. The ARTISST event management engine is in charge of the propagation of the events from and to the modules according to this circuit. This event management engine is the only component of the overall architecture that the ARTISST user cannot change. In contrast, the circuit is user-defined and the modules may be interchanged or extended.

During the simulation, the events travel from producer module(s) to consumer module(s). Many modules may be connected as input (producer) modules to any given module. Likewise, many modules may be connected as output (consumer) modules for any given module.

Each module is able to generate output events resulting from some internal processing in reaction to input events. Each module is also allowed to mask some of their incoming events, *i.e.* not to forward those events to their output module(s).

ARTISST adopts the object-orientation paradigm, which allows to rely on a unique base class for defining a common interface for all the modules. This is essential for building a generic event management engine, and it allows the designer to extend this base class in order to add new functionalities to the modules.

The base class actually defines the methods needed for a module to be integrated into a simulation circuit. It also defines the method used to extract events from a module (the module is then said to be a *pull* module), and the method used to send events to the module (the module is then said to be a *push* module). For example, on the figure 5, the 3 modules on the left are used as *pull* modules, and the 3 modules on the right are used as *push* modules, the `gateway` module being an "event pump" used as a *pull* module inside the central event loop for the simulation.

### 4.1.2 Events

The *events* are the informations that are generated, processed, and propagated by and between the modules. An event is fully characterized by *i)* its *type* tag indicating what kind of information it provides; *ii)* a *data* field particular to each type of event providing this information (for instance: the task identifier for a task being suspended, ...); *iii)* a timestamp according to the simulated real-time scale (see 4.2). The events look like:

```
<date> INTERRUPT_RAISED <num_of_irq>
<date> SUSPEND_TASK_EXECUTION <id_of_task>
```

ARTISST defines a set of basic event types that may be extended by the programmer. These basic events are partitioned into 4 classes: simulated interrupts management (interrupt raised/enter/leave), task management (task creation/preemption/deletion), system state changes (kernel/user mode), and user-defined instrumentation messages.

Since the simulation events not only drive the simulation process, but also can provide instrumentation informations, the events are also named "messages" in ARTISST.

## 4.2 Events and real-time in ARTISST

Discrete event simulation means that the occurrence of events managed during the simulation models the flow of time. Usually, the scale associated with this flow is termed the *logical order* among the events. This means that a total order among the events exists that reflects the sequence of the operations performed in the simulated system. In ARTISST, the *global event management engine* (described in 4.3) takes care of enforcing the total order among the messages, even when they come from multiple input modules of a single module.

But, in addition to this implicit[1] logical total order, all the ARTISST events are also timestamped. This timestamp defines the absolute global *simulated real-time scale*. The real-time scale is shared across the simulated system(s) and the external environment. Of course, it has nothing to do with the "real life" time scale: it is purely virtual, only valuable for the simulated world.

[1]The logical total order among the messages is not available to the simulated system's programmer.

The real-time scale actually represents an additional partial order among the messages: if a message $m_1$ comes logically before a message $m_2$, then timestamp($m_1$) $\leq$ timestamp($m_2$).

All the delays passed to the `hold_cpu()` primitive in the system simulator are relative to this real-time scale. The resolution for the simulated real-time scale is the nanosecond.

## 4.3 Global event management engine

The Artisst architecture is based on the *global event management engine* library, used for connecting the modules to one another. This library is in charge of the propagation of the messages while enforcing the total logical order among the messages and cannot be extended.

Thanks to this global event management engine library, the designer of modules is able to consider the input events for the modules as a chronological stream of messages, without having to take care of their ordering, even if multiple input modules are connected to it.

## 4.4 Sample modules

We describe here the input, output, system simulator and gateway modules that were introduced by figure 5. These sample modules are available by default on Artisst, and may be used or even extended. Of course, the Artisst user may implement and use her/his own set of modules in addition to these ones.

### 4.4.1 Input modules for the system simulator

Basically, an input module for the system simulator is in charge of simulating the stream of events generated by the external environment of the simulated system. In Artisst, this is modeled by the generation of (simulated) hardware interrupts that the simulated RTOS is in charge of processing. Two such modules are available:

**Random-based events generator.** This module provides a stream of events of a given type generated at dates chosen according to a probabilistic distribution. 18 distributions are available by default (constant *ie* purely periodic, binary, uniform, normal (Gauss), Erlang, $\gamma$, Poisson, ...). Artisst also supports user-defined (cumulative or frequency-based) distributions.

**Log-based event generator.** Generates a stream of events according to a text file listing a calendar of events to be generated. This file might have been generated by a previous simulation.

### 4.4.2 System simulator module

The system simulator is a module by itself. It is in charge of simulating the behavior of an application made of tasks on top of an operating system, with respect to the real-time scale. The simulated tasks are customizable and extensible. So is the simulated RTOS (including its scheduler).

The system simulator takes a set of (simulated) hardware interrupts as input events, and generates the output events issued by the instrumentation of the simulated application and RTOS (including its scheduler).

This module is detailed in section 5.

#### 4.4.3 Output modules for the simulator

An output module for the system simulator is essentially aimed at logging and/or analyzing the simulated system behavior.

The output modules available by default in Artisst are:

**Trace generator.** This module simply writes the messages generated by its input module(s) in text format into a file. This file is suited for being read by the log-based event generator described above.

**Statistical analyzer.** This module is in charge of analyzing the temporal characteristics of its submitted event stream: for the moment it only computes the mean value and variance (but will be extended). It can also draw graphs (cumulative or frequency-based) ready to be printed (using `gnuplot` or `GNU plotutils`).

**Grantt chart (chronogram).** This module represents the input events as items in a graphical window (currently: `X-Window` system with `gtk` widgets) whose X axis is the simulated real-time scale. It allows to zoom in or out along the simulated real-time scale.

#### 4.4.4 Gateway module

This is a does-nothing module, which is only in charge of relaying its input message stream(s) to its output(s). Its main interest is that it can be used as the "message pump" of the simulation circuit, also known as the "event loop".

### 4.5 Extensibility

Implementing new modules is a light task in Artisst since the framework defines a base class for the modules, and prevents the module designer from taking care of any input messages ordering (see 4.3).

Designing a module then consists in sub-typing the base module class, and defining the methods enumerated in the base class needed for getting the messages generated by the module. Once this is done, the designer may choose to add any object data or method to his module.

The default modules provided by Artisst are all designed this way, including the complex system simulator module.

## 5 The system simulator module

Here, we focus on the most interesting default Artisst module regarding the purpose of Artisst: the system simulator module. This module is responsible for managing the simulated tasks, for forwarding the (simulated) hardware interrupts to the user-defined interrupt service routines, and for generating the Artisst instrumentation messages.

## 5.1 Overview

A system simulator module actually simulates a single instance of the RTOS and its associated application tasks. That is, it represents a single CPU managed by the simulated RTOS. A simulation circuit may contain multiple interconnected system simulator modules, thus simulating a distributed or multiprocessor system.

The system simulator module itself is modular and extensible: it is designed as a set of well-defined distinct software layers. The figure 6 gives an illustration of its overall structure.

Actually, the RTOS services are divided into 2 categories: *i)* the system simulator module's core RTOS services (task management routines, interrupts management mechanisms, and system time management routines), which are not customizable because they correspond to the basic functionalities of the module; and *ii)* the additional RTOS extension services that the simulated system's designer may customize.

Besides the RTOS extensions, a large subset of the entities constituting this module (the application tasks, the interrupt service routines, the scheduler) are completely customizable.

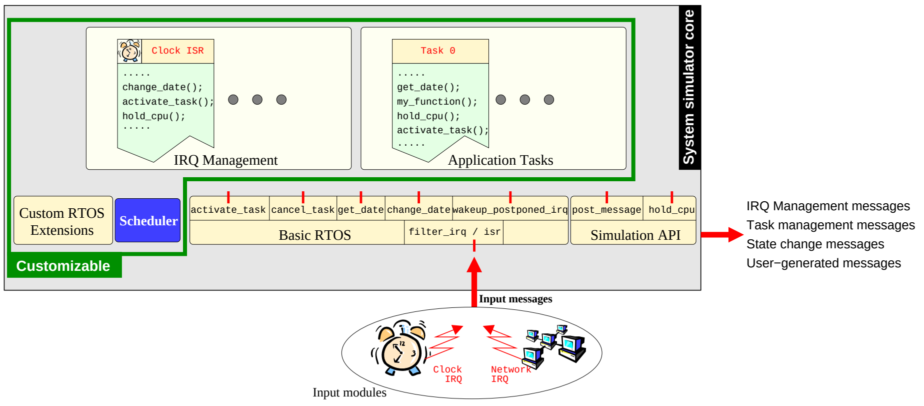


Figure 6: System simulator module overall structure

## 5.2 System simulator core

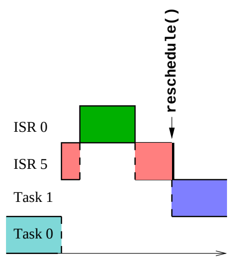
The system simulator core (shaded background gray box on figure 6) is the underlying infrastructure that is in charge of managing the execution contexts (interrupts, tasks), and which acts as a proxy between the simulated system and the ARTISST architecture (input/output modules by way of messages).

More precisely, its job is:

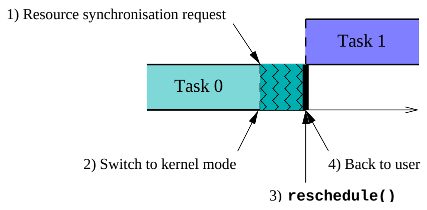
- To react to the (simulated) hardware interrupts it receives by launching the associated Interrupt Service Routine (*ISR*) in the simulated system.
- To call the system scheduler (`reschedule()` function) each time a scheduling decision is necessary, that is: at the end of an application task, at the end of the most external ISR (ISRs are further detailed in 5.3.3), and when switching from the simulated kernel back to a task (*ie* after a system call).

- To switch to the correct task each time a scheduling decision imposes to do so.

The operation of the system simulator core is tightly related to the `hold_cpu()` primitive. Actually, preemption is impossible outside an `hold_cpu()` call, since the simulated computations are instantaneous with respect to the simulated real-time. And inside a `hold_cpu()`, preemptions are allowed. They can be due to *i)* (simulated) hardware interrupts and their associated ISR (figure 7(a)), *ii)* RTOS services that induce resource synchronization and hence task-switching (figure 7(b)).



(a) ISR Preemption



(b) Resource synchronization

Figure 7: Sources of task or ISR preemption

## 5.3 Simulated system

### 5.3.1 Extensible RTOS API

The RTOS API is a set of functions (C code), or a base class (C++). The default API is rather primitive. It only consists in functions for managing tasks in the system (creation and deletion), for handling the (simulated) hardware interrupts, and for setting and getting the system date (further detailed in 5.3.4).

When necessary, the designer may extend this API by adding his own set of functions. For example, the designer may add the necessary primitives for resource synchronization (mutex, semaphores, conditions, ...). Implementing this kind of service would also require to use a scheduler that is aware of these resources, so that it won't schedule a task that is waiting for a resource possessed by an other task.

### 5.3.2 Customizable scheduler

The scheduler is a single function associated with the system simulation module: `reschedule()`. ARTISST only comes with the function declaration. The simulated system's designer is in charge of implementing it, or of choosing it among the default schedulers available.

The `reschedule()` function is automatically called by the system simulator core whenever a scheduling decision needs to be taken, *ie* at the end of a task, or when switching from kernel mode back to user mode, or after having handled the simulated interrupts. It takes as input the list of the tasks status (as described in 3.2) of the tasks currently executing in the system. As output, it returns the identifier of the task to be scheduled for execution on the CPU.

The scheduler largely relies on the informations provided by the task model and status data structures (see section 3). So, adding new functionalities to the simulated RTOS, which implies adding new decision criteria to the scheduler, also implies that the task model and status data structures contain the necessary information. For example, when implementing resource synchronization inside the RTOS extensions, the task model and status must be extended so that the resources that the tasks possess or are waiting for, be mentioned. This way, using these two new task data structures, the system scheduler can deal with resource synchronization.

Currently, ARTISST comes with two default schedulers: Earliest Deadline First[3] (EDF) and TaskPair Scheduling[4] (TPS). Both are dynamic on-line schedulers. EDF simply schedules the task with the closest deadline, and thus relies on the basic task model for the *relative deadline* attribute.

TPS achieves on-line guarantees for the tasks, together with an exception-handling mechanism. With TPS, a task is splitted in two sub-tasks: the main task (MT) and the exception task (ET). The timing characteristics (WCET or arrival law for instance) of the MT may not be fully characterized, whereas a WCET of the ET must be known. As a consequence, the task model for TPS obviously differs from the basic task model. The scheduler in TPS is based on an acceptance test associated with a specified ET scheduling policy which must at least guarantee that the ET of the accepted tasks will meet their deadline. Once a task has been accepted, the MT is scheduled according to a dynamic on-line scheduling algorithm. If its relative deadline is to be missed, the MT is cancelled, and the ET is scheduled for execution according to the specified ET scheduling policy.

### 5.3.3 Interrupt management

The IRQ ("Interrupt ReQuest") management in ARTISST consists in three primitives of the simulated RTOS: `filter_irq(int irq_level)`, `isr(int irq_level)` and `wakeup_postponed_irq(irq_mask mask)`.

`filter_irq(int irq_level)` is a callback: it is automatically called whenever a (simulated) hardware IRQ is raised. It is in charge of indicating whether the system accepts it for processing by an interrupt service routine (ISR), rejects it, or postpone it for later processing after it has been explicitly re-activated (this provides an interrupt masking mechanism).

The `isr()` primitive is also a callback: it is automatically called whenever a pending (simulated) hardware IRQ is ready to be processed. It corresponds to the real processing associated with the IRQ.

For both `filter_irq()` and `isr()`, ARTISST only comes with the function declaration. The simulated system's designer is in charge of implementing them.

The `wakeup_postponed_irq(irq_mask mask)` is the RTOS primitive in charge of re-enabling the raised IRQ that have been postponed (by the `filter_irq()` callback), according to the `mask`.

In ARTISST, the ISR are prioritized by default, which means that the ISR processings can be nested (The ISR for IRQ 0 having the highest priority), as shown in the figures 7(a) and 8.

Dealing with the nesting of the ISR, the system simulator takes care of calling the scheduler only when needed, that is: at the end of the outermost ISR execution only.

### 5.3.4 System and real time scales

In 4.2, the real-time was defined as the time scale according to which all the events in the whole system are relative (timestamped). This is an absolute and continuous time scale: any event can have any timestamp that can take an arbitrary value in $\mathbb{R}^+$.

In the real OS world, the OS has only access to a discrete time scale known as the "system time" or "system clock", or simply "clock". The OS is in charge of updating it. Usually, it is updated by the *clock* ISR.

Artisst follows the same scheme: the flow of events delivered to the system simulation module defines the real-time, and the calls to `set_date()` define the system time, which is completely discrete and may be decorrelated with the real-time. This allows to simulate a drift of the system date compared to the real-time, or jitter, or any other (mis)behavior.

The Artisst system simulation module takes care of updating the system time each task has been consuming so far, based on the value of this system time clock. It also updates the system time the RTOS kernel has been consuming.

The simulated system's designer is simply in charge of updating this system clock when needed: normally this would be done inside the clock ISR, but it could also be part of the remaining RTOS or application code (*eg* when dealing with clock synchronization in a simulated distributed system).

All the scheduling decisions that need to take time into account (which is rather frequent in the real-time computing field) should only be based on the system time. The simulated real-time has no meaning for the simulated system: it is kept available in the task execution status only for evaluating the overall system behavior.

## 5.4 Generated events

A typical simulation session would generate the event stream shown in figure 8.

All these events are automatically generated by the system simulation module. The user may also generate any message using the `post_message()` primitive.

The basic events generated by the system simulator are described in table 1. This set of basic events defines a base class, which can be extended through object-oriented inheritance.

Distinguishing between the arrival of an IRQ (`INTERRUPT_RAISED`) and the beginning of the related ISR by the simulated RTOS (`INTERRUPT_ENTER`) is due to the fact that the ISR are prioritized, as figure 8 suggests, and as section 5.3.3 explained.

## 5.5 Distributed systems simulation

With Artisst, simulating a distributed system simply means connecting two or more system simulation modules together by a simulated network.

| Date (sec.ns) | Event Type | Event Parameters |
|---|---|---|
| 56.300000000 | INTERRUPT_RAISED | 2 ACCEPT |
| 56.300000000 | SUSPEND_TASK_EXECUTION | 0 |
| 56.300000000 | INTERRUPT_ENTER | 2 |
| 56.300250000 | INTERRUPT_RAISED | 1 ACCEPT |
| 56.300250000 | INTERRUPT_ENTER | 1 |
| 56.300500000 | INTERRUPT_RAISED | 0 ACCEPT |
| 56.300500000 | INTERRUPT_ENTER | 0 |
| 56.301500000 | INTERRUPT_LEAVE | 0 |
| 56.302250000 | KERNEL_SERVICE_ENTER | |
| 56.303250000 | TASK_ACTIVATION | Task0 1 14.000000000 |
| 56.303250000 | KERNEL_SERVICE_LEAVE | |
| 56.303250000 | INTERRUPT_LEAVE | 1 |
| 56.304000000 | KERNEL_SERVICE_ENTER | |
| 56.305000000 | TASK_ACTIVATION | Task1 2 7.000000000 |
| 56.305000000 | KERNEL_SERVICE_LEAVE | |
| 56.305000000 | SCHEDULER_ENTER | |
| 56.306000000 | SCHEDULER_LEAVE | 2 |
| 56.306000000 | INTERRUPT_LEAVE | 2 |
| 56.306000000 | BEGIN_TASK_EXECUTION | 2 |

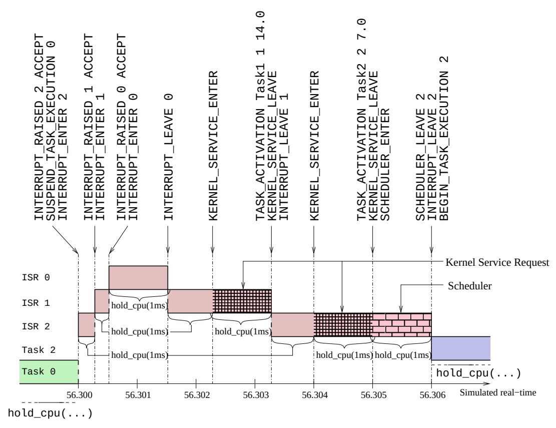


Figure 8: Events generated by a typical system simulation session

A basic simulated network can simply be a connection between the modules in such a way that a simulated network packet sent by one node is received as a network interrupt by the other nodes. A more elaborate simulated network could be a dedicated module that introduces for instance packet loss, jitter, variable delay.

Furthermore, the decorrelation between the absolute real-time scale, and the local system-time scales, allows for the simulation of realistic timing anomalies, such as clock desynchronization between nodes.

### 5.6 System costs simulation

As mentioned earlier in this section, the only way to simulate CPU occupation is to call the `hold_cpu()` function. Artisst allows this function to be called from the application tasks of course. But it also allows it to be called from inside the RTOS kernel extensions, the scheduler, and the ISRs.

This exactly comes down to taking all the system costs into account.

## 6 Related works

The field of computer-driven simulation proposes a large panel of simulators. Some come in the form of a general purpose simulation programming language, others are libraries that provide simulation functionalities to standard programming languages, and some are a version of an operational system targeted to simulation. Here, we present some related works regarding discrete event simulation, and present in which aspect(s) they are or they are not suited to our goal: building a framework for simulating and evaluating various RTOS and in particular their scheduler.

| Message Class | Message Name | Meaning |
|---|---|---|
| IRQ Management | `INTERRUPT_ENTER` | Indicates that a (simulated) hardware interrupt reached the simulated CPU. Such an interrupt is marked `pending` for immediate or later processing by an Interrupt Service Routine (ISR) of the simulated RTOS. |
| | `INTERRUPT_WAKEUP_POSTPONED` | IRQ that were postponed are re-activated |
| | `INTERRUPT_ENTER` | Indicate the beginning and the end of the processing of a pending IRQ by an ISR. |
| | `INTERRUPT_LEAVE` | |
| Task Management | `TASK_ACTIVATION` | Creation of the task. |
| | `BEGIN_TASK_EXECUTION` | First execution of the (previously created) task. |
| | `RESUME_TASK_EXECUTION` | The task's execution is resumed. |
| | `SUSPEND_TASK_EXECUTION` | The task's execution is preempted. |
| | `END_TASK_EXECUTION` | The task's execution ends. |
| System state change | `KERNEL_SERVICE_ENTER` | Indicate that a task or an ISR is requesting a kernel service, in order to take the RTOS services' costs (except the scheduler) into account. |
| | `KERNEL_SERVICE_LEAVE` | |
| | `SCHEDULER_ENTER` | Indicate that the system switches from the current ISR or task to scheduler and vice-versa, in order to take the scheduler costs into account. |
| | `SCHEDULER_LEAVE` | |

Table 1: Basic system simulator instrumentation messages

## 6.1 General-purpose discrete event simulators

In a discrete-event simulator (DES), the logical time reflects the state changes in the system, *aka* the occurrence of events. Typical application domains for discrete event simulators are digital electronics, computing and network simulation.

This class of simulator matches our application domain, since it is feasible to map the problem of simulating a complex computing system to a set of events being generated and processed by logical simulated entities.

However, to our knowledge, the programming languages tailored to simulation (QNAP2[5] for example), or the simulation APIs available to general purpose programming languages (ADEVS[2] for example) both suffer from the lack of a primitive similar to the Artisst `hold_cpu()` primitive. That is: they don't provide a way to simulate the occupation of a shared resource during a given period, **independently** of the preemptions.

It could be of course possible to implement this primitive on top of the existing tools, but this would come down to managing some event calendar on top of the default event management engine, which precisely appears to have been the major difficulty during the Artisst development.

## 6.2 Computing systems simulators

In the more specific field of the computing systems simulation, which is a special category of DES, a relatively large panel of simulation frameworks already exists. There are basically two existing alternatives that could fit the needs for real-time system simulation: the scheduling simulators, or the RTOS simulators.

A large subset of the scheduling simulators (Scheduler for Java[6] or Schedsim[7] for example) considers the tasks as abstract time slots, *ie* they are defined by a probabilistic execution time and/or arrival law. This kind of simulators ignores the internal processing of the tasks and the precedence constraints. Some other simulators in this class don't allow to implement one's own scheduler.

An other subset of scheduling simulators (MAST[8], PERTS[9]) offers a rich environment for evaluating a set of tasks, defining the synchronization constraints for instance. But everything is oriented toward off-line evaluation in the form of an off-line feasibility (or "what-if") test (typically: RMA[10]). Apart from a work still in progress in MAST[8], but oriented towards synthetic workload simulation, no actual simulation for evaluation in a varying environment can be carried out, which is one of our goals.

Concerning the RTOS simulators, it is noticeable that almost any RTOS proposes its own simulator. As a consequence, the main limitation of the classical RTOS simulators, except for CarbonKernel[11], is that they are targeted to a particular API and scheduler (VxSim[12], OSE Softkernel[13]), thus contradicting one of the Artisst goals: being generic and extensible. Furthermore, all these existing RTOS simulators focus primarily on resource synchronization (deadlocks chasing), rather than in fine-grained timing evaluation of the whole system; real-time behavior is rather secondary, if not superfluous.

# 7 Conclusion & perspectives

The Artisst framework provides the basic tools for RTOS simulation and evaluation targeted to simulation in an evolving environment, with possibly unknown system timing characteristics.

Artisst is strongly oriented towards modularity: a simulation consists in interconnecting modules that are in charge of modeling the environment, the system, and of evaluating the behavior of the simulated system. In conjunction with the object-oriented approach characterizing this architecture, it results in a great extensibility and customization potential.

The central `hold_cpu()` primitive for the simulated system enables to accurately simulate a real-time system. Furthermore, since the `hold_cpu()` primitive can be embedded in the simulated RTOS code, it can be used in order to take the RTOS (including its scheduler and interrupt service routines) costs into account.

The Artisst framework comes with a set of default modules, and a set of default schedulers and task models. Some other input/output modules and schedulers are still under development. But the current basic software already allows to develop its own simulator without the help of these modules: extending the framework turns out to be a light task.

Finally, in addition to the simulation of centralized real-time systems, both the modular Artisst framework and the decorrelation between the absolute global real-time scale and the local system-wide system time scales, enable to simulate distributed systems.

Future work includes implementing as many schedulers and RTOS personalities as possible. Another perspective is designing, implementing, and evaluating (comparison with other algorithms, such as Spring[14], TPS[4], Delacroix[15]) a reconfiguration-based dynamic on-line scheduler.

## References

[1] John A. Stankovic. Misconceptions about real-time computing. *IEEE Computer*, 21(10):10–19, 1988.

[2] Daryl R. Hild. *Discrete Event System Specification (DEVS) / Distributed Object Computing (DOC) Modeling and Simulation.* PhD thesis, University of Arizona, March 2000.

[3] C. L. Liu and James W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *Journal of the ACM*, 20(1):46–61, 1973.

[4] H. Streich. Taskpair-scheduling: An approach for dynamic real-time systems. *Int. Journal of Mini & Microcomputers*, 17, No. 2:77–83, 1995.

[5] Dominique Potier Michel Véran. Qnap2 : A portable environment for queueing systems modelling. Technical Report Rapports de Recherche No.314, Institut National de Recherche en Informatique et en Automatique, 1984.

[6] Eric Lefevre. *Scheduler for Java.* http://www.chez.com/elefevre/english/scheduler/scheduler.html.

[7] *SchedSim Datasheet.* http://www.spacebel.be/schedsim.htm.

[8] José María Drake, Michael González Harbour, José Javier Gutiérrez, and José Carlos Palencia. Mast: Modeling and analysis suite for real time applications. In *13th Euromicro Conference on Real-Time Systems*, Delft, The Netherlands, June 2001.

[9] Jane W. S. Liu, Juan-Luis Redondo, Zhong Deng, Too-Seng Tia, Riccardo Bettati, Ami Silberman, Matthew Storch, Rhan Ha, and Wei-Kuan Shih. Perts: A prototyping environment for real-time systems. Technical report, CS Illinois at Urbana-Champaign, 1993.

[10] L. Sha, M. Klein, and J. Goodenough. Rate monotonic analysis for real-time systems. Technical report, 1991.

[11] Loic Dachary and Philippe Gerum. *CarbonKernel User Manual.* http://www.carbonkernel.org/docs/latest/ck-user.pdf, 1.3 edition, July 2001.

[12] *VxSim Datasheet.* http://www.wrs.com/products/html/vxsim_ds.html.

[13] *OSE Soft Kernel.* http://www.enea.com/prodserv/tools/softenvironment.asp.

[14] A. Bondavalli, J. Stankovic, and L. Strigini. Adaptable fault tolerance for real-time systems. Technical Report UM-CS-1994-039, University of Massachusetts, Amherst, Computer Science, May, 1994.

[15] J. Delacroix. Towards a stable earliest deadline scheduling algorithm. *Real-Time Systems*, 10:263–291, 1996.