ARTISST: An Extensible and Modular Simulation Tool for Real-Time Systems*

David Decotigny Isabelle Puaut
Institut de Recherche en Informatique et Systèmes Aléatoires
Campus de Beaulieu, 35042 Rennes Cédex, France
E-mail: {ddecotig,puaut}@irisa.fr

Abstract

*ARTISST (ARTISST is a **R**eal-**Ti**me **S**ystem **S**imulation **T**ool) is a modular event-driven simulation framework for real-time systems. It is targeted towards the performance evaluation of complex dynamic real-time 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, since it allows, among other things, to take the operating system (including its scheduler) costs into account. Furthermore, thanks to its modular and object-oriented software architecture, the simulator is not dedicated to a particular Operating System API, but is fully customizable instead. This paper describes ARTISST, and illustrates its functionalities and customization facilities on two example real-time systems.*

1. Introduction

1.1. Context

The context of this work is the one of *real-time* computing systems, in which the correctness of the computations does not only depend upon the logical correctness of the computation but also upon the time at which the result is produced [12].

An approach for the evaluation of a real-time system consists in *analyzing* its behavior according to a set of metrics before any real or simulated execution. For example, 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 *schedulability analysis*, which establishes whether all the time constraints will be met or not when the system operates. Actually, the evaluations based on off-line analysis of the system assume that the system designer did provide at specification time the necessary material needed to characterize 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 the evaluation of a real-time system consists in studying its behavior according to a set of metrics through *execution* (examples of metrics: guarantee ratio, release jitter). This kind of evaluation, be it conducted on real or simulated executions of the system, allows for the evaluation of real-time systems that may be less statically determined than with an analytic approach, 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.* evolve dynamically), thus making it difficult if not impossible to be statically analyzed.

1.2. Motivations

In a 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 run-time 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.

Another advantage of evaluating a real-time system through simulated execution is that it is possible to study the system behavior under the very same execution conditions, with successively varying internal characteristics (scheduler costs, scheduling policy...).

*This work as been published in the 5th IEEE International Symposium on Object-oriented Real-time distributed Computing, April 2002.

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 (refer to section 6), we decided to implement our own generic, customizable and extensible real-time systems simulation framework: ARTISST.

1.3. ARTISST overview

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 centralized or distributed real-time systems[1]. 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*.

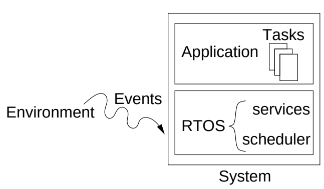


Figure 1. Model of the simulated world

The main contributions of ARTISST are :

1. its ability to be customized in order to simulate a large panel of RTOS schedulers and APIs;
2. its ability to simulate complex task execution patterns at a user-defined level of detail;
3. its ability to take the RTOS (including its scheduler) costs into account in the simulation.

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.

1.4. Paper organization

This paper is organized as follows. In section 2, we present the extensible task model that the ARTISST framework is able to simulate. In section 3, we detail the architecture of the ARTISST framework, and discuss its extensibility. In section 4, we present the system simulator core engine, responsible for the simulation of the studied system. Section 5 illustrates the evaluation of real-time systems using ARTISST with two simple examples, and gives an overview of the work needed to implement them. Section 6 discusses the related work in the field of simulation for the evaluation of real-time systems. Finally, section 7 gives our conclusions and discusses future work.

[1]It is currently available as an open source software by asking the authors, and will be publicly released under an open source license soon.

2. Task Model & Task Execution

ARTISST identifies a simulated *task* in the system as being a sequential flow of control which is being executed by the simulated CPU, or waiting for execution by the CPU (*i.e.* 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 system simulation module, and available to the simulated system.

Figure 2 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.

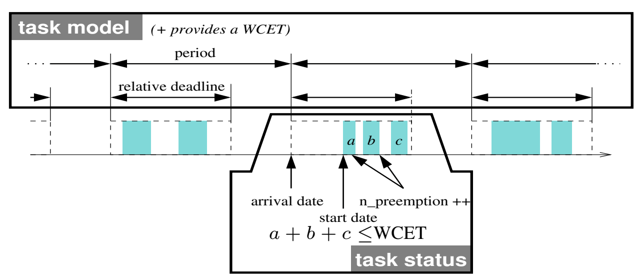


Figure 2. Task model & status example for a periodic task preempted by other tasks (not shown)

2.1. Basic Task Model

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

The basic task model contains some structural information such as the maximum number of concurrently executing tasks belonging to this model, or the computation code to be executed by each task instance. It also provides some temporal behavioral information regarding the tasks that belong to this model, such as the arrival law and its parameters (for instance, period for periodic tasks, minimum interarrival time for sporadic tasks), and such as the temporal characteristics or constraints of the tasks.

For example, on figure 2, 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 execution is characterized by a specified worst case execution time (*WCET*).

2.2. Basic Task Execution Status

The basic task status is a data structure that provides the execution context of the task in the host system for the simulation (*i.e.* stack and program counter), a reference to the task model the task belongs to, and some information regarding the behavior of the task (for instance, consumed simulated CPU time, number of interruptions or preemptions).

All this information is provided by the core of the system simulator, and is made available to any entity in the system simulator (simulated tasks and RTOS, including its scheduler and interrupt handlers).

2.3. Customization & Extensibility

Both the task model and the task execution status are essentially aimed at servicing the simulated RTOS and its scheduler. The designer of the simulated RTOS is not urged to use any of the default basic attributes previously enumerated: she/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 RTOS API or scheduler, which can easily be achieved using classical object-oriented inheritance.

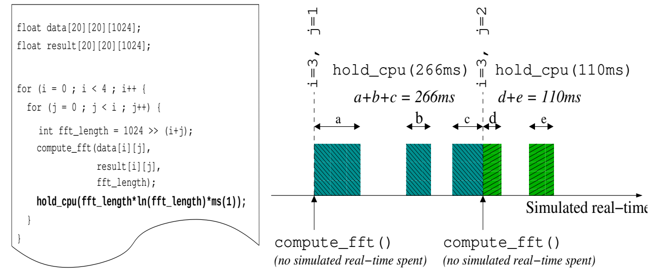
2.4. Task computations

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

Any computation made inside the task, no matter how complex it is, actually takes absolutely no time with respect to the simulated real-time. In ARTISST, simulating time advance is achieved by calling the `hold_cpu()` function which is part of the ARTISST API.

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. Figure 3 illustrates this property: the computation of the deepest iteration when $i = 3$ and $j = 1$ simulates 266ms of CPU occupation with respect to the simulated real-time, though it is being preempted twice; *i.e.* calling `hold_cpu(266ms)` ensures that $a + b + c = 266\text{ms}$.

The programmer may add as many calls to `hold_cpu()` as he wants inside his code, and pass



(a) Sample code for a task

(b) CPU time consumption with respect to the simulated real-time in the presence of preemptions by other tasks (not shown)

Figure 3. CPU time consumption simulation

them any valid C expression. 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 3(a) shows).

Furthermore, as we shall see in section 4.6, 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, including its scheduler.

3. ARTISST overall software architecture

Here, we describe the way the overall ARTISST simulation framework is structured: we define the notions of modules and events (3.1), and present the way the simulated real-time is managed (3.2). We then give an overview of the default ARTISST modules available (3.3).

3.1. ARTISST modules and events

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.

Modules. A module is an object in charge of reacting, generating or propagating the simulation events. 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 4. The ARTISST event management engine is in charge of the propagation of the events from and to the modules according to this circuit.

ARTISST adopts the object-orientation paradigm, which allows to rely on a unique base class in order to define a common interface for all the modules. This is essential to

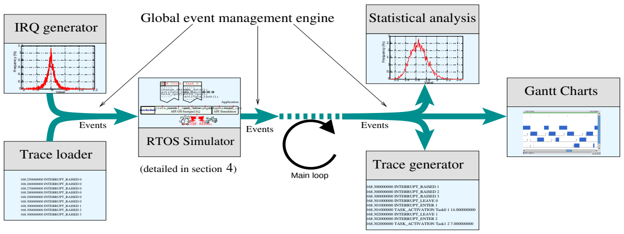


Figure 4. Modular structure in ARTISST

build a generic event management engine, and it allows the designer to extend this base class in order to add new functionalities to the modules.

Events. The *events* are the objects 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, for a "task suspend" event: the task identifier of the task which is being suspended); *iii)* a timestamp related to the simulated real-time scale (see section 3.2). Translated into text format, 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 events.

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

3.2. Events and real-time in ARTISST

Discrete event simulation means that the occurrence of the events that are generated during the simulation models the flow of time. In ARTISST, the *global event management engine*, responsible for the messages propagation, takes care of enforcing this *total logical order* among the messages, even when they come from the multiple input modules of a single module.

In addition to this implicit total logical order, all the ARTISST messages are also timestamped. This timestamp represents an additional partial order among the messages (if a message m_1 comes logically before a message m_2, then $\mathrm{rt\_date}(m_1) \leq \mathrm{rt\_date}(m_2)$), and defines the absolute global *simulated real-time scale*. The real-time scale is shared across the simulated system(s) and the external environment.

All the delays passed to the `hold_cpu()` primitive in the system simulator are relative to this real-time scale.

3.3. Default modules available

Input modules for the simulator. An input module for the system simulator is in charge of generating the input messages for the system simulator module (in the form of simulated *hardware interrupts*) in order to simulate the environment *stimuli*. ARTISST comes with 2 default input modules that respectively generate the input messages according to a random distribution (one of the 18 default distributions available, or a user-defined distribution), or according to a previously generated log file.

System simulator module. The system simulator is a module by itself. It takes a set of (simulated) hardware interrupts as input messages, and generates the output messages issued by the instrumentation of the simulated application and RTOS (including its scheduler). This module is detailed in section 4.

Output modules for the simulator. An output module for the system simulator is essentially aimed at logging and/or analyzing the simulated system behavior. ARTISST comes with a set of 3 default modules that are in charge of analyzing the behavior of the system being simulated in the form of *i)* a chronogram (X Window client), *ii)* execution traces in readable text format, *iii)* a statistical analysis (gives the probabilistic distribution of a set of user-defined metrics).

4. The system simulator module

Here, we focus on the most interesting default ARTISST module regarding the purpose of ARTISST: the system simulator module.

4.1. Overview

A system simulator module actually simulates a single instance of the RTOS and its associated application tasks. The system simulator module itself is modular and extensible: it is designed as a set of well-defined distinct software layers. Figure 5 gives an illustration of its overall structure.

The RTOS services are divided into 2 categories: *i)* the 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 designer of the simulated system 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.

4.2. System simulator core

The system simulator core (shaded gray box in the background of figure 5) is the underlying infrastructure that is

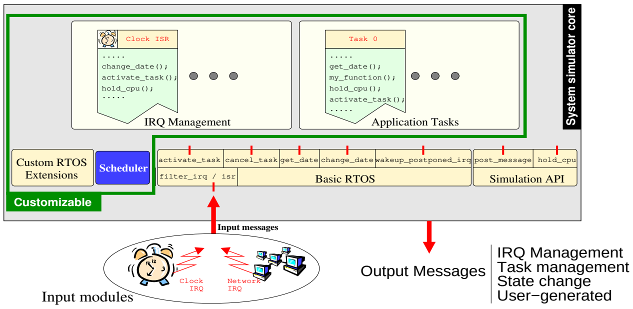


Figure 5. System simulator module structure

in charge of managing the execution contexts (interrupts, tasks), and which acts as the interface 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.
- 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, which may lead to preemptions. By default the execution of the RTOS services is fully preemptive. However, it is possible to mask the interrupts in order to design a set of non-preemptive RTOS services.

4.3. Simulated system

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 the functions used to manage tasks in the system (creation and deletion), to handle the (simulated) hardware interrupts, and to set and get the system date (further detailed in 4.3).

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 (message queues, 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 another task. By default, ARTISST comes with a very simple mutex facility.

Customizable scheduler. The scheduler is a single callback associated with the system simulation module: `reschedule()`. ARTISST comes with the function declaration, and the designer of the simulated system is in charge of either 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, *i.e.* 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 currently eligible in the system. As output, it returns the identifier of the task to be scheduled for execution.

The scheduler largely relies on the informations provided by the task model and status data structures (see section 2).

Default schedulers available. Currently, ARTISST comes with four default schedulers: a static priority scheduler, and three dynamic on-line schedulers: the Earliest Deadline First scheduler [8] (EDF), EDF with an acceptance test based on the evaluation of the residual times of the tasks [2] (GED), and TaskPair Scheduling [14] (TPS).

TPS achieves on-line guarantees for the tasks, together with an exception-handling mechanism. Actually, in TPS, the scheduler handles *taskpairs*, not directly tasks. A taskpair is composed of two 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. The scheduler in TPS is based on an acceptance test associated with a specified ET scheduling policy. Once a taskpair 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.

As a consequence of TPS being in charge of scheduling taskpairs instead of tasks, the task model for TPS obviously differs from the basic task model; in ARTISST, the implementation of TPS takes advantage of the object-oriented approach of ARTISST regarding both the task model and execution status data structures.

In ARTISST, we have chosen to implement the TPS acceptance test for a taskpair as the computation of a feasible non-preemptive scheduling plan for the accepted exception tasks. The main tasks are scheduled in a preemptive way according to the EDF policy.

Interrupt management. The IRQ ("Interrupt ReQuest") management in ARTISST consists in three primitives of the simulated RTOS responsible for masking, unmasking and handling the (simulated) hardware interrupts. The interrupt handlers ("*ISR*", for *Interrupt Service Routine*) are prioritized by default, which means that they can be nested (The ISR for IRQ 0 having the highest priority). Dealing with the nesting of the ISR, the system simulator is in charge of calling the scheduler only when needed, that is: at the end of the outmost ISR execution only.

System and real time scales. In the real OS world, the OS has not access to the absolute, real-time scale, but 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: by the clock ISR).

ARTISST follows the same scheme: the flow of messages delivered to the system simulation module defines the real-time, and the calls to `set_date()` define the system time (usually inside the clock ISR), which is completely discrete and may be decorrelated with the real-time. This allows to simulate drift, jitter or any other (mis)behavior of the system date compared to the real-time (or compared to the system dates of other simulated systems in a distributed system simulation).

All the scheduling decisions that need to take time into account should only be based on the system time.

4.4. Generated messages

A typical simulation session would automatically generate the message stream shown in figure 6, which introduces most of the basic messages defined in ARTISST. This set of the basic messages defines a base class (detailed in [4]), which can be extended through object-oriented inheritance.

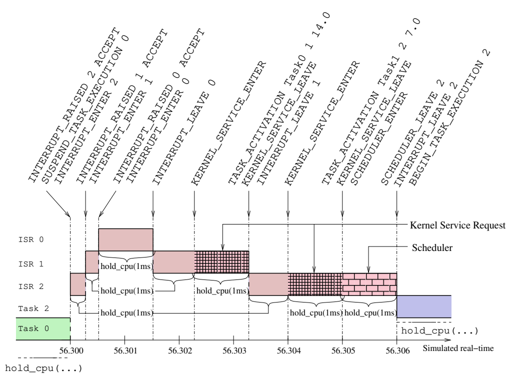


Figure 6. Simulation messages generated

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 section 4.3 explained.

4.5. Distributed systems simulation

With ARTISST, simulating a distributed system means connecting two or more system simulation modules together by a simulated network, in which a simulated network packet sent by one node is received as a network interrupt by the other nodes, with user-defined transmission delays, message loss and alteration.

In ARTISST, 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.

By default, in ARTISST two different types of modules are responsible for carrying network simulations. One module (unique in the whole system) is responsible for enforcing the global consistency of the real-time scale across the nodes. And a set of modules (one per simulated CPU) is responsible for modeling the network links, in order to simulate variable transmission delays, message loss or alteration. We do not detail them here for space considerations.

4.6. 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.

5. Sample application

In this section, we discuss the usability of ARTISST by showing how the evaluation of two real-time systems was conducted.

5.1. Simulated systems

The simulated systems, "GED" and "TPS", are made of tasks scheduled respectively according to the GED and the TPS scheduling policies (see section 4.3). In this application, the mean simulated execution times (by way of `hold_cpu()`) of the exception tasks in TPS are ten times shorter than those of the main tasks. In both simulated systems, the tasks (the taskpairs in TPS) are independent of each others (there are no resource synchronization or precedence constraints).

5.2. Workload

In order to compare the two systems, the same two input modules are used in both simulations.

The first one generates the system clock interrupts at a fixed rate with respect to the real-time (every 10ms). Actually, it is one of the ARTISST default input modules (namely, a random-based event generator based on a Dirac distribution).

The second input module is in charge of *i)* generating the interrupts which directly activate the tasks/taskpairs; *ii)* generating the corresponding task/taskpair execution attributes (*i.e.* WCET, simulated execution time, relative deadline). It is derived in a straightforward manner from the standard default input modules. The rate of the task activations is given by a normal distribution centered on a period of 100 system clock ticks. Each of these tasks are declared with a uniformly distributed WCET between 30 and

200 system clock ticks. The effective simulated CPU occupation time is given by this WCET minus a uniformly distributed residual time between 0 and 20 system clock ticks. The deadline is computed according to a normal distribution in such a way that the mean utilization of the individual tasks (defined as the execution time/relative deadline ratio) equals a given value named the "*individual load factor*" in the following. This individual load factor is the only variable input parameter for these evaluations.

5.3. Implementation of the evaluated systems

The ISR of the system simulator module connected to the clock tick generator is in charge of incrementing the system time, and the ISR handler connected to the workload generator is in charge of activating the tasks with the parameters computed by the generator. The system simulation module is in turn connected to the statistical analysis module described below.

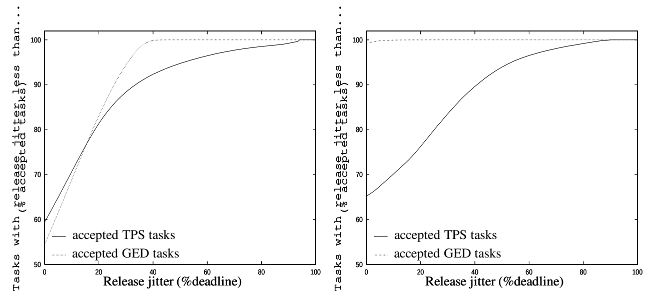
5.4. Experiment & Some results

The experiment consists in successively simulating GED and TPS with different individual load factors (described in 5.2), namely 0.1, 0.2, ..., 0.9. For each simulation, we chose to simulate the systems during 60480000 system clock ticks (*i.e.* 7 simulated days with a 10 ms system clock tick period, which takes approximately 12 hours to simulate on an Ultra Sparc 60 / 360MHz).

The instrumentation consists in sending some of the task/taskpair status data of the tasks/taskpairs being rejected or terminated (namely the arrival date, the start date, the absolute deadline and the actual executed time on simulated CPU) to a statistical analysis module. This module computes a series of metrics from the data it receives, and then constructs an histogram for each computed metric. We chose to compute the following metrics: the normalized release jitter (*i.e.* $(\mathrm{start} - \mathrm{arrival})/\mathrm{relative\ deadline}$), the normalized utilization (*i.e.* $\mathrm{execution\ time}/(\mathrm{end} - \mathrm{arrival})$), the normalized residual time (*i.e.* $(\mathrm{end} - \mathrm{arrival})/\mathrm{relative\ deadline}$).

In this experiment, we chose not to take the system costs into account. This could however be made possible by adding `hold_cpu()` calls inside the code of the RTOS and its scheduler.

As an illustration, figure 7 shows the cumulative distributions of the normalized release jitter of both systems, that were obtained for two distinct individual load factors (0.5 and 0.9).



(a) individual load factor=0.5 (b) individual load factor=0.9

Figure 7. Distribution of the normalized release jitters

6. Related work

Here, we present some related works regarding discrete event simulation, and explain in which aspect(s) they are or they are not suited to our goal: building a framework to simulate and evaluate various RTOS and in particular their scheduler.

6.1. General-purpose discrete event simulators

This class of simulators 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, most of the programming languages tailored to simulation (QNAP2 [10] for example), or the simulation APIs available for general purpose programming languages (ADEVS [6] for example) suffer from the lack of a primitive similar to the ARTISST `hold_cpu()` primitive. Of course, it could be 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 [7] for example) considers the tasks as abstract time slots, *i.e.* they are defined by a probabilistic execution time and/or arrival law. This kind of simulators ignores the internal processing of the tasks and their precedence constraints. Other simulators in this class don't allow to implement one's own scheduler. Some others rely on a static graph to represent the abstract time slots interdependencies (such as

Ghost [11] and the simulator described in [3], both from the Santa Anna's School of Advanced Studies), which makes the implementation of more elaborate constructs, such as loop or conditional computations, rather difficult.

Another subset of scheduling simulators (MAST [5], PERTS [9], DRTSS [13], or STRESS [1]) offers a rich environment suited to the evaluation of a set of tasks, which supports the synchronization constraints for instance. But it relies on a static off-line definition of the system.

The simulation tool described in [15] from the Université Libre de Belgique provides a simple language (equipped with loop constructs) to simulate the tasks computations, which allows to simulate CPU consumption and resource synchronization, but is quite limited for other constructs. It also defines another more elaborate language to implement the schedulers. But it cannot take the RTOS costs into account in a fine grained way (other than a global statistical overhead), and cannot operate at a simulated real-time granularity finer than the system clock tick. Besides, it does not allow the definition of new task models.

Concerning the RTOS simulators, it is noticeable that almost any RTOS comes with its own simulator. As a consequence, the main limitation of the classical RTOS simulators is that they are targeted to a particular API and scheduler, thus contradicting one of the ARTISST goals: being generic and extensible. Furthermore, all these existing RTOS simulators primarily focus on resource synchronization (deadlocks chasing), rather than on a fine-grained timing simulation.

7. Conclusion & Perspectives

The ARTISST framework provides the basic tools for centralized or distributed real-time systems 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(s), the optional simulated network, 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.

Future work includes implementing as many schedulers and RTOS personalities (such as a subset of the Posix 1003.1b real-time extension) as possible. Another perspective is designing, implementing, and evaluating a reconfiguration-based dynamic on-line scheduler (in comparison with other on-line dynamic scheduling algorithms).

References

[1] N. C. Audsley, A. Burns, M. F. Richardson, and A. J. Wellings. STRESS: a simulator for hard real-time systems. *SP&E*, 24(6):543–564, 1994.

[2] G. Buttazzo and J. Stankovic. Red: Robust earliest deadline scheduling. In *Proceedings of the International Workshop on Responsive Computing Systems*, 1993.

[3] A. Casile, G. Buttazzo, G. Lamastra, and G. Lipari. Simulation and tracing of hybrid task sets in distributed systems. In *Proceedings of the IEEE International Conference on Real-Time Computing Systems and Applications*, pages 249–256, Oct 1998.

[4] D. Decotigny and I. Puaut. Artisst: An extensible and modular simulation tool for real-time systems. Technical Report 1423, IRISA, ftp://ftp.irisa.fr/techreports/2001/PI-1423.ps.gz, Nov 2001.

[5] J. M. Drake, M. G. Harbour, J. J. Gutiérrez, and J. C. Palencia. Mast: Modeling and analysis suite for real time applications. In *13th Euromicro Conference on Real-Time Systems*, Delft, The Netherlands, June 2001.

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

[7] E. Lefevre. *Scheduler for Java*. http://www.chez.com/elefevre/english/scheduler/.

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

[9] J. W. S. Liu, J.-L. Redondo, Z. Deng, T.-S. Tia, R. Bettati, A. Silberman, M. Storch, R. Ha, and W.-K. Shih. Perts: A prototyping environment for real-time systems. In *Proceedings of the 14th IEEE Real-Time Systems Symposium*. CS Illinois at Urbana-Champaign, Dec 1993.

[10] D. P. Michel Véran. Qnap2 : A portable environment for queueing systems modelling. Technical Report 314, Institut National de Recherche en Informatique et en Automatique, 1984.

[11] F. Sensini, G. C. Buttazzo, and P. Ancilotti. Ghost: A tool for simulation and analysis of real-time scheduling algorithms. In *Proceedings of the IEEE Real-Time Educational Workshop*, pages 42–49, Jun 1997.

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

[13] M. F. Storch and J. W.-S. Liu. A simulation environment for distributed real-time systems. In *Proceedings of the SCS Simulation Multiconference*, Apr. 1994.

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

[15] S. D. Vroey, J. Goossens, and C. Hernalsteen. A generic simulator of real-time scheduling algorithms. In *Proceedings of the 29th Simulation Symposium*, pages 242–249, Apr 1996.