A Generic RTOS Model for Real-time Systems Simulation with SystemC

R. Le Moigne, O. Pasquier, J-P. Calvez
Polytech, University of Nantes, France
rocco.lemoigne@polytech.univ-nantes.fr

Abstract

The main difficulties in designing real-time systems are related to time constraints: if an action is performed too late, it is considered as a fault (with different levels of criticism). Designers need to use a solution that fully supports timing constraints and enables them to simulate early on the design process a real-time system. One of the main difficulties in designing HW/SW systems resides in studying the effect of serializing tasks on processors running a Real-Time Operating System (RTOS). In this paper, we present a generic model of RTOS based on SystemC. It allows assessing real-time performances and the influence of scheduling according to RTOS properties such as scheduling policy, context-switch time and scheduling latency.

1. Introduction and objective

A real-time system is characterized by its timing constraints in a sense that results or actions must be provided within a specific time window otherwise they are considered to be wrong, whatever their value is [10]. Beyond the correctness of algorithms, checking early on the design process the system's time-related behavior and constraints is essential.

In this paper, we present a solution for modeling and simulating real-time systems including software and hardware parts (co-simulation) with SystemC. Our solution is based on a set of generic C++ classes that model a RTOS.

The choice for working with a System Level Design Language (SLDL) (like SystemC or SpecC) is important for ensuring portability of the model. We chose to work on SystemC [3] because it emerges as the industry standard for transactional-level modeling and system-level design, and co-simulation of mixed HW/SW systems. However, our solution can easily be adapted to another SLDL language like SpecC for example.

As a reminder, SystemC is a language and a simulation kernel based on C++ that allows the representation of software and hardware components and communications at different abstraction levels. It appears as a well-suited answer to the increase of systems' design complexity, by providing an executable model early on the design process.

Since its introduction in 1999, the capabilities of SystemC are evolving; SystemC 1.x has similar modeling capabilities as VHDL and Verilog -intended for system design at the RTL level-, whereas the most recent versions -2.0 and 2.0.1- allow for modeling at the system level [3]. However, up to now, SystemC does not allow describing and simulating real-time applications using a Real Time Operating System (RTOS) dealing with the serialization of tasks on a processor. Therefore, our objective is defining an accurate RTOS model to allow designers of real-time systems to validate their concepts with SystemC. Since there are numerous scheduling policies used by designers, this model should be also generic.

This document is organized as follows. First, we describe the problems to be solved and related solution. Then, we present the main features of our RTOS model. Next, we present two different techniques to implement a RTOS model. The last part presents experiments and results.

2. Problems to be solved and related solution

Generally speaking, a real-time system is composed of a set of tasks, each running a sequential algorithm, and communicating between them with high-level communication mechanisms. With most RTOS, these communication mechanisms are:

- **Synchronization** (based on events or semaphores)
- **Message passing** (based on message queues);
- **Data sharing** (based on global data protected by mutual exclusion).

Our work is based on the MCSE methodology [5] and its functional model which describes a system by a set of functions (e.g. tasks) and relations between them. These

1530-1591/04 $20.00 (c) 2004 IEEE

relations, similar to those of a RTOS, can be of three kinds:

- **Event**: it is used to synchronize functions. Several policies are used to model different behaviors: fugitive (no memorization like SystemC sc_event), boolean (one level of memorization) or counter.
- **Message queue**: it implements a producer/consumer type of relation. Its message capacity is a parameter.
- **Shared variable**: it exchanges data without any synchronization except mutual exclusion.

We use the MCSE functional model for system modeling because it describes at a high abstraction level all features that can be found in a real-time system. However, this work is not specifically linked to this model, and can be applied to any other high level abstraction model.

Simulation of a system at a high abstraction level becomes very important for early design-space exploration. A first level of difficulty resides in simulating the behavior of the system without considering the effect of implementing tasks on processors. This kind of simulation can be easily achieved with SystemC 2.0 and a set of extended classes for example [8]. But this only allows verifying the correctness of the system's behavior and algorithms, because it does not take into account the influence of implementation choices or physical constraints (processor, RTOS, communications network …). However, it is essential to take into account the implementation early on the design process to explore efficiently the design space. For this purpose, it is necessary to simulate the system according to the platform on which it runs (processor, DSP, FPGA …). A SLDL can simulate easily concurrent behaviors, but not the effect of task serialization on a processor when using a RTOS.

In this paper we focus on modeling and simulating the effect of a RTOS on the system's behavior. The objective is to provide results to help designers in their design-space exploration and timing-constraints verification as early as possible on the design process.

Different techniques have been proposed for RTOS simulation. TIMA's team proposes a first technique that consists in using a simulator dedicated to a specific RTOS [2] like WindRiver's VxSim® for VxWorks® [9]. The produced results are very precise, since the simulator is designed to perfectly model the RTOS, but they are also fully linked to a specific RTOS and its API. Thus, the design-space exploration is limited to the possibilities of the selected RTOS.

To avoid this limitation, a highly abstract model of a RTOS can be developed. In [11], such a model (called SoCOS) is presented, but it is based on a specific simulation engine. With this solution, it is difficult to synchronize the simulation of the RTOS part with the simulation of hardware part which may run on another simulation engine (SystemC for example). We presented the same idea in [7].

Another solution consists in developing a RTOS model on top of a SLDL. Gajski's team proposes in [1] a RTOS model which is developed for SpecC. This kind of model allows obtaining results on dynamic real-time behavior without being influenced by technical choices. But for us, this model is related to SpecC and does not model RTOS preemption with enough time accuracy since its precision depends on the model's clock accuracy. The solution we present in this paper is close to Gajski's team proposal [1] but it uses SystemC and provides a time-accurate preemption model of RTOS independent from any clock considerations. Our solution is also already integrated into a global tool [12] for performance analysis developed and sold by CoFluent Design. It can be considered as a mixture of the virtual OS and aggregate timed model of OS also proposed by TIMA in [4].

In the next section, we present a description of the RTOS behavior and then, the model we propose.

3. The RTOS model

A RTOS is not only defined by its behavior (scheduling algorithm), but also its timing properties, i.e. contribution to the time evolution of a set of tasks. A global system timing parameter is called the RTOS overhead. In our solution we accurately model both the RTOS behavior and its timing properties.

3.1 RTOS behavior modeling

The priority-based preemptive scheduling policy is the most widely used, but there are many others [10] and some are developed specifically for an application. At a high abstraction level, we can characterize a RTOS behavior by two parameters:

- The **scheduling policy**: it defines the RTOS algorithm used to select the running task among the ready tasks. It can be based on task priorities or deadlines for example.
- The **preemptive/non-preemptive mode**. A preemptive RTOS can suspend a running task between two of its RTOS calls (hardware interrupt occurrence for example). This parameter is very important in RTOS modeling since it can delay or not (non-preemptive) the consideration of external events.

Our model considers these two parameters. Several scheduling policies are implemented but since we cannot implement all specific ones, designers can also define their own policies by overloading the *SchedulingPolicy* method of our *Processor* class.

While the scheduling policy is set at the simulation's start and cannot be dynamically changed, the preemptive/non-preemptive mode can be changed during the simulation. This enables to model critical regions during which task preemption is not allowed.

3.2 RTOS timing modeling

The RTOS overhead may be neglected if it is very much smaller than tasks' durations. But it is not always the case. RTOS overhead depends mainly on 3 parameters:

- The **scheduling duration**: it characterizes the time spent by the RTOS to select a ready task. This duration depends not only on the scheduling algorithm, but also on the number of ready tasks when the algorithm runs.
- The **context-load duration**: it characterizes the time required by the RTOS to load the context of the running task (processor registers).
- The **context-save duration**: it characterizes the time required by the RTOS to copy the context of the suspended task from the processor registers to the memory.

In our model, these three parameters are modeled and can be fixed or defined by a user formula computed during the simulation according to the current state of the simulated system (number of ready tasks for example).

To summarize our solution, a system is modeled by a set of C++ objects that inherit from the processor and function classes defined in our model. Figure 1 illustrates a part of the UML class diagram of our solution.

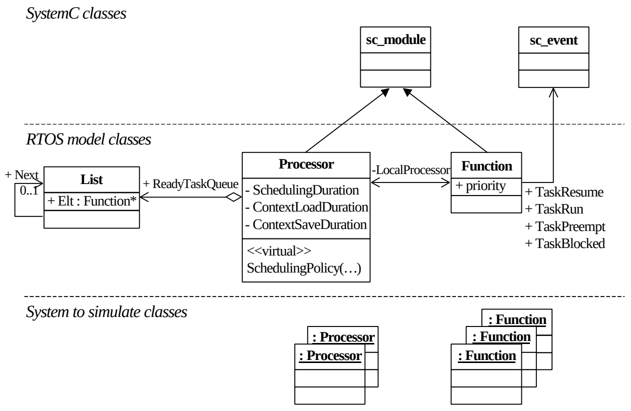


Figure 1 – UML class diagram.

To provide the SystemC code that models the system, we have enhanced our SystemC code generator presented in [8]. This code generator is able to automatically provide an executable model including functions and processors in a few seconds.

4. RTOS model implementation

First we present a task scheduling implementation based on a thread that simulates the behavior of the RTOS. For us, this first approach seemed natural in its principle. After analyzing the pros and cons of this first solution, we finally present a second approach based on the definition of a set of RTOS procedures called by tasks as they really do with a real RTOS.

Before presenting our approaches, let us remind that each task running on a RTOS can be in only one of the following states at each moment [10]:

- **Waiting**: waiting for a synchronization;
- **Running**: running on the processor;
- **Ready**: waiting to be selected by the RTOS to enter the Running state.

4.1 Task scheduling using a dedicated thread

For our first solution, the behavior of each task implemented on a processor is implemented by a SystemC thread. The behavior of the RTOS is also modeled by a SystemC thread. Each task is implemented by a C++ object which uses four SystemC events (TaskResume, TaskRun, TaskPreempt and TaskBlocked) which the thread can wait on according to the task's current state.

The RTOS thread waits on a SystemC event (RTKRun). The simulation technique consists in controlling the evolution of all these threads by using the SystemC events so that at any moment only one thread is running. The time evolution of the system and RTOS is based on a delay procedure similar to the one presented by TIMA's laboratory [2]. Each task in the Ready state is referenced in the RTOS ReadyTaskQueue.

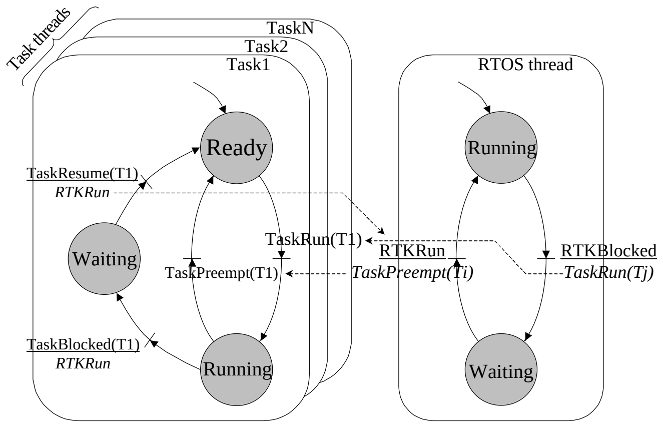


Figure 2 – Task states and RTOS states.

Figure 2 presents the possible different states of a task and of the RTOS and the synchronization links between these threads. During the simulation, system tasks notify the RTOS thread when they enter or leave the Waiting state. Then the RTOS thread runs the scheduling algorithm and decides what task in its *ReadyTaskQueue* must be activated and then notifies it by its *TaskRun* event. Figure 3 shows an example of thread switching to simulate two tasks running on a RTOS.

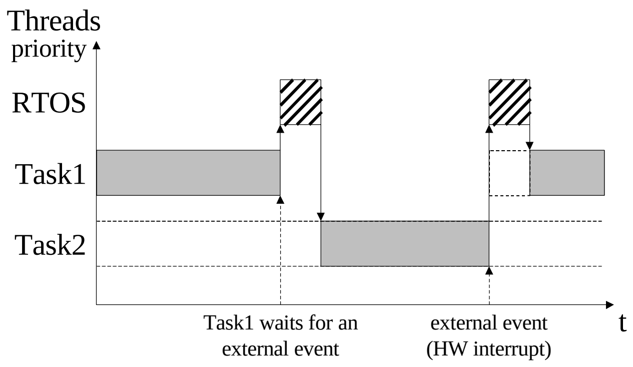


Figure 3 – Task scheduling with a RTOS thread.

4.2 Task scheduling using procedure calls

Our first solution implies many SystemC thread switches (between tasks and RTOS). This has a huge effect on the simulation duration. So we optimize the RTOS model implementation by removing the RTOS thread, without altering the model's possibilities. This solution is close to the real implementation of a RTOS which is based on a set of procedures (called primitives) [10].

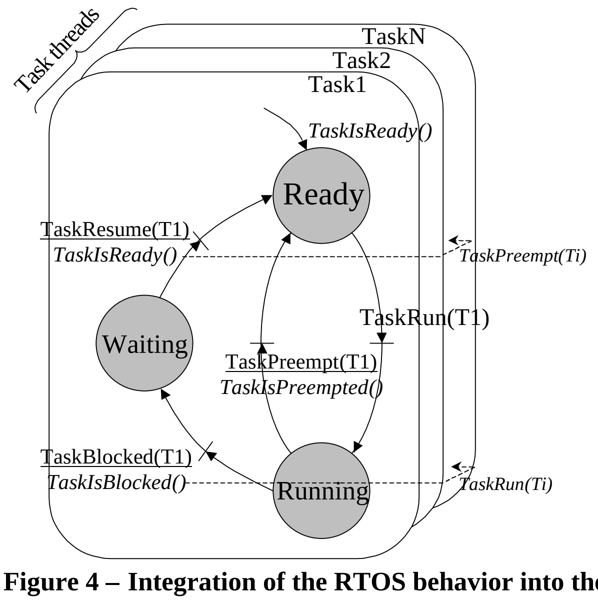


Figure 4 – Integration of the RTOS behavior into the behavior of tasks state transitions.

In this second solution, we use one thread per task; the RTOS is implemented by a C++ object with a set of methods, but without using a thread (see Figure 4). Each task notifies the other ones by using methods of the RTOS object.

The RTOS object has three main methods used by the tasks:

- **TaskIsReady()**: it is called when the task enters the Ready state. If the scheduling policy allows the ready task to preempt the running one, then the ready task sends the *TaskPreempt* event to the running task.
- **TaskIsBlocked()**: it is called by a task that enters the Waiting state. The scheduling algorithm must select another task to run and notifies it with the *TaskRun* event.
- **TaskIsPreempted()**: this method is called by the running task when receiving the *TaskPreempt* event. It computes the remaining time for completing the current operation.

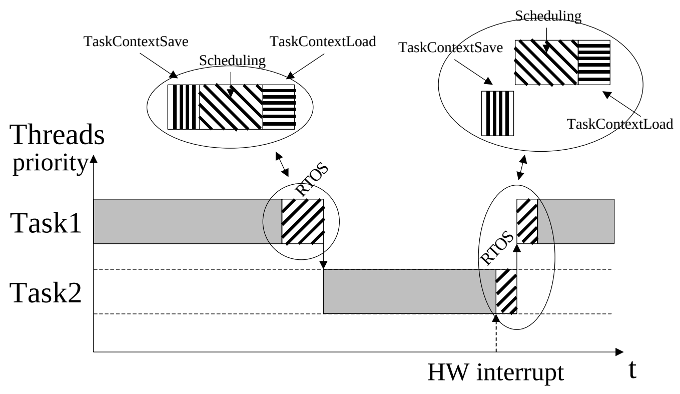


Figure 5 – Task scheduling without a RTOS thread.

Figure 5 shows the thread's behaviors for this solution. We can observe that fewer thread switches occur than in the previous solution and that the RTOS algorithm is executed by the thread of the running task which becomes blocked and by the thread of the task which was awaked. Figure 5 also presents the RTOS overhead decomposition in three basic times which correspond to the context save, scheduling and context load operations of a RTOS.
To conclude on these two approaches, we think that the use of a thread dedicated to the task scheduling makes the modeling of some scheduling policies easier, like modeling the Time Sharing algorithm for example. However, this approach increases the simulation duration since there is a context switch for each call to the scheduler and each return, what is not the case when we use procedure calls. In the case of using procedure calls, the only thread switches are those of the tasks of the system we're designing that occur.

5. Experiments and results

We developed a tool [8][12] that allows users to capture the model of real-time systems, and obtain automatically an equivalent SystemC model including our RTOS model for its behavioral verification. Results can be displayed in different ways [6][8]. The most interesting diagram that let users observe the influence of the RTOS is the *Timeline* chart.

A *TimeLine* chart displays the task's states and interactions with various types of relations (read, write, signal…). A vertical arrow represents a task accessing a communications link and the arrow style informs on the kind of access (read, write). Each horizontal line represents the state of each task with a different style and color (Creation, Running, Destruction, Waiting for processor availability (Ready), Waiting for a synchronization (Waiting), Waiting for resource).

From such a display tool we can get much information such as synchronization sequences between tasks and the time spent by a task waiting for synchronization or for mutual exclusion. Therefore, from a *TimeLine* chart we can extract performance results that are useful to define the best architecture to implement the application. On a *TimeLine* chart, we can make also time measurements to verify time constraints of the system. As an example, we can measure the time spent between an external event and the system's reaction.

Figure 6 presents a part of a *TimeLine* chart produced by our tool. To make understanding easier, we added on the figure references and time durations, as these measurements can be easily done with our tool.

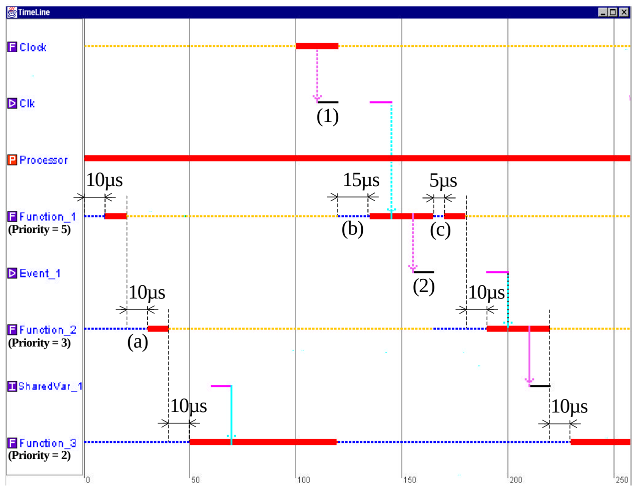


Figure 6 – Example of a TimeLine result.

The system is composed of a hardware task named *Clock* and of three software tasks named *Function_1*, *Function_2* and *Function_3* running on the same *Processor*. Here, tasks are scheduled by *priority-based preemptive scheduling*. First, we can observe the scheduling of the tasks at the beginning of the simulation: *Function_1*, *Function_2* and *Function_3* are executed sequentially. Next, we can see *Function_1* preempting *Function_3*. On (1), *Clock* notifies the event *Clk* and so awakes *Function_1*. Then *Function_1* preempts *Function_3* and starts its processing. During its execution, *Function_1* sends the *Event_1* event (2) and awakes *Function_2*. *Function_2* does not preempt *Function_1* because it has a lower priority. When *Function_1* ends, *Function_2* starts its processing. Finally, when *Function_2* ends, *Function_3* resumes its execution where it was preempted. From this display, we can observe also the RTOS overheads. Here, we have defined a RTOS that has a *SchedulingDuration*, a *TaskContextLoad* and a *TaskContextLoad* that all equal to 5µs. Then, we can observe the overhead duration when a task ends and when a task is resumed (a), when a task is preempted (b) or when a task is not preempted (c).

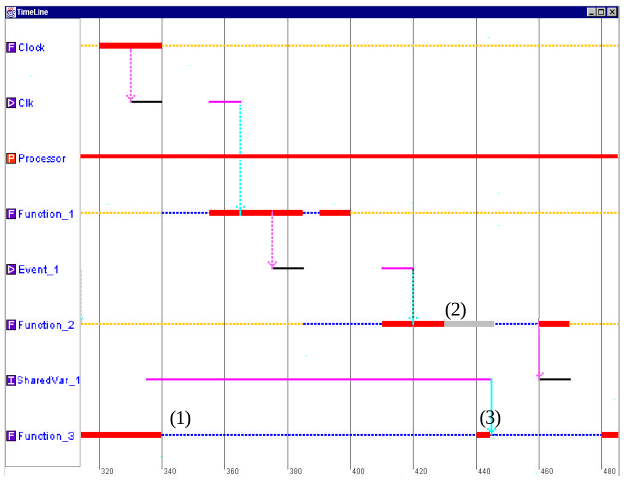
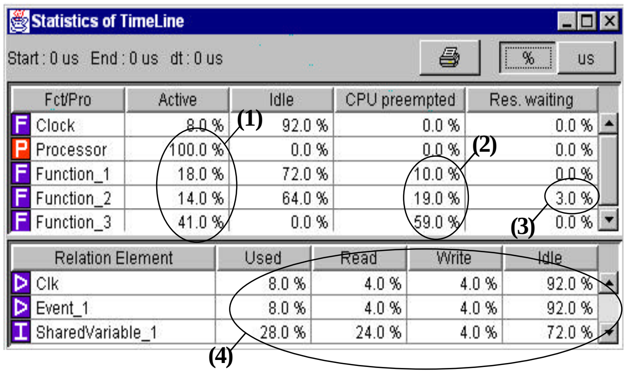


Figure 7 – Example of mutual exclusion blocking.

Based on the simulation of the same application presented before, figure 7 presents a blocking mutual exclusion situation. On (1), *Function_3* is preempted by *Function_1* during a read operation of the *SharedVar_1* shared variable. On (2), *Function_2* is blocked, waiting for the *SharedVar_1* resource. Then *Function_3* resumes its execution after an overhead duration. On (3), *Function_3* releases the *SharedVar_1* resource and is preempted by *Function_2* which has a higher priority.

This "priority inversion" problem can be avoided by disabling preemption during access to shared data [10]. With our RTOS model, this behavior can be modeled. Designers can easily check the need or benefit of such a solution for their system.

The timeline display tool allows a detailed analysis of the behavior of a real-time system, but it is not easy to get a global view on the system. We can provide statistics for the whole simulation from a global analysis of the system's simulation.



Fct/Pro	Active	Idle	CPU preempted	Res. waiting
Clock	8.0 %	92.0 %	0.0 %	0.0 %
Processor	100.0 %	0.0 %	0.0 %	0.0 %
Function_1	18.0 %	72.0 %	10.0 %	0.0 %
Function_2	14.0 %	64.0 %	19.0 %	3.0 %
Function_3	41.0 %	0.0 %	59.0 %	0.0 %

Relation Element	Used	Read	Write	Idle
Clk	8.0 %	4.0 %	4.0 %	92.0 %
Event_1	8.0 %	4.0 %	4.0 %	92.0 %
SharedVariable_1	28.0 %	24.0 %	4.0 %	72.0 %

Figure 8 – Example of statistics from a TimeLine.

Figure 8 presents an example of statistic results obtained for the application we have previously presented. We can observe the activity ratio of each task on a processor (1), their preempted ratio (2) and the ratio when waiting on resources (mutual exclusion) (3). We can also get statistics concerning the communications like their utilization ratio (4).

All these results are presented for a very simple system in this paper. We have also used our simulation model and display tools to explore the design space of a more complex application such as a video MPEG-2 compressing and decompressing SoC. The system is composed of 18 tasks implemented on six processors, three of them are software processors with a RTOS model.

6. Conclusion and future work

In this paper, we propose a generic RTOS model for simulation of real-time systems at a high abstraction level with SystemC. This model is implemented as a set of C++ classes which run on top of the SystemC 2.0 simulation engine of and does not require any modification in the language. This solution allows co-simulating with SystemC hardware and software parts, including our RTOS model and application tasks.

Compared to other existing models, our model allows an accurate RTOS time representation based on three parameters (context load and save durations and scheduling algorithm duration). They allow to analyze the effect of processor change (context load and save durations) and of RTOS change (scheduling algorithm duration) early in the design space exploration. Our model also accurately depicts task preemption by a hardware event without adding any delay due to simulation technique.

Thus, it is very easy to explore the design space of real-time systems implemented on SoC composed of several processors and FPGA and obtain accurate results. Our solution is already implemented and available in a commercial product named CoFluent Studio™ [12].

Our technique is close to the real implementation of a multitask application running on a RTOS. This approach has been selected for simulation efficiency reasons, but also to ease software generation for a final implementation using commercial RTOS. This software generation is a goal of our future work. Another improvement we can imagine now is automatic verification of timing constraints by simulation after setting these constraints in the initial system model.

7. References

[1] A Gerstlauer, H Yu and D D Gajski, UC Irvine, US, "RTOS Modeling for System Level Design", DATE 2003

[2] S Yoo, I Bacivarov, A Bouchhima, Y Paviot, and A A Jerraya, TIMA Laboratory, FR, "Building Fast and Accurate SW Simulation Models Based on Hardware Abstraction Layer and Simulation Environment Abstraction Layer", DATE 2003

[3] "Functional Specification for SystemC 2.0", available at www.systemc.org

[4] S Yoo, G Nicolescu, L Gauthier, A A Jerraya, TIMA Laboratory, FR, "Automatic generation of fast timed simulation models for operating systems in SoC Design", IEEE 2002

[5] J.P. Calvez, Embedded Real-Time Systems. A Specification and Design Methodology, John Wiley, 670 pages, 1993

[6] J.P. Calvez, O. Pasquier, Performance Monitoring and Assessment of Embedded Hw/Sw Systems. In "Design Automation for Embedded Systems", Vol 2, Kluwer Academic Publishers, 1997

[7] O. Pasquier , J-P. Calvez, "An object-base executable model for simulation of real-time Hw/Sw systems", Proceedings of DATE 99, March 1999, Munich.

[8] R. Le Moigne, O. Pasquier, J-P. Calvez, "A Graphical Tool for System Level Modeling and Verification with SystemC", FDL' 03, September 2003, Frankfurt.

[9] VxWorks® and Wind River Systems products available at www.windriver.com.

[10] G. C.Buttazzo, "Hard Real-Time Computing Systems", Kluwer Academic Publishers 2002, ISBN 0-7923-9994-3

[11] D. Desmet, D. Verkerst, H. De Man, "Operating System Based Software Generation for System On Chip" in DAC, June 2000.

[12] CoFluent Studio^{TM}, available at www.cofluentdesign.com.