POSIX modeling in SystemC

Hector Posadas, Jesús Ádamez, Pablo Sánchez, Eugenio Villar
Microelectronics Engineering Group, University of Cantabria, Santander, Spain
www.teisa.unican.es/gim

Francisco Blasco
DS2, Valencia, Spain

Abstract - Early estimation of the execution time of Real-Time embedded SW is an essential task in complex, HW/SW embedded system design. Application SW execution time estimation requires taking into account the impact of the underlying RTOS. As a consequence, RTOS modeling is becoming an active research area. SystemC provides a framework for multiprocessing, HW/SW co-simulation at several abstraction levels. In this paper, a SystemC library for POSIX modeling and simulation is presented. By using the library, the SystemC specification using POSIX functions is converted automatically into a timed simulation estimating the execution time of the application SW running on the POSIX platform. The library works directly on the source code. Therefore, it provides an early and fast estimation of the performance of the system as a consequence of the architectural mapping decisions. Although accuracy is lower than when using lower-level techniques, it supports high-level design-space exploration as simulation time is significantly less than RT (ISS) simulation[1].

I. Introduction

Cost of design has been identified as the greatest threat to the continuation of microelectronic technology improvement towards larger integration scales. Among the different costs, embedded software development represents the major part of the SoC development cost [1]. In this context, there is a clear need for new methodologies supporting efficient design of Real-Time, Embedded (RT/E) systems on complex platforms [2].

Performance analysis is an increasingly important and challenging task in embedded system design since performance parameters (time, size, consumption, cost, etc.) can be as important as functional requirements [3]. Complex, HW/SW embedded systems demand accurate estimation of their timing characteristics before implementation. Such system timing analysis requires system modeling taking into account the close interaction between the application SW and hardware-dependent SW running on the different processors and the application-specific HW through the platform communication resources [2-3].

Time execution estimation has been a traditional problem in real-time embedded SW engineering [4]. Execution time figures are necessary to develop timed SW simulation models [5].

Accurate SW simulation requires taking into account the effect of the RTOS [6-12]. In order to be efficient and, therefore, applicable to complex systems, the SW code can be directly executed using an abstract model of the RTOS. System specification languages like SpecC [6-7] or SystemC [10][12-13] have proven to provide a useful HW/SW co-simulation platform.

SystemC can be used to effectively create accurate models of complex, HW/SW embedded systems. SystemC allows the hardware and software design team to develop an executable specification of the system which can be used to quickly simulate and explore various algorithms, and validate and optimize the design [14]. SystemC supports efficient generation of the embedded software including interface drivers and the RTOS [15]. Nevertheless, certain RTOS characteristics such as priority-based preemption are very difficult to be adequately modeled in SystemC.

In this paper, a high-level, POSIX simulation library in SystemC is presented. The library allows the designer a fast, sufficiently accurate, timed simulation of the application SW running on top of POSIX [16]. As most current RTOSs support this standard, the library is portable to different development frameworks. Moreover, SystemC provides a flexible infrastructure for multiprocessing, HW/SW co-simulation at different abstraction levels. As a consequence, the POSIX simulation library can be used in any SystemC HW/SW co-simulation environment.

The structure of the paper is the following. In the next section, the contribution of the paper is described in the context of related work. In Section 3, the POSIX modeling and simulation methodology is presented. Experimental results are provided in Section 4. Finally, the main conclusions of the work are presented.

II. Related work

Traditionally, simulation capability has not been a main criterion in selecting a RTOS [17]. Nevertheless, most RTOS vendors provide a simulator of their OS to help software designers to develop and emulate application code on the host before having developed the actual hardware prototypes [18]. These simulators usually take the application code, compile it with a native compiler, link it with the OS Kernel libraries and produce a host executable. The simulator is completed with the libraries of the additional RTOS modules (TCP/IP stack, file system, link handler, etc.) in the simulated

[1]This work has been partially supported by the ITEA IP 03002 Medea project and the TIC-2002-00660 project.

0-7803-9451-8/06/$20.00 ©2006 IEEE.

environment. Simulation cannot be timing accurate because it relies upon the host operating system scheduling. Although it can guarantee the order of the events, the simulation is untimed. Very few commercial RTOS simulators are timed simulators [19]. They are based on proprietary languages and do not support HW/SW co-simulation.

Architectural exploration of complex, HW/SW embedded systems require accurate profiling of their timing characteristics. Precise co-simulation at the RT level is unfeasible for system-level profiling due to its excessive host execution time. SW simulation using cycle-accurate Instruction-Set Simulators (ISS) only alleviates the problem. Moreover, it requires setting up a co-simulation infrastructure [9]. Faster co-simulation can be achieved by directly executing the code in a SW-HDL co-simulation environment [20]. With this approach, the SW execution time is very difficult to model. A rough approximation of the SW execution time can be achieved using the native system clock [11].

Fast, sufficiently accurate, HW/SW co-simulation requires adequate modeling of the RTOS in a system-level language like SpecC [6-7] or SystemC [10][12]. The RTOS model is abstract, based on APIs including the most common RTOS functions [7], wrappers [8] or an implementation-specific model of each channel [10]. An advantage of these techniques, in addition to the improvement in simulation time, is that they can be applied early in the design process avoiding costly design iterations. As these techniques use an abstract RTOS model, they cannot simulate embedded code using actual functions of any specific RTOS. As a consequence, the simulation methodology does not support refinement of the original specification apart from the inclusion of the RTOS modeling itself [7]. Moreover, the application code has to be modified with specific RTOS modeling functions [7-8]. Only by direct modeling of a standard RTOS it is possible to use the same code for both simulation and implementation [11-12].

Execution time of both the application code and the RTOS is taken into account during simulation by introducing 'wait' statements [5-12]. More the number of 'wait' statements introduced, higher the accuracy. Nevertheless, the 'wait' statements are introduced at certain static, predetermined points. As a consequence, low-level, dynamic timing characteristics of the RTOS like time-slicing, preemptive scheduling, interrupts and exceptions are very difficult to model.

A SystemC library called PERFidy was developed for system-level, timed simulation and performance analysis [10]. The library is able to obtain timing cost estimations from a set of values that characterize the target platform in which the embedded systems will be implemented. This is made possible by redefining all the C++ operators with new ones with the same functionality but estimating the corresponding execution time on the chosen platform resource. This detailed estimation of the SW execution time allows deciding dynamically the place of the application code where the task (process or thread) has to be preempted. As a consequence, the low-level, dynamic characteristics of the RTOS can be efficiently modeled.

In this paper, an extension of the timed-simulation methodology used in PERFidy is proposed in order to allow the timed-simulation of RTOS functions in SystemC. Although the SystemC 2.0 simulation kernel executes processes following a non-preemptive scheduling policy without priorities, the proposed simulation methodology is able to model preemption as well as different scheduling policies based on priorities. In order to ensure independence from any specific RTOS, the simulation methodology supports POSIX, thus ensuring wide portability. In this way, the proposed extension, called PERFidiX supports the early performance analysis of the system specification as well as the impact of the decisions taken during design refinement. Using SystemC ensures a flexible, portable framework for multiprocessing, HW/SW co-simulation.

III. POSIX modeling in SystemC

A. Analysis of the POSIX standard

The POSIX standard covers a wide range of system facilities. As the scope of this work is embedded systems, the focus of interest of the paper is the Real-Time extension. Concurrency is supported both by processes and threads. Three standard scheduling policies are defined: FiFo (FF), Round Robin (RR) and Sporadic Server (SS). Furthermore, other policies can be offered but they are implementation specific. The standard also defines several mechanisms for communication and synchronization. Apart from shared variables, they are mutexes, conditional variables, message queues, semaphores, sockets, streams, signals, etc. Timing facilities are based on timers and real-time clocks. Traces are supported for system analysis and verification. Additionally, file systems and memory management are also included.

Real-time requirements define functions to support source portability of applications. The presence of many of these functions is dependent on support for implementation options described in the standard. The specific functional areas included in this section include the following:

- Priority Scheduling
- Semaphores
- Timers
- Real-Time Signal Extension
- Synchronized Input and Output
- Inter-process Communication
- Process Memory Locking
- Memory Mapped Files and Shared Memory Objects

The first six functional areas require an appropriate modeling in order to ensure a correct timed and functional simulation of the code. As the simulation runs over one process of the host OS, the modeling of the last two features using the host memory is not straightforward. The influence of these facilities in the execution time will be included in the simulation, but the development of SystemC functions that implement this functionality is currently under study.

The model assumes the correct memory access of the code.

B. POSIX simulation library

The correct simulation of all the POSIX functions described above requires extending the SystemC simulation kernel with additional functionality not covered by PERFidy. This modification does not affect the original SystemC simulation kernel. This means that the standard kernel has not been modified; the required functionality is emulated over the original one by using its facilities. This increases the portability of the library, as it does not require the installation of a completely new version of SystemC; the standard one can be used. Moreover, the library is external, so the use of future SystemC versions will produce no problems since compatibility with previous versions will be ensured.

The PERFidiX library including all facilities described above contains three different parts. The first one contains the execution support that is provided over the original SystemC kernel. It implements process and thread management, including scheduling capabilities, and timing facilities. The second part implements the POSIX API by using the facilities provided by the new execution support kernel. The last part carries out the performance analysis task by using the timing and traces facilities, and reusing the technology of the original PERFidy library [10].

C. Concurrency modeling

Parallelism is modeled by using the SC_THREAD process of SystemC. Therefore, both POSIX processes and threads are modeled in the same way. Thus, the library implements the required actions that give each element its own characteristics. The characteristics of processes and threads are loaded in a list when they are created and these parameters can be modified during simulation using the methods the POSIX standard defines. However, modeling the capabilities derived by the use of separate memory spaces in SystemC is not straightforward[2].

SystemC does not allow dynamic thread creation. In order to support dynamic thread creation, a thread-pool is initialized with the simulation. This pool has a predefined number of SC_THREADS (the number can be modified in the source code) starting in a blocked state. Each time a new thread is declared, a thread in the pool is resumed. This means that there are a maximum number of dynamic threads that can be executed at the same time.

D. Modeling the scheduler

The SystemC underlying kernel activates in each δ-cycle all the threads that are not blocked without any considerations about priorities. However, the scheduler model running over the SystemC kernel has to ensure that only one thread is executed in each processor each time. To solve the problem it is necessary to ensure that all threads remain blocked, except the one with the greatest priority, which is awoken. It is also necessary to manage preemption and priorities. As many schedulers are modeled as processors are defined, and each process is assigned only to one scheduler.

Following the POSIX standard, there is one thread list for each priority. A runnable thread is placed on the thread list for that thread's priority when it is ready to execute. When a thread reaches a blocking point or its time slot is consumed, it releases the processor, and in the second case, it is added to the corresponding list. Then, the first thread of the first non-empty list is deleted from there and resumed.

However this approach does not model preemption as a SC_THREAD is executed until a wait statement is reached. POSIX defines that a thread must be preempted when a thread with a higher priority is awoken. In order to model preemption adequately, a new list is instantiated. This list is similar to the previous one, but it does not contain the threads ready to execute, but the threads that are under execution. As a consequence, the execution state has two parts. The first one is the functional execution, and the second one is the temporal execution. That is, the code is executed in zero time (in the simulation) and then the thread is slept to take up the corresponding time in the processor. This placement in time is produced before a data exchange is made. If during the time the thread is slept, another thread with higher priority is awoken, it is executed, and it indicates to the other thread the time it has been preempted. After checking if preemption has taken place, and it is slept again during this time. As a result, when the data exchange is made, the execution state is correctly placed in time. This implementation needs another list including the processes in the execution state, because preemptions can be chained. Considering that global variables and signals are generated only in the same processor where they are delivered, it is possible to support preemption and delivering of signals just by placing in time the thread execution.

As commented above, only software timers, channel accesses (e.g. a mutex unlock) or signals could resume threads. Moreover, only a thread, a timer or an external interruption can generate a signal. Then, the time when the events will be produced is known except for external interruptions or channel accesses. The events generated by the software are known in advance. Thus, the way to model preemption is to place in time not only channel accesses but also these temporal points, and then resume the adequate thread. This means that all accesses to global variables can be done in the right order and no errors are made.

Correct timed simulation of the SW code including the RTOS requires stopping the running thread each tick-time interval. In this way, any expected or external event can be taken into account. In Figure 1, an example is used to show the result when using the proposed solution. The system is composed of three threads:

Thread 1 presents the lowest priority and a Round Robin policy with a time limit of 20μs. The thread is ready in T=0.

Thread 2 has a medium priority and a FIFO policy. The

[2]This feature should be taken into account in embedded processors with MMU. Certain specific functions are still under study.

thread is blocked with a timeout that expires at T=30μs.

Thread 3 has the highest priority and FIFO scheduling policy. It is blocked awaiting a hardware interruption (a POSIX signal).

Although PERFidiX takes into account the execution time of the OS, it will not be considered here for the sake of simplicity. At T=0 only thread 1 is ready and it is moved to execution state. Two time events have to be considered, the time limit (T=20μs) and the timeout expiration (T=30μs). Thus, the thread can be executed for 20μs (or until another event is detected such as a channel access, a new timer, etc.). Then, the thread code is executed until PERFidiX estimates that the executed code will take 20μs in the target platform. This is done in zero time in the SystemC simulation. To make the simulation take into account the estimated execution time, the thread is slept until T=20μs and then the process is moved to the ready state and the scheduler is awoken. In this way, the actual behavior of the processor is closely modeled.

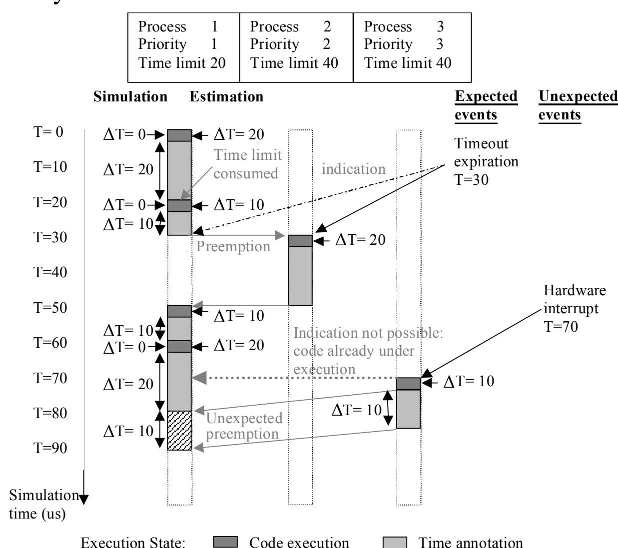

Figure 1: Preemption modeling in PERFidiX.

At T=20μs only thread 1 is ready so it is executed. However, this time it can be executed during 10μs because at T=30μs the timeout expires. Then the same procedure described previously is followed. At T=30μs, thread 1 has used 10μs of the time interval and another 10μs remains. However, thread 2 is ready and has a higher priority, so thread 1 has to be preempted, and thread 2 resumed. Thread 2 execution takes 20μs until it is slept (no events are expected) at T=50μs. At that time, thread 1 is resumed and it finishes the 10 μs of the time limit. At T=60μs it is moved to the ready state and the scheduler is awoken. However, as there are no other threads in this state, it is resumed with a new time limit, until T=80μs. Nevertheless, at T=70μs a hardware interruption is produced and thread 3 is ready with highest priority, so thread 1 should be preempted. Thread 1 has scheduled its finishing event at T=80μs while, as a consequence of the execution of thread 3, its actual finishing time is T=90μs. To correctly model this behavior, thread 3 is executed at T=80μs with a finishing event scheduled at T=90μs and then, thread 1 variable where preemption is indicated, is incremented in 10μs. As a consequence, at T=80μs, thread 1 is slept again during the time the preemption variable indicates and it is set to 0. Results (as well as the implementation) are deterministic if no global variables are used from T=70μs to T=90μs in both thread 1 and thread 3. As explained above, threads that are awoken by unexpected signals should not make use of global variables without including additional synchronization mechanisms.

As explained before, this entire model does not modify the SystemC kernel. It is based on the use of "wait()" and "notify" SystemC primitives. The scheduler also provides the functions to model the blocks produced by communication and synchronization POSIX facilities.

F. Modeling Signals

Once the scheduler is implemented, signals can be modeled as defined by the POSIX standard. The signal manager can access the scheduler to allow all blocking communications to implement signals that mean that a thread can be stopped or unblocked independently of the cause that produced this blockage. There is a SystemC thread used only by the signal manager that will execute the actions related to signals that have to be delivered, since no other process can execute them.

G. Clocks and timers

The POSIX real-time standard requires the implementation of clocks for each process and thread, and for the whole simulation. Timers, sleep facilities and alarms are defined by using these clocks. The values of the clocks are updated and the execution time estimated by PERFidiX for each code segment. The actions of the elements declared over them, are executed by adding the time each event will take to the events list of the scheduler.

The elements that depend on the real-time clock of the system have been implemented in a different way. With this purpose, a SC_THREAD has been defined that is slept until the next event of that clock is required.

*H. POSIX Interface modeling*POSIX services are provided in three different ways. Some of them use the underlying host functions, others are completely new, and those that depend strongly on the hardware platform have to be adapted to model correctly its platform-dependent functionality.

If the OS of the host computer is POSIX based, such as UNIX or Linux platforms, some of the host POSIX functions can be reused. These functions are basically those that are platform independent. Mathematical functions, string management, etc., maintain their functionality in every platform and they do not interfere with the scheduler or the parallelism capabilities of the system. Thus, they can be used to model, at least, the platform functionality. To include the timing cost, these functions will be wrapped into

new functions that will take into account the time the function will take in the final processor.

The second group of the API functions is composed of those facilities that allow the designer to interact with the elements that have been implemented in the software execution support described below. Parallelism, scheduling, communication, synchronization and timing features are completely platform dependent, so new implementations are required. They have been developed in two different ways. Some of them provide access to the software support but they have no functionality implemented inside. These are the functions used to indicate to the kernel the characteristics of threads and processes, to activate the timers, to obtain data from the system clocks, to indicate to the signal manager the generation of a new signal, etc. The other functions implement their own functionality using the facilities provided by the simulation kernel. Mutexes, semaphores or message queues use the kernel facilities to block or resume threads, generate signals or implement timeouts.

The last group of POSIX API functions is composed of those functions whose implementation is strongly dependent on the hardware platform. Thus, a general platform execution support model is not possible. Some examples are the I/O functions, which strongly depend on the system drivers, so the implementation cannot be reusable on different platforms. Therefore, PERFidiX cannot provide accurate implementations that model all platform implementations. Instead of that, models that allow the designer to simulate the functionality are provided. In some cases, the host equivalent functions can be used to simulate it by adding some parameters that model their timing characteristics. In other cases new functions have to be developed because the host versions can interfere with the parallelism and scheduling capabilities of the platform model. In this context, files and sockets are in the same situation.

I. Traces

The original PERFidy library implements a set of functions to allow the designer to indicate and get the data required in order to obtain an adequate performance analysis. Now this set has to be extended to implement the functionality described in the POSIX standard to achieve this goal. First, the events are traced in POSIX standard to two classes: User trace events that are generated by the application of instrumentation functions and system trace events, which are generated by the operating system. Each trace event of the latter group may be an implementation-defined action such as a context switch, or an application-programmed action such as a call to a specific operating system service (for example, *fork*()) or a call to *posix_trace_event*().

IV. Experimental results

PERFidiX has been evaluated in a sufficiently complex case study, the EN 301 245 vocoder for GSM applications standard of ETSI [21]. The original, sequential, standard golden model was structured in 13.500 lines of SystemC code. The SystemC specification was refined into a POSIX version, closer to the final implementation.

Experimental results are shown in Table I. The first column shows the estimated execution times for each thread and the RTOS obtained with PERFidy from the system-level, SystemC specification (SCS). The second column shows the corresponding estimated execution times obtained by PERFidiX from the POSIX code (PP). The third column shows the actual execution times obtained from the implementation of the code on an OpenRisk 1500 platform [22]. Execution times have been obtained directly from the prototype board inserting counters for each thread directly in the RTOS (eCos) code.

TABLE I
Estimated and actual execution times of the experiment

Thread	SCS (ms)	PP (ms)	IP (ms)
pre_filtering	1,085.10	1,035.60	1,061.16
homing_frame_test	84.72	73.76	73.12
frame_lsp_func	12,774.98	11,955.65	11,901.95
frame_int_tol_fun	12,308.99	11,646.93	11,246.62
subframe_coder_fun	40,004.07	37,585.17	37,498.38
serializer_fun	158.24	184.44	173.41
vad_comp_fun	10.89	11.82	11.44
CN_encoder_fun	54.74	61.85	58.72
sid_encoding_fun	223.32	249.53	269.15
RTOS	7,021.70	7,721.35	7,581.67

The corresponding error percentages are shown in Table II. As expected, the error obtained on the refined, POSIX code is smaller than the error at the original SystemC code. This indicates the accuracy of the technique in estimating the time slicing and the context changes. In any case, the error is kept smaller than 8%.

TABLE II
Estimation accuracy

Thread	SCS (%)	PP (%)
pre_filtering	3.45	2.41
homing_frame_test	13.67	0.88
frame_lsp_func	4.51	0.45
frame_int_tol_fun	0.17	3.56
subframe_coder_fun	3.04	0.23
serializer_fun	6.72	6.36
vad_comp_fun	9.36	3.34
CN_encoder_fun	7.45	5.33
sid_encoding_fun	12.18	7.29
RTOS	3.94	1.84

As shown in Table III, the increase in simulation time

implied by PERFidiX is negligible with respect to PERFidy. In any case, the gain with respect to a cycle accurate ISS is large (77 times faster).

TABLE III
Simulation times

	SCS (ms)	PP (ms)	ISS (ms)
Simulation time	1,070	1,560	124,000

V. Conclusions

Like other recent contributions in the field, the paper shows that a high-level modeling and timed simulation of application SW is possible at the source code level including the chosen RTOS. The main advantage of the proposed underlying technology over other similar techniques is that it avoids the complex, three-step process of estimating the execution times, annotating them in the appropriate points of the code and simulating. This technology was proven effective in PERFidy, a SystemC performance analysis library. In the paper, this technology is applied to the modeling and simulation of POSIX. The new library models the concurrency, communication and synchronization functionality required by the standard without modifying the original SystemC simulation kernel. This ensures full portability.

The new library, called PERFidiX, supports refinement of the original SystemC specification by optimizing the code including POSIX functions. Reusability is also improved as now it is possible to include legacy code and COST components making use of POSIX functions.

SystemC has been shown to be a flexible framework for system specification, refinement and simulation at different abstraction layers.

Experimental results assess the execution time estimation capability of the library. Although some error is unavoidable, accuracy is enough to allow the designer a fast and early performance estimation of the system taking into account the architectural mapping decisions, but avoiding the need for HW/SW synthesis.

References

[1] ITRS. International Technology Roadmap for Semiconductors: 2003 Edition. http://public.itrs.net.

[2] A.A. Jerraya, S. Yoo, D. Verkest and N. When: “Embedded Software for SoC”, *Springer*, 2003.

[3] A. Sangiovanni-Vincentelli and G. Martin: “Platform-based design and software design methodology for embedded systems”, *IEEE Design and Test of Computers*. November-December, 2001, 23-33.

[4] P. Puschner and C. Koza: “Calculating the maximum execution time of real-time programs”, *The Journal of Real-Time Systems, 1*, 1989, 159-176.

[5] S. Yoo, I. Bacivarov, A. Bouchima, Y. Paviot and A. Jerraya: “Building fast and accurate SW simulation models based on hardware abstraction layer and simulation environment abstraction layer”, in *Proceedings of the Design, Automation and Test Conference*, IEEE, 2003, 550-555.

[6] H. Tomiyama, Y. Cao and K. Murakami: “Modeling fixed-priority preemptive multi-task systems in SpecC”, *Proceedings of the 10th Workshop on System And System Integration of Mixed Technologies (SASIMI’01)*, IEEE, 2001.

[7] A. Gerstlauer, H. Yu and D. Gajski: “RTOS modeling for system-level design”, in Embedded Software for SoC, A.A. Jerraya, S. Yoo, D. Verkest and N. When (Eds.), *Springer*, 2003.

[8] S. Yoo, G. Nicolescu, L. Gauthier and A. Jerraya: “Automatic generation of fast timed simulation models for operating systems in SoC design”, in *Proceedings of the Design, Automation and Test Conference*, IEEE, 2002, 620-625.

[9] Y. Yi, D. Kim and S. Ha: “Fast and time-accurate cosimulation with OS scheduler modeling”, *Design Automation of Embedded Systems, 8*, 2003, 211-228.

[10] H. Posadas, F. Herrera, P. Sánchez, E. Villar and F. Blasco: “System-level performance analysis in SystemC”, in *Proceedings of the Design, Automation and Test Conference*, IEEE, 2004, 378-383.

[11] S. Honda, T. Wakabayashi, H. Tomiyama and H. Takada: “RTOS-centric HW/SW cosimulator for embedded system design”, *Proceedings of CoDes-ISSS’04,* ACM, 2004.

[12] M.A. Hassan, K. Sakanushi, Y. Takeuchi and M. Imai: “RTK-Spec TRON: A simulation model of an ITRON based RTOS kernel in SystemC”, *Proceedings of the Design, Automation and Test Conference*, IEEE, 2005.

[13] L. Benini, D. Bertozzi, D. Bruni, N. Drago, F. Fummi and M. Ponzino: “SystemC cosimulation and emulation of multiprocessor SoC design”, *IEEE Computer,* April, 2003.

[14] W. Müller, W. Rosenstiel and J. Ruf: “SystemC: Methodologies and Applications”, *Springer*, 2003.

[15] F. Herrera, V. Fernández, P. Sánchez and E. Villar: “Embedded software generation from SystemC for platform-based design”, in SystemC: Methodologies and Applications, W. Müller, W. Rosenstiel and J. Ruf (Eds.), *Springer*, 2003.

[16] IEEE: “Information technology-Portable Operating System Interface”, IEEE Std 1003.1, 2004.

[17] G. Hawley: “Selecting a RTOS”, *Embedded Systems Programming Europe*, May, 1999.

[18] ENEA: “OSE Soft Kernel Environment”, available in http://www.ose.com/products.

[19] AXLOG, information available in http://www.axlog.fr.

[20] C. Liem, F. Naçabal, C. Valderrama, P. Paulin and A. Jerraya: “System-on-a-Chip cosimulation and compilation”, *IEEE Design & Test of Computers*, April-June 1997.

[21] EN 301.245, ETSI, December, 1997.

[22] M. Bolado, H. Posadas, J. Castillo, P. Huerta, P., Sánchez, C. Sánchez, H. Fouren. and F. Blasco: “Platform based on Open-Source Cores for industrial applications”, in *Proceedings of the Design, Automation and Test Conference*, IEEE, 2004, 1014-1019.