Abstract RTOS Modeling for Embedded Systems

Fabiano Hessel, Vitor M. da Rosa, Igor M. Reis, Ricardo Planner
Faculdade de Informática
Pontifícia Universidade Católica do RS
Av. Ipiranga 6681, Porto Alegre, BRAZIL
hessel@inf.pucrs.br

César A. M. Marcon, Altamiro A. Susin
Instituto de Informática
Universidade Federal do Rio Grande do Sul
Av. B. Gonçalves, 9500, Porto Alegre, BRAZIL
marcon@inf.ufrgs.br

Abstract

Raising the abstraction level is widely seen as a solution to increase productivity, in order to handle the growing complexity of real-time embedded applications and the time-to-market pressures. In this context, the use of a real-time operating system (RTOS) becomes extremely important to the development of applications with real-time systems requirements. However, the use of a detailed RTOS at early design phases is a contra sense, and the existing system level description languages (SLDL) lack support for RTOS modeling at higher abstraction levels. In this paper, we introduce an abstract RTOS model, and a set of refinement steps that allows refining the abstract model to an implementation model at lower abstraction levels. This abstract RTOS model provides the main features available in a typical RTOS, permitting the designer to model parallel and concurrent behavior of real-time embedded applications at higher abstraction levels. We use SystemC language with some extensions to build the abstract RTOS model, allowing a quick evaluation of different scheduling algorithms and synchronization mechanisms at the early stage of system design. An experimental result with a telecom system that consists of fifty tasks with four priority levels shows the usefulness of this model.

1 Introduction

In order to handle the fast-growing complexity of real-time embedded applications and time-to-market pressures, the design abstraction has been raising to the system level specification. Moreover, the software modules are taking increasingly important roles in the design of real-time embedded applications. Previsions are that embedded software represents 80% of the cost of an embedded system development [1]. Consequently, the use of a RTOS has become extremely important to manage the dynamic real-time behavior often found in embedded software. According to the SIA Roadmap, half a billion dollars in shipments of RTOSs was sold in 2002 [1]. However, the existing SLDL and methods lack support for modeling a RTOS at higher abstraction levels. The designers, consequently, must use a detailed RTOS implementation at higher abstraction levels. Nevertheless, at higher abstraction level, using a detailed RTOS is a contra sense negating the abstract system model principles.

The designer needs both new design methodologies and new design techniques to model an abstract RTOS behavior at early design phase. However, at higher abstraction levels, the system model does not have enough information available to model a specific RTOS. The capture of abstract RTOS behavior in system level models requires enhancements in current design practice.

Transaction level (TL) is an emergent description level for system level design. There are many authors [2][3] realizing TL in different ways. In fact, transaction expresses communication exchanges. In other words, transaction informs the relative order of each process communication. Transaction level is an abstraction level, which can lend some focus to the ordering of events. The TL modeling (TLM) is a sufficient model to represent the events ordering. The clock abstraction levels and the separation from computation and communication details make the model simpler and efficient for fast high-level evaluation.

This work addresses an abstract RTOS model for embedded systems at transaction level. Moreover, a set of successive refinement steps is proposed in order to synthesize the RTOS TLM into an implementation model at lower abstraction levels. The RTOS TLM is written on the top of SystemC language [4] with some extensions to model the dynamic real-time behavior. It provides the main features offered by typical RTOS in software development like real-time scheduling, interrupt handling, multitasking, task management, task synchronization and preemption [5].

Additionally, the designer can obtain power consumption estimate of the scheduling algorithms and its penalty in the overall system [6]. This estimation is a modeling feature that expresses the power consumption in the final implementation. The estimation starts from a rough

Proceedings of the 15th IEEE International Workshop on Rapid System Prototyping (RSP'04)
1074-6005/04 $20.00 © 2004 IEEE

IEEE COMPUTER SOCIETY

Authorized licensed use limited to: Pontificia Universidade Catolica do Rio Grande do Sul (PUC/RS). Downloaded on November 18,2022 at 14:22:00 UTC from IEEE Xplore. Restrictions apply.

evaluation and is improved by the back-annotation technique.

One important contribution of this work is to model an RTOS TL allowing fast evaluation of different scheduling algorithms and synchronization mechanisms at early design phases, since it enables system simulations an order of magnitude faster than lower abstraction levels like RTL. Another associated contribution is that our model can be integrated into existing system level design flows based on C/C++ languages (e.g. SystemC).

This paper is organized as follows: Section 2 presents the related work; the design flow and how the abstract model may be integrated are presented in Section 3; Section 4 presents the RTOS model; Experimental results with a telecom system that consists of fifty tasks with four priority levels are the subject of Section 5; and finally Section 6 presents conclusions and future work.

2 Related Work

Recently, several works have been focusing on automatic RTOS and code generation. Kohout [7] describes the Real-Time Task Manager (RTM) as a processor extension that minimizes the drawbacks associated with RTOSs by supporting, in hardware, a few of the common RTOS operations that are performance bottlenecks. Adomat [8] proposes an exclusive external hardware module designed to perform RTOS functions. This model improves performance, but it does not allow existing RTOSs to easily take advantage of its offerings.

Wang [9] proposes a high-level abstract model and synthesize operating system based on device drivers. O'Nil [10] uses a library for each supported OS and an automatic selection mechanism, they generate device drivers for a range of operating systems. Yi [11] proposes a virtual synchronization technique to the case where multiple software tasks are executed under the supervision of a real-time operating system in a single processor. It runs only application tasks on the ISS (Instruction Set Simulator) and models the RTOS in the cosimulation backplane to achieve faster cosimulation.

Cortadella [12] presents a way to combine static scheduling and dynamic scheduling in software synthesis. Gauthier [13] and Dziri [14] propose a methodology for automatic generation of application-specific operating systems and correspondent application software for a target processor. This methodology mainly focuses on software synthesis issues, the information regarding abstract model of the operating system integrated into whole system is not provided. Tomiyama [15] described a technique for modeling fixed-priority preemptive multi-tasking systems based on concurrency and exception-handling mechanisms provide by SpecC [16]. This model is limited in its support for different scheduling algorithms and inter-task communication.

More recently, researchers have realized the importance of dynamic behavior and propose to include it in system level design models. Such dynamic features are essentially services provided by an OS. Desmet [17] proposes a high level model of a system-on-chip operating system (SoCOS). It is used for modeling, simulation and analysis of the system, and implementation through gradual refinements. The emphasis is on the task concurrency issues. However, the SoCOS requires own proprietary simulation engine and a manual system model creation. Gonzales [18] proposes an abstract RTOS model using master-slave timed SystemC. The model has a global clock to keep track of time. Gerstlauer [19] describes an RTOS model, which is effectively a set of commonly used RTOS services, to extend the original SpecC language's ability to handle the interleaved execution behavior of dynamic schedulers. The adaptation of this model to another SLDL language like SystemC may be a hard and complex task, due to lack of support to model common services as preemption and true multitask execution.

Our abstract RTOS model is similar to Gonzales and Gerstlauer approaches. The main difference is that our RTOS model is written on top of SystemC language considering untimed system specification at higher abstraction levels [3]. By introducing some extensions in the SystemC scheduler execution model, we have a powerful and flexible RTOS model. Our model allows the preemption/resume task and the true multitask execution beyond make an estimated power consumption of the scheduling algorithms. It can be directly integrated into any SystemC-based system model and design flow, and is very easy to use.

3 Design Flow

This Section describes an embedded system design flow, starting from a TL specification, which is refined gradually to a hardware and software implementation model, as illustrated in Figure 1. The main issue is demonstrating the design flow for a specific application with automatic generation of an embedded RTOS.

The system design flow starts with TL specification written in SystemC/C++/C and IP modules, where the designer specifies the system behavior. The designer informs system requirements and architectural constraints, like power consumption limit, real-time constraints and number of processors of the target architecture. After this, two partitioning steps are accomplished. The first one determines IP components and hardware and software

IEEE COMPUTER SOCIETY

Authorized licensed use limited to: Pontificia Universidade Catolica do Rio Grande do Sul (PUC/RS). Downloaded on November 18,2022 at 14:22:00 UTC from IEEE Xplore. Restrictions apply.

processes. The IP and hardware synthesis are not the scope of this work. In the second partitioning step, the designer groups the software processes into multiple clusters. Each cluster will be mapped onto a processor in the final implementation. The result is a TLM where each cluster executes a specific behavior in parallel with other clusters. Abstract channels accomplish the communication between clusters.

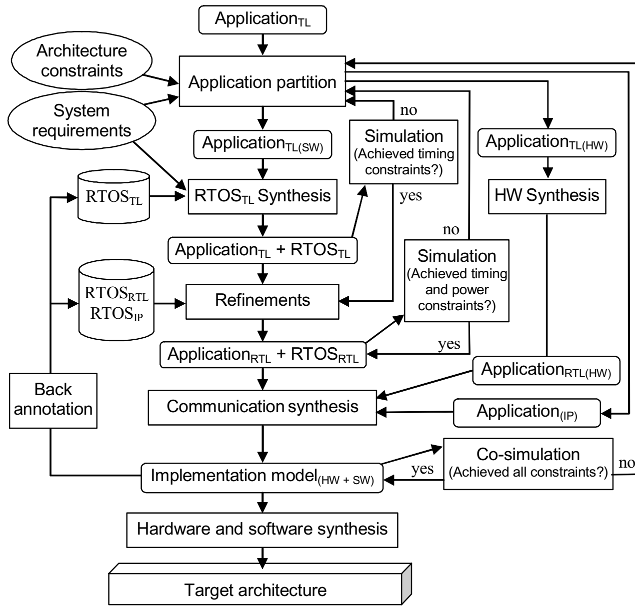

Figure 1: Design Flow

A RTOS TL library was designed to fulfill real-time constraints. It helps the designer to find the best RTOS scheduling policy at high abstraction levels considering performance evaluation and power consumption.

Many architectural aspects are omitted allowing faster design space exploration, mainly concerning to scheduling policy for multitasks and multiprocessor.

The RTOS synthesis step inserts the necessary RTOS primitives in all software processes, and the scheduling process. These primitives are operating system calls that allow memory management, interrupt request, inter-processes communication, synchronization mechanisms, and others OS features. At this point, the inter-processes communication primitives implement the abstract channels as a device driver.

We use profiles techniques to estimate the execution time of each process enabling the scheduling mechanisms to preempt software process according to the priority specified by the designer. In addition, a first power estimation of scheduling mechanisms can be done. This estimation is based on previous analysis of the scheduling algorithms. The estimation parameter is update by back-annotation techniques.

Transaction level abstracts some communication details, although it is possible to evaluate the events order and analyze if all time constraints are achieved with the chosen scheduling mechanism. It helps the designer to quickly search for the best scheduling mechanism of each processor and the inter-processor communication mechanism.

A first simulation step is applied to the system. The IP and hardware parts behavior are described as test-bench allowing software elements validation. Once the application achieves all requirements at transaction level, the designer can refine the application description and the selected RTOS for each processor.

The refinement of the application description from TLM to RTL is done manually generating a synthesizable description. The RTOS refinement is based on two available libraries: one that is the equivalent of RTOS TL at register transfer level and another that is composed by RTOS IP. The RTOS TL refinement to RTOS RTL is quite natural for our design flow, since both represent the same OS at different abstraction level. In this case, all TL primitives are changed to RTL primitives. On the other hand, the refinement to RTOS IP is harder due to different approaches adopted by IP providers, implying some extra manual steps. This level provides more precise timing and power consumption estimation.

The application and RTOS are validated by simulation and the systems requirements are evaluated. If the constraints are achieved, the flow goes to the next step, otherwise another scheduling policy or hardware/software partition can be evaluated.

For IP, hardware and software components communication interfaces are synthesized to hardware RTL according to the communication protocol and the target architecture. The design flow supports inter-process communication synthesis with shared memory, rendezvous, FIFO and buses. The communication synthesis problem is not addressed in this work [14].

A hardware/software cosimulation is the last validation step. It considers one simulator for each processor and one simulator for hardware components. This step usually expends much time, mainly due to the input simulator vector. Nevertheless, the cosimulation step is associated with an accurate power model, which allows to feedback the achieved power value to the RTOS libraries, improving possible next evaluations.

As the last step of the design flow, hardware and software are synthesized to the target architecture. Our first approach considers a single FPGA as target architecture. In this context, hardware components are synthesized using specific FPGA commercial tools. For each software cluster, all RTOS primitives are mapped to the correspondent RTOS API, enabling to compile the code into processor

Authorized licensed use limited to: Pontificia Universidade Catolica do Rio Grande do Sul (PUC/RS). Downloaded on November 18,2022 at 14:22:00 UTC from IEEE Xplore. Restrictions apply.

instruction set. Each compiled code produces the executable code for each processor.

4 The RTOS Model

As mentioned previously, our RTOS model is implemented on the top of the SystemC language. However, the SystemC lacks support to model the dynamic real-time behavior commonly found in embedded software. Typically, SystemC does not provide a mechanism to preempt and resume a thread during execution time. In order to allow the aforementioned problem, we make some languages extensions. The SystemC Open Initiative still works in a new release (SystemC 3.0) to provide mechanisms to solve it.

The RTOS model is incorporated into the RTOS TL library and can be parametrizable in terms of task parameters. The library provides RTOS models with different scheduling algorithms. Our RTOS model supports both periodic and non-periodic real-time tasks.

The RTOS model provides two major categories of services: OS management and Task management.

OS management services are responsible to the initialization of the RTOS. The *sc_rtos_init* initializes the relevant RTOS data structures and starts the multitasking scheduling. In addition, the *sc_rtos_reset* reinitializes the RTOS, it is very useful for validation purposes. In order to allow the preemption and resume tasks during execution time, we introduced two primitives: *sc_rtos_task_suspend* that preempt a task and *sc_rtos_task_resume* that resume a task. These primitives receive the task identification as parameter.

Task management services are responsible to make the interface between the kernel and the system application. One of the objectives is to provide to the user an easy way to describe an application as a set of tasks. In the following sections we will discuss the task model, scheduler model, and synchronization model.

4.1 Task Model

We model the task such that it holds all necessary information to execution. Each task is implemented as a PosixThread in order to allow preemption and resume by the scheduler. The *sc_task_create* primitive is used to characterize the execution of the task. It defines the task parameters such as: identification, name, priority, period, deadline, worst-case execution time (WCET), and best-case execution time (BCET). In addition, this primitive assigns the task to the scheduler that attributes *idle* as the initial task state.

Several others standard RTOS primitives are included in the model like as task notify (*sc_task_notify*), task termination (*sc_task_end*), and task suspension (*sc_task_wait*). To model periodic tasks, we introduced the *sc_task_end_cycle* primitive. This primitive notifies the scheduler that a task finished its computation in the current cycle. Figure 2 presents a partial source code example of task modeling. The system *sys_ex* is initialized (line 2) and executed by 100,000 ns (line 4). We have two tasks: t1 and t2 (lines 8 and 10). The task t1 is created with the following parameters: identification = id1, name = t1, priority = 1, period = 80, WCET = 14, BCET = 8, and deadline = 30. The task t1 is assigned to the scheduler in line 9. The RTOS scheduler is initialized with time slice (line 13). Tasks are derived of PosixThread class (line 16).

```
1.   int sc_main(int argc, char *argv[]) {
2.     system sys_ex("System example");
3.     ...
4.     sc_start(100000, SC_NS);
5.   }

6.   class system : public sc_rtos {
7.     system(sc_module_name name) : sc_rtos(name)  {
8.       task t1 = new task(id1, "t1", 1, 80, 14, 8, 30);
9.       sc_task_create(t1);
10.      task t2 = new task(id2, "t2", 3, 60, 12, 11, 25);
11.      sc_task_create(t2);
12.      ...
13.      sc_rtos_init(1);
14.    }
15.  };

16.  class task: public PosixThread, public sc_module {
17.    task(id, "task_name", priority, period, wcet, bcet, deadline) :
18.      sc_module("task_name") {
19.      ...
20.    }
21.    run() {
22.      while(true) {
23.      // task behavior pointed by id
24.      sc_rtos_end_cycle();
25.    }
26.  };
```

Figure 2: Task modeling

4.2 Scheduler Model

At the system level we are not interested in the exact task functionality, but rather how long it takes to compute and the tasks interactions. From this point of view, the first task of the RTOS is to determine which process runs next, e.g. to decide the tasks execution order. Task management, performed by scheduler, is the most important function in the RTOS model. Our scheduler model considers that all tasks are independent threads. Hierarchical tasks need to be flatted. Each task is characterized by deadline, period, priority, WCET, and BCET. Moreover, a task may be preempted by a higher priority task.

Authorized licensed use limited to: Pontificia Universidade Catolica do Rio Grande do Sul (PUC/RS). Downloaded on November 18,2022 at 14:22:00 UTC from IEEE Xplore. Restrictions apply.

The scheduler model considers that a task can be in one of three basic scheduling states: *ready, execute* or *idle* [6], as is depicted in Figure 3. There is at most one task executing at any time. If there is no useful work to be done, just the schedule task works. Our model considers that all tasks are in the *idle* state at the beginning (*sc_task_create*). The task stays in *idle* state while it does not enter in a new period or while it needs data that is not yet received. A task goes into the *ready* state when it receives the required data, when it enters in a new period or when it is preempted by a higher priority task. When a task is preempted (*sc_rtos_task_suspend*) it waits for a *resume* command from the scheduler (*sc_task_resume*). A task can go into the *execute* state when it receives a *run* command from the scheduler. The task will receive this command only when it has all data required, is ready to run, and the scheduler selects the task as the next task to run. Once the task has finished its computation in the current cycle, it sends a message to the scheduler (*sc_task_end_cycle*) and goes to the *idle* state. The task also goes to the idle state when it requests a data that is not available. Otherwise, when non-periodic task finished its execution, it sends a terminate message to the scheduler (*sc_task_end*). In this case, the scheduler kills the task (*sc_rtos_task_kill*).

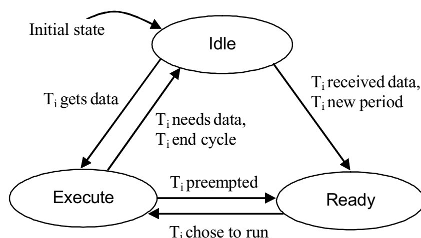

Figure 3: Scheduling state of tasks

The scheduler is modeled as a SystemC thread process (*sc_thread*) that runs continually. In case of a task goes to the *idle* or *ready* states, the scheduler selects among the ready tasks, a candidate task to run, according to scheduler algorithm. However, if there is not a candidate task (ready list is empty), the scheduler just waits until a ready task is available. For instance, our scheduler implements FCFS, Round Robin, Rate-Monotonic (RM) and Earliest Deadline First (EDF) scheduling algorithms [20].

4.3 Synchronization Model

RTOS synchronization model provides services to synchronize concurrent and cooperative tasks, supporting mechanisms that handle inter-processor and intra-processor synchronization problems. Our model offers two primitives: *sc_task_wait* and *sc_task_notify*.

The *sc_task_wait* calls causes current task to wait until another task invokes the *sc_task_notify* primitive or a specified amount of time has elapsed. When it happens the task goes to the *idle* state being inserted into a wait task list and becomes disabled for scheduling purposes. The *sc_task_notify* calls wakes up a single task that is waiting for data synchronization.

When tasks execute input/output operations, like send/receive the tasks need to notify the RTOS scheduler. We implemented this notification by the use of these two primitives. An abstract receive operation is implemented on lower levels as a receive function aggregated to *sc_task_wait* call, meaning that the task is waiting for input data. An abstract send operation is implemented on lower abstraction levels as a send function aggregated to *sc_task_notify* call. The *sc_task_notify* allows to scheduler wake up the tasks that are waiting for the sent data.

5 Case Study

We use our design flow to redesign a telecom system in context of industry/academy cooperation. The system is a digital private branch exchange (PBX) whose commercial name is XT-130. The PBX is a soft real-time system [6]. The industry goal is to aggregate new important feature, as voice over IP (VOIP), without losing time redesign all telecom product line. The main trouble is that all telecom system was ad hoc designed to support real-time requirements; generating a monolithic system, where application and operating system are strongly coupled. To aggregate new feature without much effort it is necessary to use an OS that supports the actual and new features, which is generally found in modular designs. However, monolithic to modular design swap can imply functionality reduction, mainly to real-time functions, implying many design evaluating time, reducing the industry profits and many times resulting market losses. As a solution, we proposed our approach that enables fast evaluations at earlier design stages.

The PBX is a complex system composed by more than fifty processes, with four priority levels. Twenty percent of these processes have real-time requirements. Since much code is developed in C/C++ and assembly, we proposed a partitioning where system processes are divided as follows: 92% software elements; 6% assembly routines (treated in our design flow as IP components); and 2% hardware elements. The hardware parts are mapped into Altera FLEX-10KE FPGA. The software elements are grouped in clusters (Section 3). IP modules and software clusters are mapped into AM186ES (AMD 80186) microprocessor and ADSP2185M (Analog Devices) DSP. For each processor a small custom RTOS kernel was generated and the system description was refined and targeted to the architecture using the design flow described in Figure 1. In

IEEE COMPUTER SOCIETY

Authorized licensed use limited to: Pontificia Universidade Catolica do Rio Grande do Sul (PUC/RS). Downloaded on November 18,2022 at 14:22:00 UTC from IEEE Xplore. Restrictions apply.

communication synthesis step we chose shared memory for processors communication, and rendezvous protocol for FPGA and microprocessor communication. There is no communication between FPGA and DSP processes.

There were some doubts to be solved, firstly if the new RTOS approach fulfilled all real-time constraints, secondly if the RTOS code size was acceptable, since memory was a strong design constraint. These doubts were answered by the RTOS code compilation and by the TL/RTL simulation. Table 1 depicts the code size achieved for RTOS and the rest of application for both processors.

Table 1: Code size comparison (in bytes)

	AM186ES	DSP
C/C++	457,976	16,356
Assembly	22,233	27,453
Scheduling algorithm	7,456	2,489

Test-bench vectors, extracted from real PBX operation during high activity (ten minutes of operation time), excited the PBX description during RTL simulation and cosimulation phases. The three AM186ES simulations, illustrated in Table 2, show the advantages achieved by high description levels.

We use profile techniques, with the test-bench vectors, to estimate the WCET and the BCET of each process, these times are entry of each process in TL simulation. Therefore, WCET, BCET and the execution period replaces the process behavior, allowing faster simulation with reasonable accuracy, as RTL simulation confirms. For TL and RTL simulation, the rest of the systems is considered as test-bench, on the other hand, the cosimulation considers the joint operation of three simulators (two C/C++ simulators and one VHDL simulator).

Table 2: AM186ES simulation analysis

Simulation	time
TL-simulation	16min
RTL-simulation	6h 15min
Cosimulation	98h 43min

At TL it was possible to observe that RM scheduling achieved the smallest number of context switches, as it is depicted in Table 3. Table 3 also shows the number of times that the real-time processes that did not achieve theirs deadline in TL simulation, needing to be delayed. EDF scheduling acquired the best result. Considering context switching, real-time deadline and the low algorithm complexity, we chose RM as the scheduler policy for AM186ES operation. The majority of DSP tasks are time slices scheduled by a timer interrupt. Nevertheless, some tasks with less priority are Round-robin scheduled by a small custom RTOS kernel, with less than 3 Kbytes (Table 1). The total size of AM186ES RTOS is three times bigger than the ADSP2185M, due to other included features, like memory management.

Table 3: AM186ES scheduling analysis

Scheduling	Context switches	RT constraints fail
Round-robin	465,577	97
EDF	490,254	3
RM	402,239	5

Actually, we are studying the possibility of replacing our custom RTOS by RtLinux open source. This decision is motivated to allow the evaluation of the RTOS IP branch in our design flow. The great challenge is to maintain AM186ES processor, which addresses only 1 Mbyte of memory.

6 Conclusions and Future Work

This paper addresses the issue of modeling abstract RTOS at higher abstraction levels. We presented an abstract RTOS model and a set of refinements steps that allows refining the abstract model to an implementation model at lower abstraction levels. The model allows the designer to evaluate quickly different scheduling algorithms and synchronization mechanisms at early design phases in order to validate the dynamic real-time behavior of the system. Additionally, the designer can have an estimated power consumption of the scheduling algorithms and its penalty in the overall system. The proposed abstract model provides all main features found in any modern RTOS but not available in current SystemC language through a reduce set of system calls. We apply this model in the development of a PBX system composed of fifty tasks with four priority levels and real-time requirements.

Future work includes implementing the RTOS interfaces for commercial real-time operational systems and new techniques to power estimation at higher abstraction levels.

7 Acknowledgement

This work is sponsored by CNPq/Brazil contract # 300291/2003-5.

References

[1] International Technology Working Group: *International Technology Roadmap for Semiconductors 2001 Edition: Executive Summary.* Semiconductor Industry Association, 2001.

[2] Grotker, T.; Liao, S.; Martin, G. and Swan, S.: *System Design with SystemC*. Kluwer Academic Publishers, 2002.

Authorized licensed use limited to: Pontificia Universidade Catolica do Rio Grande do Sul (PUC/RS). Downloaded on November 18,2022 at 14:22:00 UTC from IEEE Xplore. Restrictions apply.

[3] Cai, L.; Gajski, D. Transaction Level Modeling: An overview. In *Proceedings of CODES+ISSS*, pp. 19-24, Newport Beach, California, USA, October 2003

[4] Synopsys Inc.: *SystemC Version 2.0 Users Guide.* 2003. Available at: www.systemc.org.

[5] Butazzo, G.: *Hard Real-Time Computing Systems.* Kluwer Academic Publishers, 1999.

[6] Wolf, W.: *Computer as Components: Principles of Embedded Computing System Design.* Morgan Kaufmann, 2000.

[7] Kohout, P.; Ganesh, B.; Jacob, B.: *Hardware Support for Real-time Operating Systems*. In *Proceedings of CODES+ISSS*, pp. 45-51, Newport Beach, California, USA, October 2003.

[8] Adomat, J.; Furunäs, J.; Lindh, L.; Stärner, J. Real-Time Kernel in Hardware RTU: *A step towards deterministic and high performance real-time systems*. In *Proceedings of 8th Euromicro Workshop on Real-Time Systems*, L'Aquila, Italy, pp. 164-168, June 1996.

[9] Wang, S.; Malik, S.: *Synthesizing Operating System Based Device Drivers in Embedded Systems.* In *Proceedings of CODES+ISSS*, pp. 37-44, Newport Beach, California, USA, October 2003.

[10] O'Nil, M.; Jantash, A.: *Device Driver and DMA Controller Synthesis from HW/SW Communication protocol specifications*. *Design Automation for Embedded Systems*, vol. 6, pp.177-205, 2001.

[11] Yi, Y.; Kim, D.; Ha, S.: *Virtual Synchronization Technique with OS Modeling for Fast and Time-accurate Cosimulation.* In *Proceedings of CODES+ISSS*, pp. 1-6, Newport Beach, California, USA, October 2003.

[12] Cortadella, J.: *Task generation and compile time scheduling for mixed data-control embedded software*. In *Proceedings of Design Automation Conference*, June 2000.

[13] Gauthier, L.; Yoo, S.; Jerraya, A.: *Automatic generation and targeting of application-specific operating systems and embedded system software*. *IEEE Transaction on CAD*, November 2001.

[14] Dziri, M.; Samet, F.; Wagner, F.; Cesario, W.; Jerraya, A.: *Combining Architecture Exploration and a Path to Implementation to Build a Complete SoC Design Flow from System Specification to RTL.* In *Proceedings of ASP-DAC*, pp. 219-224, Kitakyushu, Japan, January 2003.

[15] Tomiyama, H.; Cao, Y.; Murakami, K.: *Modeling fixed-priority preemptive multi-task systems in SpecC*. In *Proceedings of SASIMI*, October 2001.

[16] SpecC. Available at: www.specc.org.

[17] Desmet, D.; Verkest, D.; DeMan, H.: *Operating System based Software Generation for System-on-Chip*. In *Proceedings of Design Automation Conference*, June 2000.

[18] Gonzales, M.; Madsen, J.: *Abstract RTOS Modeling for Multiprocessor System-on-Chip*. In *Proceedings of International Symposium on SoC*, 2003.

[19] Gerstlauer, A.; Yu, H.; Gajski, D.: *RTOS Modeling for System Level Design*. In *Proceedings of DATE*, March 2003.

[20] Silberschatz, A.; Galvin,P.: *Operating System Concepts.* John Wiley & Sons Inc., 2000.

1074-6005/04 $20.00 © 2004 **IEEE**

IEEE COMPUTER SOCIETY

Authorized licensed use limited to: Pontificia Universidade Catolica do Rio Grande do Sul (PUC/RS). Downloaded on November 18,2022 at 14:22:00 UTC from IEEE Xplore. Restrictions apply.