RTOS-Centric Hardware/Software Cosimulator for Embedded System Design

Shinya Honda Takayuki Wakabayashi
Department of Information and Computer Science
Toyohashi University of Technology
1-1 Hibarigaoka, Tenpaku-cho, Toyohashi 441-8580, Japan

Hiroyuki Tomiyama Hiroaki Takada
Graduate School of Information Science
Nagoya University
Furo-cho, Chikusa-ku, Nagoya 464-8603, Japan

{honda, takayuki, tomiyama, hiro}@ertl.jp

ABSTRACT

This paper presents an RTOS-centric hardware/software cosimulator which we have developed for embedded system design. One of the most remarkable features in our cosimulator is that it has a complete simulation model of an RTOS which is widely used in industry, so that application tasks including RTOS service calls are natively executed on a host computer. Our cosimulator also features cosimulation with functional simulation models of hardware written in C/C++ and cosimulation with HDL simulators. A case study with a JPEG decoder application demonstrates the effectiveness of our cosimulator.

Categories and Subject Descriptors

C.3 [**Special-Purpose and Application-Based Systems**]: Real-Time and Embedded Systems; D.4.8 [**Performance**]: Simulation; B.7.2 [**Design Aids**]: Simulation and Verification.

General Terms

Design, Performance, Verification

Keywords

RTOS, Cosimulation, Embedded Systems

1. INTRODUCTION

Nowadays, real-time operating systems (RTOSs) have become one of the most important components of embedded systems due to the growing complexity of the system functionalities as well as the increasing time-to-market pressures. With this trend, a demand for fast cosimulation of hardware and embedded software including an RTOS is also becoming stronger in order to validate the functionality of the overall systems.

For this purpose, we have developed a fast cosimulator which fully supports services of an RTOS. Since the RTOS model plays a central role in our cosimulator, we call our cosimulator an *RTOS-centric cosimulator*.

The RTOS model in our cosimulator is provided to a system designer in the form of C/C++ source code. Application software tasks written by the designer are compiled and linked with the RTOS model to generate an object code which is directly executable on the host computer. Due to the native execution, the speed of our cosimulator is much higher than that of traditional cosimulation which uses an instruction-set simulator (ISS) of the target processor for software execution. It should be noted that simulation using our cosimulator is basically untimed, but timed simulation is also possible by inserting a delay function into the software code.

Our cosimulator supports two types of hardware simulation. One is cosimulation with functional hardware models written in C or C++. This enables fast functional validation of the overall systems. The other is cosimulation with HDL simulators, e.g., *ModelSim* [1]. Using this ability, register-transfer level (RTL) design implementation of the hardware can be simulated together with embedded software (i.e., RTOS and application tasks). In this case, our cosimulator works as a testbench which generates input data to and receives output data from the hardware in an interactive manner. Thus, the hardware design can be validated efficiently.

Our cosimulator supports all the services of the *μITRON 4.0* standard [2][3]. μITRON is one of the most popular RTOSs in Japan for small- to middle-scale embedded systems. μITRON is not a specific RTOS product, but is an application programming interface (API) standard. It only defines a set of API functions, and implementation of the function bodies may differ among μITRON-based RTOSs. It is reported in [3] that over 40% of RTOSs used in Japan are based on the μITRON standard. Our simulator implements all of 81 API functions which are defined by *μITRON 4.0 Standard Profile*. Therefore, one can use the same application code for both simulation and final implementation if a μITRON-based RTOS used in the implementation. Due to this, the design productivity can be significantly improved.

In summary, our RTOS-centric cosimulator features

- native (hence, fast) software execution on a host computer,
- cosimulation with functional hardware models in C/C++,
- cosimulation with HDL simulators, and
- complete support of RTOS services.

To our knowledge, no other cosimulator satisfies all of the above features.

This paper is organized as follows. Section 2 surveys related work on hardware/software cosimulation with RTOS supports. Section 3 describes principles and implementation of the RTOS-centric cosimulator which we have developed. A case study using a JPEG decoder application is presented in Section 4. Section 5 concludes this paper with a summary.

2. RELATED WORK

Hardware/software cosimulation has been studied around the world for more than a decade, and a number of commercial cosimulators have come onto the market. The cosimulators are currently one of indispensable CAD tools in the design of embedded systems and systems-on-chip. However, since the traditional cosimulators do not feature explicit supports for RTOSs, a designer needs to use an ISS in many cases to run embedded software including an RTOS. Due to the slow execution speed, extensive simulation of large software is impossible.

In the recent years, several research efforts have been made to model RTOSs for system-level design and cosimulation. *SoCOS* presented in [4] is a system-level design environment where the *OSAPI* library provides generic RTOS system calls to application software. OSAPI is a virtual RTOS to enable native execution of embedded software. After simulation-based validation, the OSAPI calls are replaced with the system calls of the actual RTOS used in the final implementation to obtain the final software code. In [5], a similar approach is presented. A main difference is that the RTOS model in [5] is build upon an existing system-level design language, i.e., SpecC [6], so that existing CAD tools such as simulators can be used. Techniques presented in [7] also use the SpecC language to model the preemptive behavior. In [8], an OS model is proposed for fast and time-accurate cosimulation. The model focuses on accurately modeling the RTOS overhead during task execution as well as the preemptive behavior. In [9], a method which automatically generates RTOS-dependent software from SystemC description is presented. The method replaces SystemC's constructs for concurrency and communication with corresponding RTOS service calls. In this sense, it can be considered that SystemC involves a simple RTOS model in itself.

A common weakness of these OS models is that they support only a limited set of RTOS services in order to make the models generic and independent of specific RTOSs. μITRON-based RTOSs, for example, have more than 80 service calls. These services may need to be fully utilized in order to write high-quality software. However, the RTOS model in [5], for example, supports only 16 service calls. Therefore, it is easily imagined that the quality of software automatically generated by these previous methods is lower than that of hand-crafted RTOS-dependent software. Since the RTOS-centric cosimulator which we have developed fully supports RTOS services, such high-quality software can be designed and simulated. Instead, our simulator has a different weakness that it can be used only for μITRON-based platforms.

In fact, our RTOS-centric cosimulator is not a replacement of the cosimulators with generic RTOS modeling, but is complementary to them. The cosimulators with generic RTOS modeling can be used in the high-level design phases where the RTOS to be used in the implementation is not determined yet. After RTOS selection, the software is refined into RTOS-dependent code. Of course, the RTOS-dependent software needs to be cosimulated to verify the correctness of its functionality. So far, the RTOS-dependent software has been executed using an ISS, so the cosimulation speed is slow. With our RTOS-centric cosimulator, however, the RTOS-dependent software can be cosimulated at a very high speed. After the cosimulation-based functional validation, cycle-accurate cosimulation will be done using an ISS. Thus, our cosimulator fills the gap of abstraction between system-level RTOS-independent cosimulation and cycle-accurate cosimulation.

Some RTOSs have their simulation models for native execution on a host computer. For example, *VxWorks* from WindRiver Systems [10] has its simulation model named *VxSim*. Our cosimulator is similar to the RTOS simulation models. To our knowledge, however, the RTOS simulation models do not explicitly support cosimulation with hardware. On the other hand, our cosimulator supports cosimulation with standard HDL simulators as well as untimed hardware models in C or C++.

A different approach to embedded software design is presented in [11], where an application-specific RTOS and its simulation model is automatically generated. In their approach, application software is analyzed first, and then only the RTOS services used in the application are included in the final RTOS. The work is similar to ours in the sense that a set of services which can be used in application software is pre-defined. The major difference is that the work in [11] puts a special focus on customizing an RTOS while our cosimulator is based on a standard RTOS.

In [12], the authors proposed a timed HW/SW cosimulation method where an RTOS and application tasks are executed natively on a host computer. The work is similar to ours in terms of native execution of the RTOS, but the goals are different. Their goal is to accurately model timing of native software execution. The RTOS simulation model is based on the work in [11]. It is assumed that hardware modules are modeled in SystemC. On the other hand, our primary goals are to develop a complete simulation model of a standard RTOS and to provide a flexible cosimulation framework which allows easy plug-and-play of HDL simulators and functional hardware models. Thus, the work in [12] and our work are complementary to each other.

3. RTOS-CENTRIC COSIMULATOR

The RTOS-centric cosimulator which we have developed implements all the services defined in *μITRON 4.0 Standard Profile*. In this section, an overview of μITRON is presented, and then our cosimulator is described in detail.

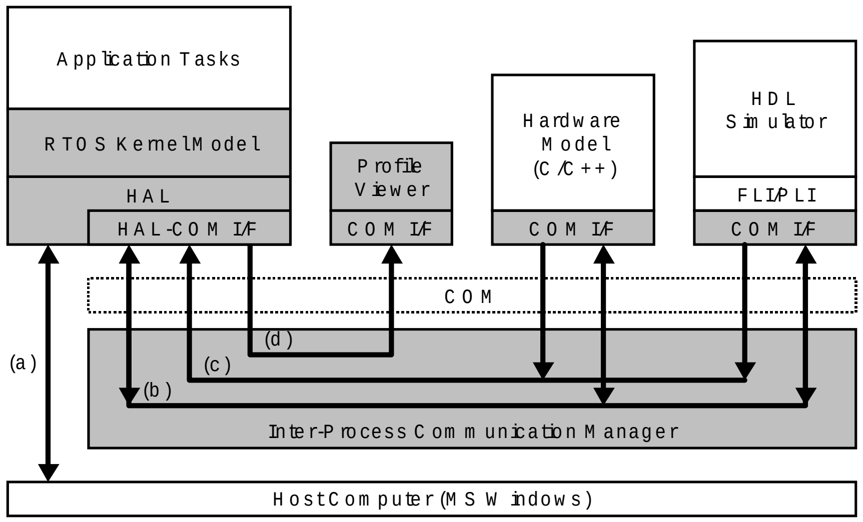


Figure 1. Overall organization of the RTOS-centric cosimulator

3.1 µITRON

µITRON is a set of APIs which has been standardized mainly in Japan. µITRON is designed for small- to middle-scale embedded systems such as consumer electronic products, mobile devices, automotive electronic control systems, and so on. µITRON-based RTOSs have been widely used in industry, especially in Asian countries. It is reported in [3] that over 40% of RTOSs used in Japan are based on the µITRON standard. The latest formal release at present is µITRON 4.0. The minimal set of µITRON APIs is called *Standard Profile*, and µITRON 4.0 Standard Profile includes 81 API functions in total [1].

µITRON employs a priority-based preemptive scheduling policy, and the priority of tasks can be changed at runtime. µITRON 4.0 Standard Profile states that the number of priority levels must be at least 16. µITRON supports several basic synchronization and communication mechanisms, such as semaphores, events, data queues, and mailboxes. Dynamic memory allocation using so-called memory pools is supported in µITRON. Since µITRON is designed for small embedded systems, virtual memory or dynamic module loading is not supported. Interrupt handling is one of important mechanisms in RTOSs. µITRON provides a number of API functions for defining interrupt handlers, allowing and prohibiting interrupts, changing interrupt masks, and so on. The number of interrupt levels is not determined by µITRON but implementation-dependent. µITRON has three types of time event handlers, i.e., cyclic handlers, alarm handlers, and overrun handlers. Cyclic handlers are invoked periodically, alarm handlers are invoked at a specified time, and overrun handlers are invoked when the execution time of a task exceeds a specified time.

In addition, µITRON provides a number of services which are necessary for industrial use. More detailed documents on µITRON are found in [3].

3.2 Cosimulator Overview

In the rest of this section, we describe our RTOS-centric cosimulator which is based on the µITRON standard. The cosimulator implements all the services defined in µITRON 4.0 Standard Profile. Figure 1 shows the overall organization of our RTOS-centric cosimulator. The shaded parts in the figure are included in the cosimulator.

The cosimulator is developed in the C and C++ languages and runs on *Microsoft Windows*-based computers. The cosimulator has been released as a part of the *TOPPERS/JSP Kernel* package [16][2]. The TOPPERS/JSP Kernel is a µITRON-based RTOS kernel which we have developed earlier, and it is freely available for both research and commercial purposes.

The RTOS model, consisting of the RTOS kernel and HAL (hardware abstraction layer) in Figure 1, is provided to a system designer in the form of C/C++ source code. Application tasks which are written in the C language by the designer are compiled and linked with the RTOS model, in order to generate an object code which is directly executable on the *Windows*-based host computer.

Our RTOS-centric cosimulator supports cosimulation with functional hardware models written in C or C++, which enables fast validation of the overall system functionality. Cosimulation with standard HDL simulators is also supported so that RTL hardware designs can be validated efficiently.

3.3 Task Management

Application tasks and the RTOS model are compiled together using a native C/C++ compiler on a host computer, and an object

[1] In this paper, the term µITRON often refers to µITRON 4.0 Standard Profile depending on the context.

[2] The ability to communicate with HDL simulators is not included in the current release.

file is generated. The object file is executable directly on the host computer. Thus, from a viewpoint of the host computer, the object code (i.e., application tasks and the cosimulator) is no more than an application process, and is scheduled in a time-sharing manner with the other Windows applications.

Within the cosimulator, each of the application tasks is implemented as a thread, and the RTOS kernel schedules the threads in a priority-based preemptive manner. This thread-based implementation facilitates debugging of the application tasks since the context of each task, such as mode (running, ready, waiting, suspended, etc.), program counter, contents of stack memory, and so on, can be easily monitored with normal C/C++ debuggers. If all the tasks are linked into a thread, it is not easy to retrieve and monitor the contexts of individual tasks without RTOS-specific support tools.

3.4 Timer

Unlike the reference simulator of SystemC [13] or SpecC [6], our RTOS-centric cosimulator does not have a global timer for managing simulation cycles. Therefore, our cosimulator does not support cycle-accurate execution of software.

Instead of the global clock for simulation, our cosimulator uses the timer of the host computer running on MS-Windows. In Figure 1, the arrow labeled (a) denotes the system call to the host computer for getting the time.

In our cosimulator, the period of timer ticks is customizable, but it must be equal to or longer than 1 millisecond. If an application task executes a system call, *dly_tsk(100)*, the RTOS kernel suspends the execution of the task for 100 milliseconds in the real time on the host computer. The system call does not mean that the task waits for 100 simulation cycles.

3.5 Cosimulation Mechanism

As mentioned earlier, our cosimulator supports two types of hardware models, i.e., functional simulation models in C/C++ and hardware designs in HDL. The functional simulation models are compiled with a native C/C++ compiler to generate object code which is executable on the host computer. The HDL designs are simulated on HDL simulators which support PLI (Programming Language Interface) or FLI (Foreign Language Interface) [3]. It should be noted that both functional simulation models and HDL simulators can be executed simultaneously, and the number of the functional models and that of HDL simulators can be more than one. For example, it is possible to cosimulate a software model (consisting of the RTOS model and application tasks), two functional hardware models, and three HDL simulators simultaneously.

In our cosimulator, as shown in Figure 1, *Inter-Process Communication Manager (IPCM)* manages communication between the software model and the hardware models. IPCM is based on the *COM (Component Object Model)* technology developed by Microsoft Corporation [14]. COM is an object-oriented interface technology which enables binary software components to interact with other software components in order to realize higher-level services. In our cosimulator, the software model, functional hardware models, and HDL simulators run as different processes on the host computer. Each of the software model, the functional hardware models and the HDL simulators interacts with IPCM through COM. Then, communication between the software and the hardware is performed by way of IPCM.

μITRON defines a set of guidelines for designing hardware devices and their drivers. In the guidelines, it is recommended that all devices (except a timer and a primary serial I/O port) be accessed through memory-mapped I/O. The guidelines also define a hardware abstraction layer (HAL) which includes a set of functions for memory-mapped I/O accesses. For example, a function, *sil_reb_mem(mem)*, reads 1-byte data from the device which is mapped to the address pointed by *mem*. Our RTOS-centric cosimulator assumes that hardware devices and the drivers are designed based on the guidelines.

Let us assume that *sil_reb_mem(mem)* is executed in the software. First, the RTOS model sends a request message to IPCM through COM. The message includes the access type (read or write), the size of required data, and the address. IPCM has a memory-map table, and based on the table and the received address, IPCM sends the request message to the corresponding functional hardware model or HDL simulator. The requested data is then sent from the hardware to the software by way of IPCM. Thus, IPCM serves as a router of the messages and data. In Figure 1, communication based on memory-mapped I/O is depicted by arrow (b). Interrupts from the hardware to the software are also handled by IPCM, and are depicted by arrow (c) in Figure 1.

The cosimulation mechanism with IPCM is flexible in several points. First, our cosimulator can simulate any number of hardware models simultaneously as long as there is no conflict in the address map. Also, it is possible to simulate both C/C++ models and HDL models simultaneously. Second, a C/C++ model can be easily replaced with an HDL model or vice versa without modifying application tasks, provided the address map of the hardware remains unchanged. This ability facilitates validation of hardware designs. Finally, the RTOS model can be easily replaced with an ISS running on a μITRON-based RTOS by adding the HAL-COM interface to the ISS, without modifying application tasks or hardware models. Thus, our cosimulator enables plug-and-play of software models as well as hardware models. This is due to the flexible communication mechanism of IPCM as well as the complete supports of the μITRON standard.

3.6 Profiler

Another remarkable feature in our cosimulator is the profiler to collect various data during execution. The data includes start/finish times and return code (if any) of application tasks, interrupt handlers, time event handlers, service calls, and so on. Such data is useful to debug or optimize the designs.

The obtained data can be displayed at runtime on a window (i.e., named *Profile Viewer* in Figure 1). The profiler is implemented within HAL, and the profiled data is sent to the viewer through IPCM. Of course, profiling requires some amount of overhead on simulation speed and memory requirement. In order to reduce the overhead, the profiler can be enabled and disabled during simulation. Therefore, only the necessary data can be obtained with the minimal overhead.

[3] At present, only the *ModelSim* HDL simulator [1] is supported in our cosimulator.

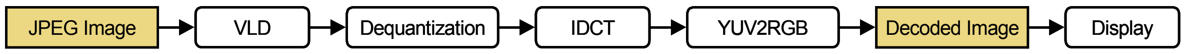


Figure 2. A JPEG decoder example.

It should be noted that much more information can be obtained by using normal C/C++ debuggers if the cosimulator is compiled with the debugging option enabled, i.e., the "-g" option for many C compilers. It is also possible to embed any C code for profiling into the cosimulator, since the source code of the cosimulator is available.

4. DESIGN EXAMPLE

In order to evaluate the effectiveness of our RTOS-centric cosimulator, we designed a JPEG decoder system and simulated it with our cosimulator. The simulation was done on dual Xeon 2.4 GHz processors with hyper-threading technology, running on MS-Windows XP. Therefore, up to four threads can be run in parallel on the host computer. Figure 2 shows the flow of the JPEG decoder where there are mainly four functions: *VLD, Dequantization, IDCT,* and *YUV2RGB*. In addition, there is a function, *Display*, to display the decoded image on a window of the host computer.

First, we implemented all of the functions in software except the display function. The display function was designed only for the simulation purpose, so it was written as a functional simulation model in C/C++. A JPEG image is decoded by software, and then the decoded image is sent from the software to the display window through IPCM. The overall JPEG decoder was implemented as an application task. We compared cosimulation speed of our cosimulator with that of an instruction-set simulator, *ARMulator* [15]. On ARMulator, the JPEG task was executed with the TOPPERS/JSP Kernel [16]. Note that we used the same JPEG code for both our cosimulator and ARMulator without modification. Table 1 summarizes the results on the simulation time.

Table 1. Cosimulation time of the JPEG decoder. All of the functions are implemented in software.

CPU Utilization for the JPEG task	100%	50%	25%
Our Cosimulator	0.009 sec	0.008 sec	0.008 sec
ARMulator	23.8 sec	59.3 sec	125 sec

In this table, the first row labeled *"CPU Utilization for the JPEG Task"* indicates the percentage of CPU time allocated to the JPEG task. 100% utilization means that there is no other application task. Even in this case, there exist a few system tasks such as the timer and the profiler. 50% means that another task exists while 25% means that three tasks exist in addition to the JPEG task. The application tasks running concurrently with the JPEG application include an infinite loop whose body is empty. All of these application tasks had the same priority as the JPEG application. Due to this, all of the application tasks (including the JPEG task) were scheduled in a time-sharing manner. The timer tick was set to 10 milliseconds, so that the application tasks switch every 10 milliseconds. Table 1 demonstrates that our cosimulator is three or four orders of magnitude faster than the ISS.

Next, we changed hardware/software partitioning. IDCT was moved from software to hardware, and we described a functional simulation model of the IDCT function in C. The cosimulation time is shown in Table 2.

Table 2. Cosimulation time of the JPEG decoder. IDCT is implemented in hardware, and is described as a functional simulation model in C.

CPU Utilization for the JPEG task	100%	50%	25%
Our Cosimulator	46 sec	49 sec	97 sec
ARMulator	71 sec	94 sec	141 sec

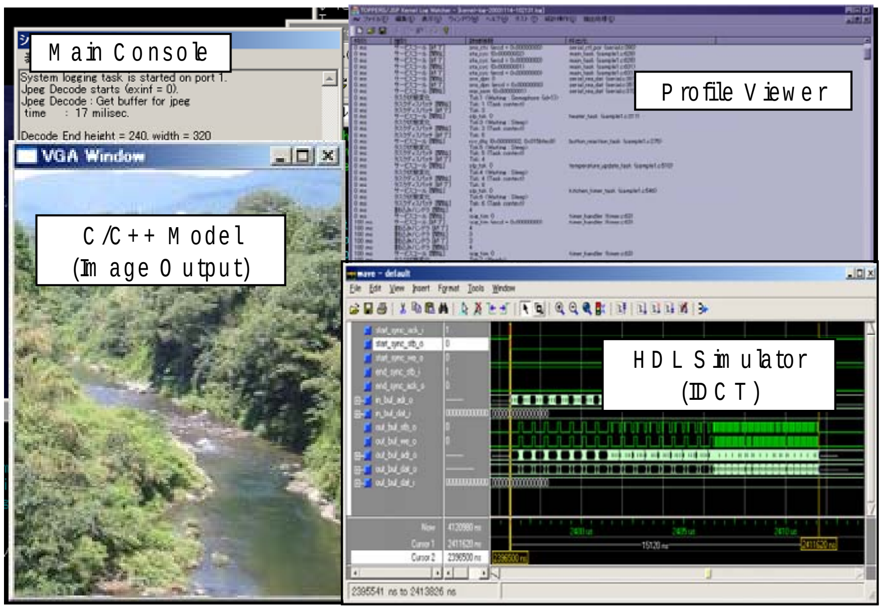
With our cosimulator, most of the cosimulation time was spent for communication between the software and the hardware when the CPU utilization was 100%. When the CPU utilization was 50% and 25%, the simulation time presented in Table 2 includes the time for simulating the other tasks.

Finally, we designed an IDCT circuit at the register-transfer level in VHDL. The IDCT simulation model was replaced with an HDL simulator, ModelSim [1], to simulate the RT-level IDCT design. Note that the JPEG application code did not require any modification from the one used with the C model. The simulation speed is shown in Table 3.

Table 3. Cosimulation time of the JPEG decoder. IDCT is implemented in VHDL, and is executed with an HDL simulator.

CPU Utilization for the JPEG task	100%	50%	25%
Our Cosimulator	182 sec	327 sec	333 sec
ARMulator	248 sec	961 sec	2802 sec

A snapshot of this cosimulation is presented in Figure 3, where the software model (i.e., the RTOS model and the application tasks), the HDL simulator, the C/C++ simulation model of the display function, and the profiler were running simultaneously, with communicating with each other through IPCM.



```
System logging task is started on port 1.
Jpeg Decode starts (exinf = 0).
Jpeg Decode : Get buffer for jpeg
 time    :  17 milisec.

Decode End height = 240, width = 320
```

Figure 3. A snapshot of JPEG simulation with the RTOS-centric cosimulator.

5. CONCLUSIONS

RTOSs have become one of necessities in the design of complex embedded systems. In order to validate the overall functionality of such embedded systems, RTOSs need to be simulated together with application software and hardware. We have developed a fast, flexible RTOS-centric cosimulator for embedded system design. Our cosimulator features complete supports of RTOS services, native execution of software, cosimulation with functional hardware models in C or C++, and cosimulation with HDL simulators. In this paper we have described principles and implementation of our cosimulator. We have also shown a case study using a JPEG decoder example to demonstrate the effectiveness of the cosimulator.

At present, the timing of communication and synchronization is inaccurate in our cosimulator. Improvement of timing accuracy remains as one of our future work.

REFERENCES

[1] Mentor Graphics Corporation, http://www.mentor.com/.

[2] H. Takada and K. Sakamura, "μITRON for small-scale embedded systems," *IEEE Micro*, vol. 15, no. 6, pp. 46-54, Dec. 1995.

[3] ITRON, http://www.assoc.tron.org/itron/.

[4] D. Desmet, D. Verkest, and H. De Man, "Operating system based software generation for systems-on-chip," *Proc. of Design Automation Conference (DAC)*, 2000.

[5] A. Gerstlauer, H. Yu, and D. Gajski, "RTOS modeling for system level design," *Proc. of Design Automation and Test in Europe (DATE), Embedded Software Forum*, 2003.

[6] SpecC Technology Open Consortium, http://www.specc.org/.

[7] H. Tomiyama, Y. Cao, and K. Murakami, "Modeling fixed-priority preemptive multi-task systems in SpecC," *Proc. of Proc. of Workshop on Synthesis and System Integration of Mixed Technologies (SASIMI)*, 2001.

[8] Y. Yi, D. Kim, and S. Ha, "Virtual synchronization technique with OS modeling for fast and time-accurate cosimulation," *Proc. of Int'l Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, 2003.

[9] F. Herrera, H. Posadas, P. Sanchez, and E. Villar, "Systematic embedded software generation from SystemC," *Proc. of Design Automation and Test in Europe (DATE), Embedded Software Forum*, 2003.

[10] WindRiver Systems Inc., http://www.wrs.com/.

[11] S. Yoo, G. Nicolescu, L. Gauthier, and A.A. Jerraya, "Automatic generation of fast timed simulation models for operating systems in SoC design," *Proc. of Design Automation and Test in Europe (DATE)*, 2002.

[12] I. Bacivarov, S. Yoo, A. A. Jerraya, "Timed HW-SW cosimulation using native execution of OS and application SW," *Proc. of Int'l High-Level Design Validation and Test Workshop (HLDVT)*, 2002.

[13] SystemC Open Initiative, http://www.systemc.org/.

[14] Microsoft Corporation, http://www.microsoft.com/.

[15] ARM Corporation, http://www.arm.com/.

[16] TOPPERS Project, http://www.toppers.jp/.