A SystemC Refinement Methodology for Embedded Software

Jérôme Chevalier, Maxime de Nanclas, Luc Filion, Olivier Benny, Mathieu Rondonneau, and Guy Bois
École Polytechnique de Montréal

El Mostapha Aboulhamid
University of Montreal

Editor's note:
This article presents a design environment that provides an interface for user-written SystemC modules that model application software to make calls to a real-time operating system (RTOS) kernel and cosimulate with user-written SystemC hardware modules. The environment also facilitates successive refinement through three abstraction layers for hardware-software codesign suitable for embedded-system design.

—Sandeep Shukla, Virginia Tech

Technology advances have tremendous effects on chip complexity. For example, the evolution from 0.5- to 0.18-micron technology has freed 88% of chip space.[1] This trend continues to follow Moore's law with 130- and 90-nm technologies. It is now possible to manufacture chips with 100 million transistors—theoretically enough to fit the logic of more than 1,000 32-bit reduced-instruction set computing (RISC) processors on a single die. Leveraging these capabilities is a major challenge, which has led to the adoption of SoC design. Complexity, combined with the narrowing of the time-to-market window, has motivated SoC designers to reuse IP blocks and build designs around platforms.[2]

Unlike ASICs, in which all desired functionality is hardwired, platforms contain programmable devices (for example, processors), and some specific functions are hardwired to meet performance requirements (such as latency and power dissipation requirements). The programmability of platforms offers flexibility, extensibility, and adaptability to new requirements. At the same time, it presents a new challenge in the design process: Designers must define higher abstraction levels that allow system modeling. They must use description languages that handle both hardware and software components to describe communication between these components and capture constraints. For instance, a transaction-level model (TLM) abstracts the functionalities of a hardware-software system, improving designer productivity. Rather than dealing with numerous microarchitecture details, as they do with RTL models, designers using a TLM can rapidly design a macroarchitecture meeting expected performance requirements.

To address abstraction in the context of platform-based design, researchers have introduced the concept of the virtual platform.[3] Virtual platforms are very high-level functional models of the SoC with no specific implementation details. Modern design tools based on these virtual platforms must support IP block integration, architectural exploration, and communication mapping.

Another solution to the abstraction problem is object-oriented languages such as C++. They reduce design and verification time by raising the system specification abstraction level, increasing component reusability, and allowing both software and hardware to be specified in the same syntax. Currently, SystemC, a C++ library,[4] seems to be the leader in system-level modeling. The SystemC approach consists of progressive specification refinements. It is not a design methodology but rather provides various abstraction levels useful for specification capture in early design flow stages. The design cycle starts with an abstract, high-level, untimed functional or timed functional (UTF/TF) representation that the designer refines to an implementation model by going through various intermediate levels.[5-8] Hardware refinement with SystemC is supported by commercial multilevel design tools.[6,9] On

0740-7475/06/$20.00 © 2006 IEEE Copublished by the IEEE CS and the IEEE CASS

the other hand, SystemC lacks the necessary constructs to provide software refinement and software scheduler emulation. This causes integration problems and increases development time. The use of software abstraction levels would help avoid this situation. The use of operating system constructions and properties earlier in the development process would be helpful at the partitioning level.

This article proposes a SystemC refinement methodology that focuses on using software abstraction levels to facilitate hardware-software partitioning. Its first contribution is that it enhances high-level embedded-software modeling support by encapsulating real-time operating system (RTOS) functionalities in a SystemC-RTOS (SC/RTOS) interface. The choice of an RTOS rather than a general-purpose operating system is appropriate for embedded applications, mainly to support preemption or priority scheduling.

The methodology's second contribution is that it lets designers easily move modules (threads) from hardware to software (and vice versa) without affecting intermodule communications. A third contribution is its integration of the SC/RTOS interface and the communication mechanism in a unique methodology containing multiple abstraction levels.

Problem definition

A major area of interest in transaction-level modeling is embedded software. A TLM enables software development, which can be a lengthy process because it takes place in parallel with rather than after hardware development. For a large class of embedded systems, an RTOS also provides a useful abstraction interface between applications with real-time requirements (usually satisfied by a preemptive scheduler) and the processor. In other words, integrating an RTOS simplifies software development and reusability.

SystemC supports an approach in which designers use a unified specification language based on C++ to iteratively refine specifications toward a final implementation. However, for some classes of applications modeled with SystemC, it is not currently possible to completely model the target architecture's software behavior.

Indeed, the SystemC core language doesn't support thread priority assignment, because its simulator doesn't offer all the necessary functions usually found in an RTOS, such as preemption or priority scheduling. Joint hardware-software refinement is thus a tedious task in SystemC 2.1. A possible way to extend the SystemC core language is to provide better software support. Unfortunately, such support is not planned for development. SystemC 2.1 consequently offers tools that synthesize hardware modules,[9] but its software synthesis is limited.

Several research efforts have concentrated on simultaneously simulating software and hardware realistically with SystemC to perform partitioning with a better understanding of the system. But currently there is no method of easily simulating various hardware-software configurations at a high level to obtain results leading to an optimum system partitioning. To reach this goal, a methodology must meet two main requirements:

- It must provide a realistic simulation of the entire system, letting the designer validate or invalidate a partition choice.
- It must let the designer move modules between hardware and software without changing modules' application code and bus or processor connections (transparency) to explore various configurations rapidly and effortlessly.

A few research proposals have achieved the first requirement (see the "Related work" sidebar), but to the best of our knowledge, no research works or commercial tools meet the second requirement (except our SPACE platform[10]).

SPACE platform

The approach we present here helps architecture exploration with a hardware-software refinement methodology for SoC development. We based our methodology on a virtual platform called SPACE (SystemC partitioning of architectures for codesign of embedded systems), which allows simulation and performance assessments to facilitate architectural exploration, particularly for hardware-software partitioning. The building of SystemC modules in SPACE follows a coding guideline that facilitates transparent module transfers between hardware and software. SPACE allows fast exploration of a large choice of possible partitions and lets the designer simulate each partition to find the optimum one. Because SPACE models a system at the transactional level, it increases simulation speed (100 times faster than simulation with an equivalent RTL model[11]).

As Figure 1 shows, the SPACE platform's configurable architecture contains strategic components for hardware-software cosimulation at the TF level. Hardware modules communicate through a TLM, and software modules connect to an SC/RTOS interface that translates SystemC-specific function calls into appropriate RTOS-specific function calls. The software bina-

Related work

Three proposed methodologies meet the simulation requirement necessary for optimum system partitioning. Nicolescu et al. propose a component-based SoC methodology using a wrapper generation flow to automatically link various heterogeneous cores of different abstraction levels in the same simulation.[1] The simulation uses a golden abstract architecture resulting from architecture exploration. For software, the operating system architecture level provides an abstract RTOS for which services are used only in the next level, the driver level. At the driver level, the implementation of operating system services is fixed, but subroutine interrupt services are abstracted. At the lower level, the instruction set architectural level, software is described in assembly code specific to the fixed hardware.

Fummi et al. present two cosimulation methodologies, using SystemC on an instruction set simulator (ISS) as a processor model.[2] The first methodology, GDB-Kernel (GNU debugger kernel), works at the SystemC kernel level and exploits the GNU suite's potentialities. The second, Driver-Kernel, uses features offered by the operating system running on the ISS.

Both the Nicolescu et al. and Fummi et al. methodologies require that modules simulating the software be modified specifically; it is thus not possible to easily transfer modules between hardware and software to test various configurations.

Herrera et al. propose a SystemC code for system-level specification and for embedded software generation after hardware-software partitioning.[3] Software modules are written in SystemC and can be simulated at a high level. The authors also present an SC/RTOS interface that enables module scheduling based on an RTOS, but they don't propose an environment to facilitate architectural exploration.

References

1. G. Nicolescu et al., "Validation in a Component-Based Design Flow for Multicore SoCs," *Proc. 15th Int'l Symp. System Synthesis* (ISSS 02), ACM Press, 2002, pp. 162-167.
2. F. Fummi et al., "Native ISS-SystemC Integration for the Co-Simulation of Multi-Processor SoC," *Proc. Design, Automation and Test in Europe* (DATE 04), vol. 1, IEEE Press, 2004, pp. 564-569.
3. F. Herrera et al., "Systematic Embedded Software Generation from SystemC," *Proc. Design, Automation and Test in Europe* (DATE 03), IEEE Press, 2003, pp. 142-147.

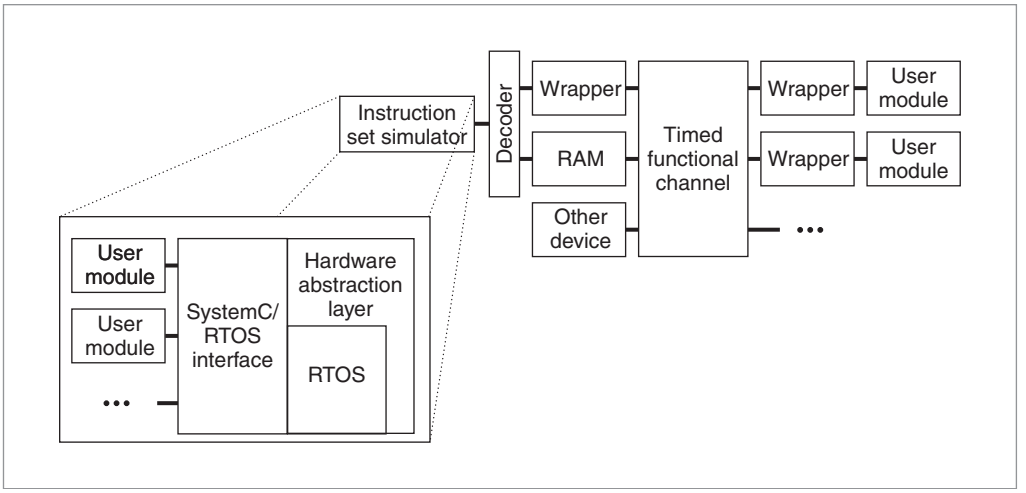

Figure 1. SPACE platform architecture. User modules (threads) can be software (left) or hardware (right).

ry code, which runs on an instruction set simulator (ISS) encapsulated in SystemC, consists of three components: the user application's software (user modules), the SC/RTOS interface, and the RTOS and its port.

One advantage of SPACE is that it enables an RTOS to schedule software SystemC threads. This property reflects the final implementation, so the software simulation can give realistic behavior and timing information very early in the design flow. The SystemC-RTOS interface creates the same environment for software and hardware modules, facilitating their transfer during partitioning.

The original SPACE methodology[10] includes a two-level refinement process. The designer first develops and validates the application at a higher level called the UTF level. At this level, there is no platform architecture or hardware-software partition. The user modules are simply connected to a routing channel. After completing and validating the user application, the

designer groups modules into two partitions and connects them to the configurable architecture at the TF level.

The earlier SPACE version didn't provide several abstraction levels with different degrees of accuracy. Thus, it limited software refinement to two extreme levels: the UTF level with untimed simulation, no platform, no RTOS, and no processor and the TF level with a complete platform, device models, an RTOS, and an ISS. In this article, we add to the SPACE platform an intermediate software abstraction level, similar to a driver level or a GDB (GNU debugger) level (see sidebar). We integrate the resulting three software levels into a partitioning methodology that meets the two requirements mentioned earlier.

Abstraction levels and methodology

Figure 2 illustrates the three software levels provided by our methodology. These abstraction levels let software modules be part of the partitioning process. Each level consists of different software components.

The first level, L1, focuses on system design. Designers specify the application and validate functionality with the normal SystemC simulator. At L2, the designer partitions the application into software and hardware modules. The designer simulates the hardware with the SystemC simulator and schedules the software with an RTOS emulation process encapsulated in the SystemC-RTOS interface. At L3, modules previously tested at L1 and L2 are reused without modification and placed in a more detailed architecture that supports cycle-accurate simulation (ISS). This implies that the RTOS encapsulated in the SystemC-RTOS interface is now running on an ISS at a chosen processor frequency. The ISS is connected to a TF channel and scheduled by the SystemC simulator.

Figure 3 illustrates the components that the methodology includes in a single process at L2 or in a single binary executable for the ISS at L3:

- user application written in SystemC—that is, software modules;
- SystemC-RTOS interface in charge of initialization sequence, communication between the RTOS and the SystemC application, and communication between software and hardware modules (through the communication manager);
- RTOS, containing the software task scheduler and the processor-dependent hardware abstraction layer (HAL). We use two different HALs: one for the RTOS to be executed as a host process in L2 and another to operate the ISS processor and platform hardware devices in L3.

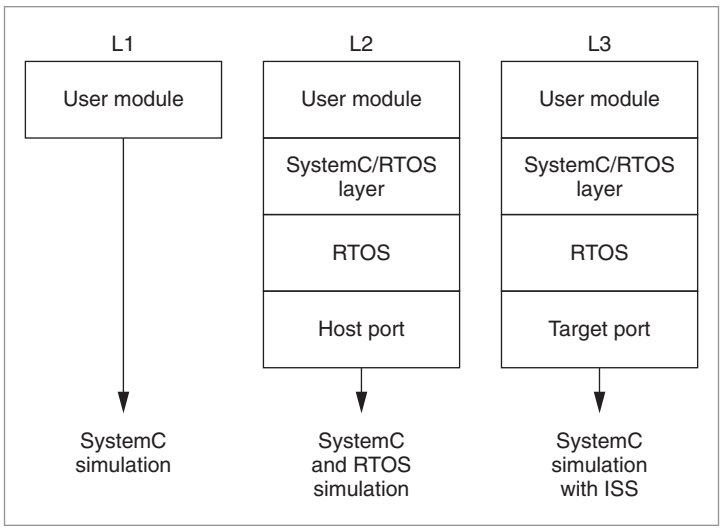

Figure 2. Software levels.

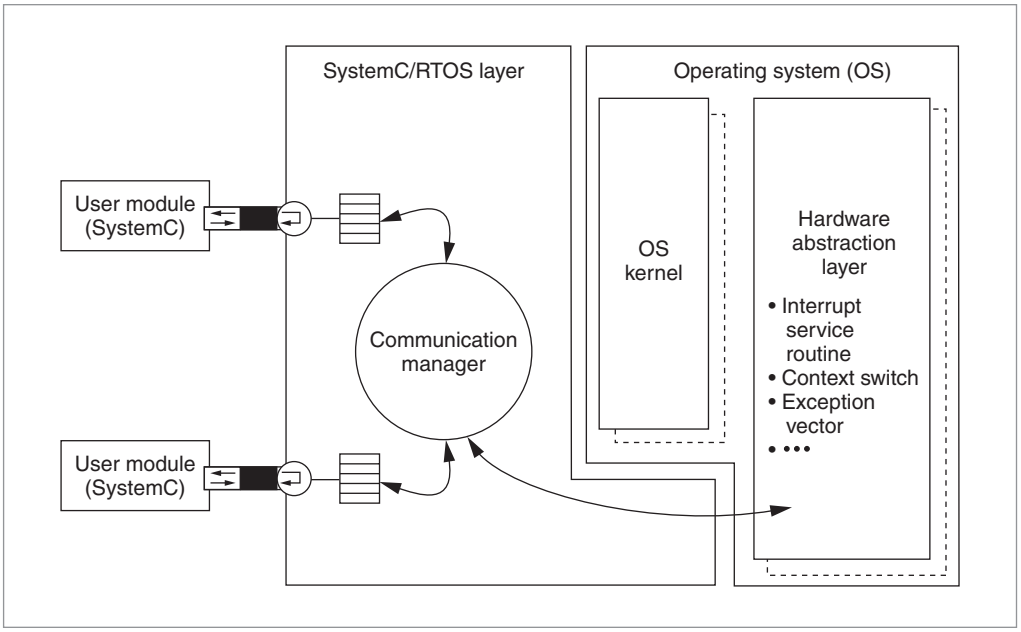

Figure 3. Software environment connecting SystemC software user modules to the RTOS.

Table 1. Mapping between SystemC functions and RTOS functions.

SystemC API		Generic API	MicroC/OS-II API
SC_THREAD		taskCreate	OSTaskCreate
SC_METHOD		taskCreate	OSTaskCreate
sc_start		startScheduler	OSStart
sc_wait		timeDelay	OSTimeDelay
sc_mutex	sc_mutex	mutexCreate	OSMutexCreate
	lock	mutexTake	OSMutexPend
	trylock	mutexTake	OSMutexAccept
	unlock	mutexGive	OSMutexPost
sc_semaphore	sc_semaphore	semCreate	OSSemCreate
	wait	semCTake	OSSemPend
	trywait	semCTake	OSSemAccept
	post	semCGive	OSSemPost
sc_fifo	sc_fifo	queueCreate	OSQCreate
	read	queueRcv	OSQPend
	nb_read	queueRcv	OSQAccept
	write	queueSend	OSQPost
sc_event	sc_event	semBCreate	OSSemCreate
	pend	semBTake	OSSemPend
	notify	semBGive	OSSemPost
sc_port<space_if>	read	Communication manager functions	
	nb_read		
	write		
	nb_write		

The RTOS we chose for the first implementation of our methodology is MicroC/OS-II.[12] It offers all the RTOS advantages: a preemptive kernel, a priority-based task scheduler, and an interrupt system. We selected MicroC/OS-II for its low complexity and source code availability. It has been ported successfully to a wide range of processors and is provided in the SPACE platform.

The SC/RTOS interface has several functions. First, it is responsible for the initialization phase. This function is very significant because it lets us build the entire software environment in which the application will execute. Initialization also includes calling the RTOS (MicroC/OS-II) to create MicroC/OS-II tasks and start the RTOS scheduler.

The SC/RTOS interface's second role is to provide mapping between system functions proposed in SystemC 2.1 and RTOS functions through a generic API. Table 1 presents a partial list of these functions, which manage processes, modules, interfaces, channels, and communication ports.

Third, the SC/RTOS interface provides a communication manager, which establishes connections between software modules and platform hardware. It answers task requests to communicate with other software or hardware modules. Thus, intermodule communications use specific read/write methods. The software communication manager handles communications between any connected modules (hardware or software) and provides the same communication model as the hardware channel. A module connected to this manager will be registered as a software module. The SPACE platform, not the software communication manager, handles hardware-to-hardware communications.

Figure 4 summarizes the details of the three platform levels and their roles in our methodology, as described in the remainder of this section.

Level L1: Functional validation

According to Donlin,[7] we can consider L1 an algorithmic level. Its goal is to allow quick validation of an application. We achieve this by keeping the communication model as simple as possible and focusing on untimed message passing. These characteristics let us validate the system without the entire platform—that is, without a bus protocol, a microprocessor, and its operating system.

Figure 5 shows the L1 connections. At this level, the application is specified in SystemC but not partitioned. The target-platform methodology forces every module to use only thread constructions (such as SC_THREAD) because the library function wait() or any of its variants can suspend execution of these constructions.[5] All modules are restricted to the use of one advanced SystemC port, making it easier to move them from the hardware partition to the software partition (and vice versa) in the lower levels. The port interface allows both blocking and nonblocking semantics, as described in Table 1. Once the code is written and the functionality validated, the next step is partitioning. You could also connect to the functional channel predefined blocks such as a memory (RAM) and devices for debugging or I/O purposes.

To ensure that communication is preserved whether we decide that modules are hardware or software, each module has a unique identification number. The system designer chooses this ID and assigns it for the entire par-

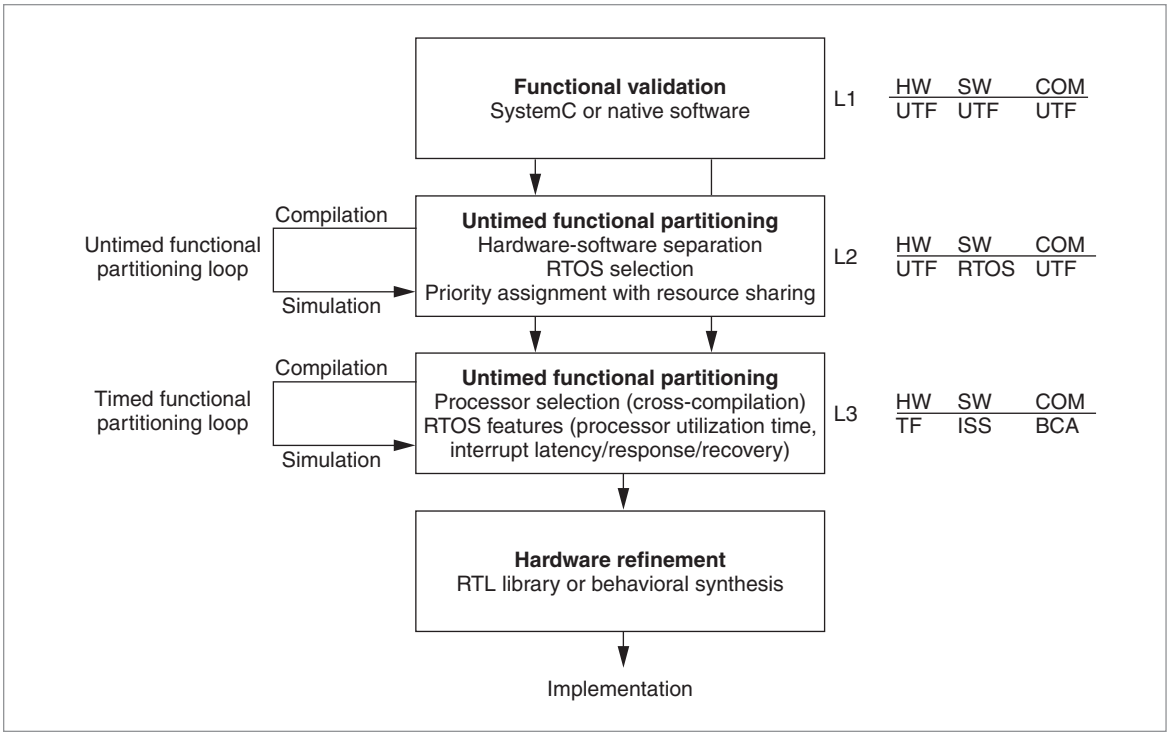

Figure 4. Methodology and verification issues handled at each SPACE platform level.

titioning and refinement processes (L2 and L3). Our implementation achieves communication with a network-like protocol, encapsulating messages with a structured header containing the sender ID, the receiver ID, and the message size so that messages can be routed dynamically. Because the communication interface is implemented at each abstraction level, any module moved from hardware to software (or vice versa) through L2 or L3 requires instantiation and connection updates only in the main top-level SystemC module. Thus, by providing the appropriate ID, a sender or receiver can always reach a module, whether its current assignment is software or hardware. This ensures that we meet the transparency requirement mentioned earlier.

Level L2: Untimed functional partitioning

Donlin[7] considers L2 a communicating-processes level because it is implementation independent, and its communication details are still unknown. At this level, application partitioning takes place. Initially, using personal experience, the designer places user modules either in the software or the hardware part of the system. SPACE makes it possible to modify this partitioning choice anytime by recompiling the two main files (hardware and software partitions) and cosimulating the new partitions, as illustrated in Figure 4 by the UTF partitioning loop. The SystemC simulator executes threads inside the hardware modules, whereas a different process executes software threads, which are scheduled by the RTOS running on Linux or Windows as a host process. Thus, the host operating system executes the RTOS simulator, which provides the preemptive scheduler based on priorities, even though it is not executed on the target processor. The SC/RTOS interface enables execution of the SystemC user code.

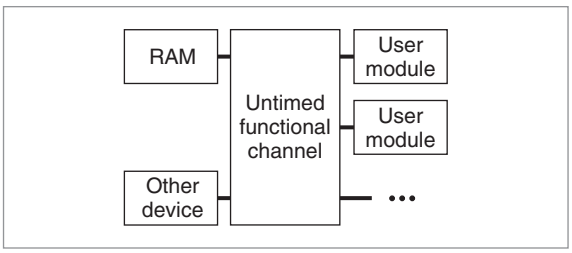

Figure 5. L1 connections.

L2 provides the first simulation results of a partitioned system without any architecture details. At this level, tasks receive various priorities; critical tasks necessarily have higher priorities than other tasks.[12] Therefore, we can verify the true preemptive scheduler priority assignment immediately to ensure acceptable behavior such as deadlock-free execution. L2 results in faster simulation than the

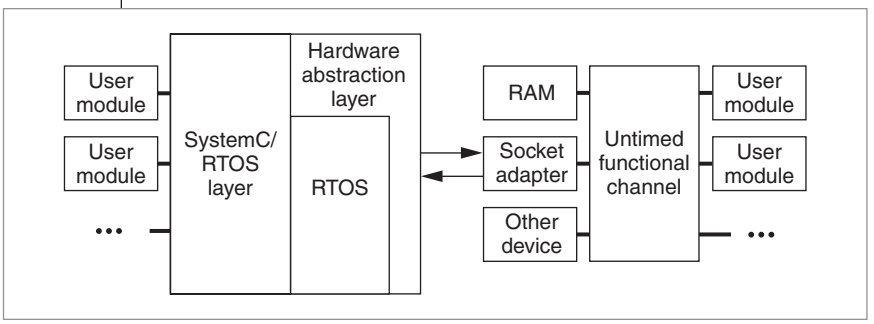

Figure 6. L2 hardware-software communication.

lower level, L3, but also the compromise of unverified timing constraints. The application is easier to debug because the compiled code is compatible with the host debugger, and no cross-compilation intervenes.

The software simulator is a single process running the application code, using the SystemC API, atop the RTOS. The software and hardware simulation processes communicate and synchronize through two message-oriented sockets. Figure 6 shows the structure and communication between hardware and software application processes. These processes are independent, and the simulation synchronizes on message passing. The simulation uses two message servers: one on the first socket on the software side and one on the second socket on the hardware side. Each side's message server (hardware or software) listens and replies to incoming requests from the other side. The hardware process (including the SystemC simulator) or the software process (including the RTOS) sends its requests to the corresponding socket. As with all message-passing communication in SPACE, only the write requests transit through the sockets. Read requests, except memory accesses, remain local within the process (software or hardware).

On the hardware side, the message server receives write requests from software and transmits them to the UTF channel. On the software side, the message server receives write requests from hardware and triggers a Posix signal[13] to interrupt the software. A signal handler is executed and the request is transmitted to the communication manager. If a software thread was waiting for the message and represented the highest priority task, the RTOS reschedules it at the next clock tick interrupt emulated by a Posix timer. Finally, a RAM module is also available as shared memory.

Level L3: Timed functional partitioning

We can consider L3 a programmer's timed view level because further architectural details are fixed, and we can resolve the communication structure with a given interconnection type.[7] Here, we reuse modules previously tested in L1 and L2 without modification and place them in a more detailed architecture. We can modify the partition choice here, using results from L2 simulations.

All software modules are cross-compiled to be executed on the ISS. The only difference in the software partition, in comparison with L2, is the RTOS hardware abstraction layer (HAL). Rather than being encapsulated in a host process as in L2, the HAL in L3 allows the operating system to operate in the target-architecture environment. This includes managing the given architecture's resources (registers, memory, interrupts, and timer).

As mentioned earlier, the communication manager uses specific software-level functions to send or receive messages to or from hardware. Through the communication manager, a module can directly access peripherals (memory, timer, and so on) or send a message to another module, whether that module is in software or hardware. When a hardware module needs to send a message to software, an interrupt is triggered and a routine is executed to receive the message. This time, the interrupt source is not a Posix signal as in L2 but a real interrupt trapped by the ISS. The drivers then send incoming messages to the communication manager so that the software modules can receive them. Also, a true timer generates clock tick interrupts. Figure 1 shows how software and hardware modules intercommunicate at this level.

To debug the software in L3, we integrated the Insight graphical user interface debugger (http://sourceware.org/insight/), based on GDB from GNU (http://sourceware.org/gdb/), into the SPACE environment. We modified the ISS to provide remote debugging via a socket connection. Insight communicates with the ISS, so two debuggers can run simultaneously for coverification. One debugger is for the hardware simulated by SystemC, and the other is for the software executed by the ISS running the RTOS.

For a first experiment, we chose the ARM7 processor (http://www.arm.com), wrapped in a SystemC module with clock, reset, interrupt request, and I/O data ports. We added a SystemC wait() statement on the clock signal in the ISS main loop to synchronize SystemC with the ISS clock. We redirected the ISS memory access functions toward the I/O data port.

Communication takes place through this port using the read() and write() functions. These functions receive two arguments, address and data, permitting an efficient and simple high-level model of the I/O-mapped memory bus. Because the memory (through its decoder) is the interface of this port, reading and writing calls complete in only one delta cycle (unless wait() is inserted to simulate a nonzero access time), making fast simulation possible.

Therefore, we can simulate hardware-software partitions selected at L2 more accurately at L3. We can also pass directly from L1 to L3 when L2 is not required. The TF partitioning loop at L3 (Figure 4) lets us rapidly change system partitions and reevaluate performance and cost metrics through simulation. For each selected configuration, the user starts the simulation, stops it at a precise moment, and then notes the clock cycle count at this time—for instance, by using the function sc_simulation_time(). In contrast with L2, L3 software modules consume clock cycles while being executed if the ISS is cycle accurate or at least cycle approximate. The target processor model determines how much CPU time the application is using. We can also consider interrupt latency, response time, and recovery time. To reflect potential RTL implementation, we can annotate hardware modules with delays by using wait() statements. Currently, metrics such as area estimation, power dissipation, and memory use are not part of the methodology, but we will consider them in future development.

At the end of TF partitioning and evaluation, the hardware-software configuration we estimate as best is the one that satisfies design constraints while minimizing hardware. We want to minimize hardware because software offers more flexibility throughout product life.

Results

Figure 7 shows functional diagrams of two implementation examples that demonstrate the benefits of the SPACE platform and our associated methodology.

RF filter

The first example, given in Figure 7a, models an adaptive digital filter for RF applications. It consists of five modules: producer, filter, analyzer, controller, and multiplexer. It produces and filters integer data and stores the data in a buffer (shared memory). When the buffer is full, the controller tells the multiplexer to store data somewhere else in memory and asks the analyzer module to use previously stored data to perform a simple calculation. The result is needed to adjust filter coefficients.

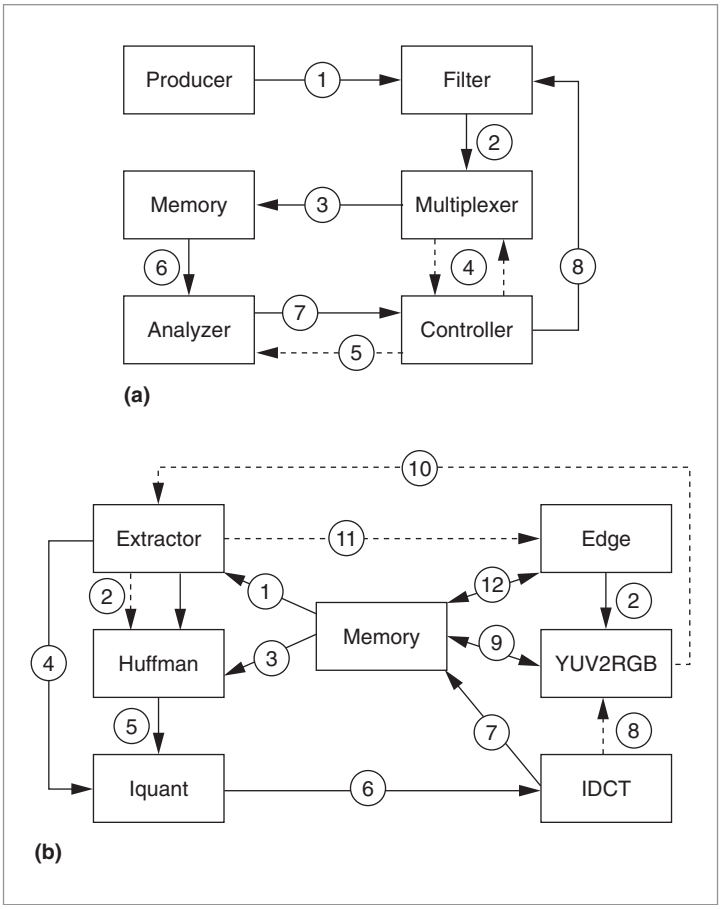

Figure 7. Examples: RF filter (a) and picture processor (b). Dashed lines represent control signals, and solid lines represent data transfers.

First, we simulated this example at the UTF L1 level to test its functionality. Then, we proceeded to the L2 level to make a first hardware-software partition choice. With five modules, there were 32 possible partitions to investigate through simulation. Table 2 presents results for the best partition (named PART). This partition decision places the controller and analyzer modules in the software partition and the producer, filter, and multiplexer modules in the hardware partition. We also executed the example at the L3 level with a 100% hardware solution and again with the same partitioning as in level L2 (PART). This example's constraints were to maintain the data flow without data loss, and to process the entire data buffer before accumulating the next buffer. We cannot meet these constraints if the filter migrates from hardware to software from the PART partition, because this partition has already used 98% of the CPU.

Figure 8 illustrates the support of the architectural exploration at level L3. The left window (Figure 8a) pre-

Table 2. Results for an RF filter on a 1.8-GHz Pentium IV, 512-Mbyte Rambus, Windows 2000, for the untimed functional (UTF) layer and the timed functional (TF) layer (where PART represents the best partition).

Layer	Level	Duration (s)	Simulation cycles (millions)	CPU usage (%)*
UTF	L1	8	N/A	N/A
	L2 PART	31	N/A	90
TF	L3 100% hardware	45	5.2	N/A
	L3 PART	49	5.2	98

* Based on the statistics task of MicroC/OS-II.[12]

sents the platform architecture; hardware modules connect to TF_channel1, whereas software modules run on arm_iss. The right window (Figure 8c) shows how modules can switch from software to hardware in a single click. For each partition, the user generates and compiles two main files (hardware and software) for cosimulation, using the Generate button at the bottom of the main tool dialog (Figure 8a). The middle window (Figure 8b) shows the SystemC top module (main) for the hardware part.

Picture processor

In the previous example, running the ISS time simulation did not represent an important difference in duration from running a host simulation. The reason for this was mainly that the filter's rather low complexity meant we could not neglect the initialization phase. Our goal with the second example was to present a more complex application for which higher abstraction levels are much faster to simulate.

We modeled a picture processor application (Figure 7b) that decompresses a JPEG image and detects the image contours. The application consisted of six modules. Table 3 summarizes our simulation results at various levels with various partitions. The input test bench consisted of 10 (128 × 128)-pixel JPEG images (each pixel uses 4 bytes). Each image generates about 200,000 transactions on the channel. In addition, modules

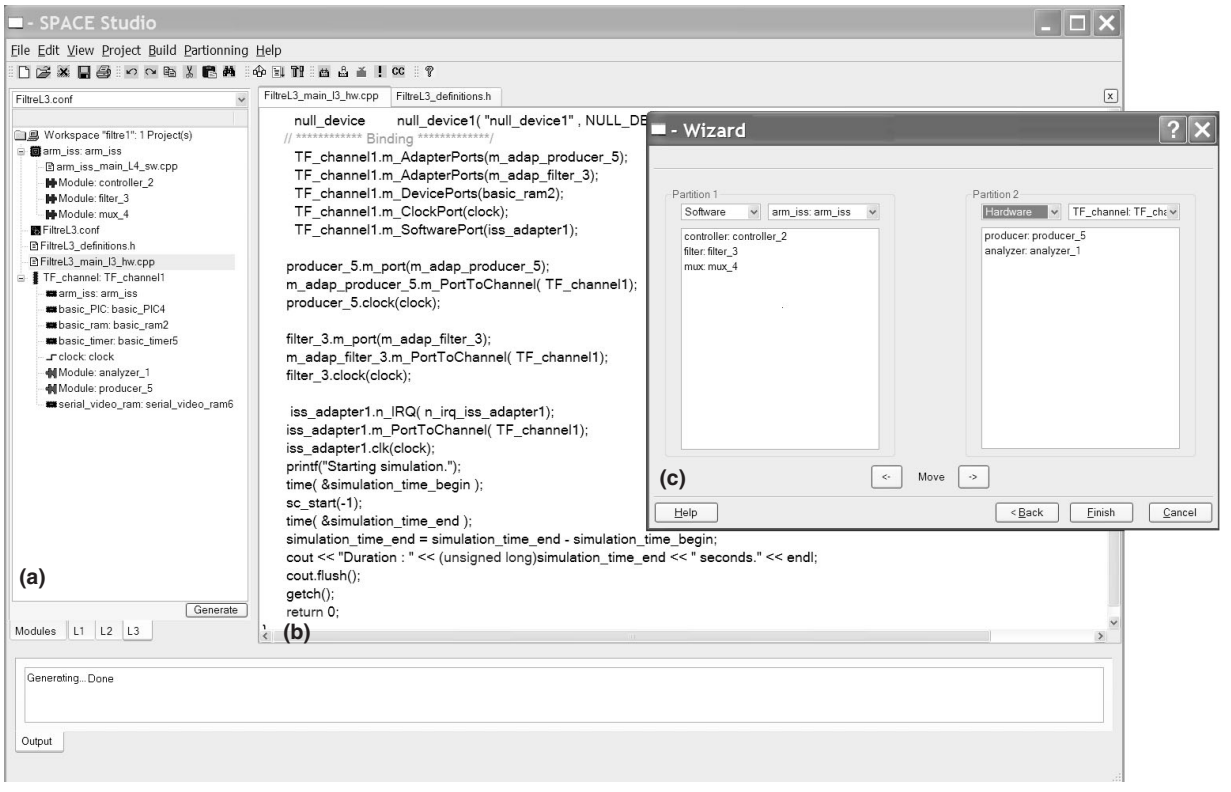

Figure 8. SPACE Studio for support of architectural exploration: platform architecture (a); SystemC top module (main) for hardware (b); software-to-hardware switching wizard (c).

Table 3. Results for a picture processor on a 1.8-GHz Pentium IV, 512-Mbyte Rambus, Windows 2000.

Layer	Level	SW partition	HW partition	Duration (s)	Simulation cycles (millions)
UTF	L1	N/A	N/A	48	N/A
	L2 PART	Extractor	Huffman, YUV2RGB, IDCT, Edge, Iquant	218	N/A
		YUV2RGB	Huffman, Extractor, IDCT, Edge, Iquant	333	N/A
		Extractor, YUV2RGB	Huffman, IDCT, Edge, Iquant	438	N/A
		Huffman	Extractor, YUV2RGB, IDCT, Edge, Iquant	1,092	N/A
TF	L3 PART	Empty (none)	Extractor, Huffman, YUV2RGB, IDCT, Edge, Iquant	110	0.6
		Extractor	Huffman, YUV2RGB, IDCT, Edge, Iquant	1,377	5.9
		YUV2RGB	Huffman, Extractor, IDCT, Edge, Iquant	2,547	14.5
		Extractor, YUV2RGB	Huffman, IDCT, Edge, Iquant	4,359	20.4
		Huffman	Extractor, YUV2RGB, IDCT, Edge, Iquant	13,193	199.9

exchange 20-byte commands to synchronize the execution flow. As a time reference, both the hardware and the processor use a 100-MHz clock. Table 3 presents performance results for four L2 partitioning configurations and five L3 partitioning configurations. Note that 100% in hardware at L2 is equivalent to L1.

Analysis of the results clearly demonstrates that higher abstraction levels are much faster to simulate. Indeed, for this application, L1 presents a speedup of 50 to 1,000 compared with L2, which simulates 600% to 1,300% faster than L3.

Introducing an RTOS slows execution time, and we must overcome this problem. We believe that the L2 level provides one solution confirmed by our results. However, simulation times vary considerably between levels and configurations. The L1-level communication system consists of a crossbar model enabling higher speeds. The L2 level is faster than the L3 level partitioned with an ISS because the software executes in host mode. But for applications showing a high communication rate, hardware-software socket communication can bottleneck the L2 level, as the picture processor simulation showed. We must experiment with more efficient interprocess communication mechanisms (such as shared memory) in the future.

There are slight variations in L3 simulation time between the purely hardware configuration and the ISS configuration. The ISS simulation is slower because the ISS constitutes an additional SystemC thread, often needing to be simulated and thus lengthening the delta cycles' duration. This clearly identifies a partition choice prohibited by performance constraints.

Our next step is to try other RTOS alternatives and support multiple processors. At level L2, it would be interesting to have multiple software processes, each running a different RTOS instance. Also, an intermediate level between L2 and L3 is under development. According to Donlin, it corresponds to a timed communication-processes level.[7] At this level, the RTOS will execute as a host process, but hardware and software modules will be connected on a TF channel and simulated in the same process so that they both advance in a locked-step cycle. We expect a timed (and fast) simulation. Also, we are extending our methodology to another level, L4 (below L3), where we can "clean" the software by removing SystemC skeleton and function calls to obtain the minimum code size for software synthesis. By combining this software with RTL hardware models, we can map applications to real physical systems. ■

References

1. F. Charlot, "Mastering Reusable Designs," *Électronique*, vol. 99, no. 1, Jan. 2000, pp. 70-74 (in French).
2. G. Martin and H. Chang, *Winning the SoC Revolution*, Kluwer Academic Publishers, 2003.
3. P. Hardee, "SystemC: A Realistic SoC Debug Strategy," *EE Times*, 3 June 2002, http://www.embedded.com/showArticle.jhtml?articleID=9900627.
4. Open SystemC Initiative (OSCI), *Language Reference Manual version 2.1*, Apr. 2005; http://www.systemc.org.
5. T. Grötker et al., *System Design with SystemC*, Kluwer Academic Publishers, 2002.
6. Synopsys Cocentric System Studio; http://www.synopsys.com.

7. A. Donlin, "Transaction Level Modeling: Flows and Use Models," *Proc. Int'l Conf. Hardware/Software Codesign and System Synthesis* (CODES + ISSS 04), IEEE Press, 2004, pp. 75-80.
8. L. Cai and D. Gajski, "Transaction Level Modeling: An Overview," *Proc. Int'l Conf. Hardware/Software Codesign and System Synthesis* (CODES + ISSS 03), IEEE Press, 2003, pp. 19-24.
9. D. Maliniak, "SystemC Closes the C-to-RTL Gap," *OSCI/OCP-IP Special Report/Supplement to Electronic Design*, 13 Jan. 2005; http://www.elecdesign.com/Issues/IssueID/300/300.html.
10. J. Chevalier et al., "SPACE: A Hardware/Software SystemC Modeling Platform Including an RTOS," *Languages for System Specification*, C. Grimm, ed., Kluwer Academic Publishers, 2004, pp. 91-104.
11. M. Caldari et al., "Transaction-Level Models for AMBA Bus Architecture Using SystemC 2.0," *Proc. Design, Automation and Test in Europe* (DATE 03), IEEE Press, 2003, pp. 26-31.
12. J.J. Labrosse, *MicroC/OS-II, The Real-Time Kernel*, 2nd ed., CMP Books, 2002.
13. B.O. Gallmeister, *Programming for the Real World, POSIX 4*, O'Reilly, 1995.

Jérôme Chevalier is pursuing a PhD in the Department of Computer Engineering at École Polytechnique de Montréal. His research interests include system-level design, hardware-software codesign, embedded software, and real-time operating systems. Chevalier has an MEng in computer science from École des Mines d'Alès, France. He is a student member of the IEEE.

Maxime de Nanclas is pursuing an MSc in the Department of Computer Engineering at École Polytechnique de Montréal. His research interests include high-level software-hardware modeling, cosimulation, and partitioning. De Nanclas graduated from École Polytechnique de Nantes, France.

Luc Filion is a PhD candidate in the Department of Computer Engineering at École Polytechnique de Montréal. His research interests include hardware-software codesign, system-level design, and embedded computer architectures. Filion has an MSc from École Polytechnique de Montréal.

Olivier Benny is an R&D system-level engineer at STMicroelectronics Canada. His technical interests include application mapping (for example, debugging and simulation tools) to embedded multiprocessor systems. He has an MSc in computer engineering from École Polytechnique de Montréal.

Mathieu Rondonneau is a firmware engineer at PMC-Sierra. His technical and research interests include system-level design, real-time systems, and operating systems. He has an MSc in computer engineering from École Polytechnique de Montréal.

Guy Bois is a professor in the Department of Computer Engineering at École Polytechnique de Montréal. His research interests include hardware-software codesign and coverification for embedded systems. Bois has a PhD in computer science from the University of Montreal.

El Mostapha Aboulhamid is a professor in the Department of Computer Science and Operational Research at the University of Montreal. His research interests include system-level modeling, formal high-level verification techniques, and formal refinement of hardware-software systems. Aboulhamid has a PhD in computer science from the University of Montreal.

■ Direct questions and comments about this article to Guy Bois, Dept. of Computer Engineering, École Polytechnique de Montréal, Montreal, Canada; guy.bois@polymtl.ca.