A Modular SystemC RTOS Model for Embedded Services Exploration

Emmanuel Huck   Benoit Miramond   François Verdier
ETIS - CNRS/UMR 8051
ENSEA, University of Cergy-Pontoise, France
Email: [firstname.name]@ensea.fr

Abstract— **System level modeling has been adopted for few years as a way to face the growing design complexity of embedded systems. In this systems the control of embedded applications is more and more often devoted to a Real-Time Operating System (RTOS). This RTOS can either be deployed in software or hardware, partially or completely, depending on the non-functional constraints of the global system. Consequently, these new design decisions concerning the implementation of the control must be taken early in the design flow for software and hardware parts.**

In this paper we present the structure of our high level RTOS model, built to rapidly explore and evaluate different operating system services strategies (as scheduling policies) and implementation choices. Indeed, modeling an RTOS in SystemC implies to take into account its dynamical mechanisms and to explicitly control the scheduling of the simulated processes. We give some experimental simulation results of the RTOS model corresponding to the exact behaviour of an embedded RTOS at the first stages of the design.

I. Introduction

The design of new embedded systems on a chip (SoC) now faces two main issues. Firstly, algorithm complexity continuously increases, implying more and more computing power. Secondly, the increasing complexity and heterogeneity of targeted hardware architectures make the corresponding systems and their control harder to design. Moreover, the software integration phase follows the same growing complexity.

In order to face these difficulties and the design constraints such as cost, time-to-market, or tools (un)availability, engineers have to take decisions very early in the design flow. Among their choices concerning the target architecture, they have to find a valid partitioning between software and hardware that respects real-time and dynamical constraints of the attended application. Automatic partitioning methods and tools have been shown to work theoretically. But it is not used practically because there is actually no way to quickly verify non-functional constraints such as performance, power consumption and so far in a predictable and error-prone manner before the prototype is realized. This often leads to overdimensioning the global architecture, leading to over costs, surface, and power consumption.

Moreover, hardware and software designs are made independently and it is then difficult to come back on fundamental design decisions for design time reasons. Indeed, a full evaluation of a hardware design with classical synthesis tools at RT level is very time consuming. Moreover, as for the software part, one can develop the application on a desktop station, but specific problems appear only when integrating code on the real platform, where timing constraints may largely differ and debugging is much more difficult. Thus, in spite of emergent System Level Design Languages (SLDL), it lacks efficient co-design tools to develop and integrate both parts together.

The problem has been amplified with the growing presence of specific RTOS for embedded applications. RTOS are used in particular for dynamical applications, where computations (timing and behavior) mainly depend on the environment. In this context it is necessary to take into account the final system's dynamics in order to simulate and evaluate its unpredictable behavior. Once again it lacks some tools for evaluating (even by simulation) the whole system, including the dedicated OS which controls the custom platform. It is the reason why specific methodologies are absolutely needed to explore OS implementations at different abstraction levels in order to gradually estimate the way they impact the system behavior, the application execution time or the required memory.

In this paper, we propose the use of high level executable and generic models, which permit to explore, validate and refine a real SoC application for both hardware, software and RTOS parts. We adopt SystemC [11] as the support language for joint modeling of both hardware and software, and define a framework allowing exploration and evaluation of various RTOS services managing such complex platforms.

This work falls under the OveRSoC project [2], which consists in developping a methodology for the design and evaluation of reconfigurable system-on-chip (RSoC). In this paper the management of the dynamic reconfiguration aspect is not addressed and constitutes one of the future developments of this work.

The structure of this paper is the following: in the next section, some related works on modeling OS are described. Then we present in Section III. the concepts of our RTOS model, discussing its implementation in SystemC. Its genericity and modularity are also presented. Some experimental results are provided in Section IV. Finally, we conclude and discuss future works.

II. State of the art

A. Modeling languages and tools

The trend in embedded system design enhances an increasing need for high level design languages and verification tools for co-design.

A lot of synchronous languages exists to design real time applications (Esterel, Lustre, Signal or SyncCharts etc.). However they do not include the behavior of the hardware part. That explains why system level design languages have been developed, such as HandelC, SpecC, SystemVerilog, AADL, YML, ImpulseC or SystemC [11]. SLDLs allow to focus on the hardware design at a higher level than the transistor level simultaneously with the embedded software simulation. Finally, SystemC become now the standard langage largely adopted by the community in the joint modeling field.

Based on these languages, a lot of EDA actors and academics now begin to provide tools allowing high level simulation and codesign, like Coware, Synopsis, SPACE codesign [4], Scicos/SyLab or Ptolemy [3], that unfortunately exclude the RTOS part.

B. Methods

To face the increasing complexity of SoC, new design methods appear to ease the design phase and avoid mistakes, detected only during integration, when hardware and software are developed separately.

Le Moigne *et al.* [10] propose the MCSE method for codesign. It allows to refine a design model from a high level view with CPUs to a view with communications details. The MARTES project provides an other methodology inspired from MARTE [14], SysML and UML4SoC. Recently, a trade-off has been made with SLDL languages between simulation speed and accuracy. This is possible thanks to a high level modeling called Transactional Level Modeling (TLM). The SystemC modeling language comes with different levels of abstraction and we try to follow its refinement methodology called TLM 2.0 [11].

C. Existing Real-time system models

Many actors tried to produce frameworks for software/hardware codesign and modeling.

Yu, Gerstlauer and Gajski [15] developed a new language in this goal : SpecC. They propose an RTOS model implemented in SpecC, with a refinement method which allows an easy and automatic implementation of its services.

SoCOS [5] is an other proposal of SLDL framework to model SoC including the RTOS. It was also developed when SystemC was not as evolved as nowadays. This model can handle multiple clocks and has the hability to create processes dynamically. It proposes three computational models : asynchronous, reactive (interruption), and synchronous. It allows an automatic refinement after adding timing data to the code.

In [7], authors describe a SystemC configurable and modular model that can reproduce the behaviour of many OSes like Linux or DSP/BiOS. Their tasks model allows to deal with interruptions if tasks are decomposed in elementary time slots inferior to the nearest interruption event. This works of course only if the interruption arrival time is predictable. They also provide a generic model of driver to wrap any real driver, in order to include the environment (and use input/output data) to the simulation.

Hastono [6] worked on extending SystemC with aspects like dynamic process creation, process control, preemption, process prioritizing. The application is first modelled by a task graph then translated into SystemC TLM by refinning communications and finally tested with different ordering policies. This model also takes into account the Worst-Case Execution Time (WCET) by modeling it as a Gumbel stochastic density.

Posadas *et al.*[13] develop a SystemC 2.0 library to model an RTOS conform with the POSIX Application Programming Interfaces (API), called *PERFidy*. This tool allows to automatically evaluate execution time of any segment of code between two system calls, which is necessary for a timed SystemC simulation.

Madsen [9] presents a SystemC-based framework modeling MPSoC controlled by multiple RTOS. The architecture corresponds to an OS per processor composed of a scheduler and a resource allocator. All RTOS are connected to a synchronizer module, based on a single clock, which allows tasks communications and handles intra and inter processors dependencies. The scheduler is implemented with the SytemC Master-Slave Library in order to attempt multiple messages at a time and is able to run a different kind of ordering policy. A resource allocator handles ressources contention and can change task priority following the Priority Inheritance Protocol to avoid dead-locks.

The approach proposed by Hessel [8] is focused on the design and refinement methodology. Author describes the application in SystemC TLM and refines it to the possible RTL or IP implementation based on libraries. The first step consists in partitioning the application in Sw/Hw pieces. Then software modules are clustered and abstract channels are added to establish communication between clusters. These clusters are assigned to processors. At this time, using the libraries with a manual treatment, first simulation could be done and different scheduling policies could be examined for optimisation. Some power consumption estimations are provided also to help this selection.

Houzet [12] also proposes a design flow intended to automatically refine and generate code for both embedded RTOS and application, in order to avoid manual hardwork and possibles mistakes. It is based on the VCI interface, which allows to wrap IPs for the communication between hardware accelerator, and the MPI-2 (Message Passsing Interface) library. The customised RTOS is based on RTEMS and the model uses the SPIRIT format to describe the architecture.

We can see it exists a lot of OS models. But it lakes some real time behavior modeling completed to a specific model structure (generic and modular) allowing fast exploration by customizing and refining the model.

III. Our RTOS model

This section presents our framework proposal allowing a rapid embedded system evaluation. We explain how to implement a simple RTOS model with SystemC, able to preempt tasks. We then present how to decompose this model into multiple modules in order to ease the exploration of different RTOS architectures.

A. *OS model definitions*

An operating system acts as a system software layer which provides an interface between application programs and the hardware support (including peripherals). Applications are divided into processes or tasks. We do not consider in this work POSIX threads particularity. In hard real-time systems, each task is annotated with time constraints (like deadline or periodicity) depending on the desired application behaviour.

An operating system can also be seen as a services provider that gives processes access to hardware resources and controls their effective and correct sharing out. In our work, we mainly focus on the following basic services:

- Scheduling : The scheduler is responsible for ordering the task execution in the execution units. The scheduling policy it employs will have a deep impact on the whole system's performance. There are many strategies to order application's tasks. These strategies may be based or not on time and process priority, and may be imposed before the execution (static scheduling, decided at compile time) or dynamically calculated, depending on the application evolutions (react to new inputs or interrupts). The last strategy being the most flexible but relies on complex scheduling algorithms and could produce unpredictable behavior.
- Preemption : As there is only one process running on a CPU at a given time, a RTOS must be able to stop the running task at any time and start an Interrupt Request (IRQ) treatment. This ability is called preemption and involves a context switch : load/restore CPU registers and memory contexts, from/to the TCB (Task Control Blocks, which contain all task parameters needed by the OS).
- Process synchronisation : To allow tasks to synchronise together, or to protect from illegal accesses on shared resources (memory or input port), we also consider synchronisation mechanisms such as a basic semaphore service.

The scheduling policy is one of the major factor of performance, but the context switch overhead, the application partition into processes and the memory and buses performances must also be taken into account to evaluate the system behaviour. We then argue that a very early modeling and simulation of an application simultaneously with its custom RTOS could help the evaluation of the future system's performance. This will ease to define, among other things, which scheduling policy best fits some given real time requirements and verify early partitioning choices.

In order to allow exploration of RTOS design choices, we propose a generic RTOS model with the followings constraints:

- Genericity : Our model is generic in the sense that its structure and its behavior allows modelling of most implementation strategies. In this sense, genericity eases services exploration. This as been made possible thanks to a modular and extensible design effort (these properties are developped in section III-D).
- Transparent from user point of view This is posible by a clear separation between the service interfaces (use standard interface like POSIX) and their implementation (heterogeneity).
- Refinement of the model : The model allows multiple levels of implementation of the same service thanks to common API.

Our approach consists in modeling only the behaviour of the software part (OS plus application) with total abstraction of the underlying platform, that is to say a top-down approach, from the OS point of view.

B. *SystemC RTOS model*

The model is based on SystemC, itself based on a C++ class library. SystemC is devoted to hardware simulation and design. It allows to model the concurrent execution of modules's (called `sc_modules`) processes, which execute C++/SystemC code.

The execution time modeled in the SystemC kernel is devoted to concurrent (hardware) modules but not to the time-sharing principle as in classical micro-processors. Indeed, the SystemC kernel carries out a functional process code of a module in a null time and then runs out the simulated time reference according to the value explicitly given by the directive `wait()` into the code. This implies knowing in advance, or to be able to estimate, the execution time and add the directive `wait()` after each portion of code. If not, the whole application runs in zero simulation time. SystemC

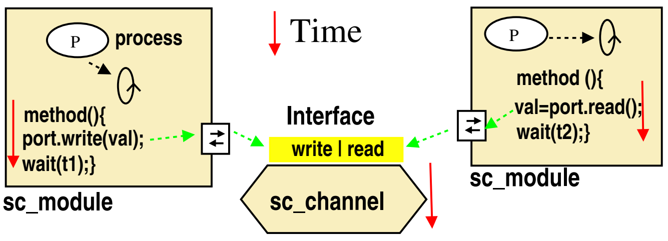

Fig. 1. SystemC simulating block in parallel

also provides mechanisms to synchronize modules execution with events (`sc_event`). Each module can communicate outside through ports (`sc_port`) connected to signals, as simple as wires (`sc_signal`) or more complex as for buses (`sc_channel`). These are in fact modules that provide a particular API and modules can call the corresponding method through their ports, as shown on Figure 1.

As for other design languages, it is possible to have modules composed of sub-modules, allowing different level of abstraction, the lowest being the transistor level. We widely use the abstraction and hierarchical modeling mechanisms offered by this modeling language.

Because the SystemC kernel executes C or C++ code, it allows us to also simulate software parts of a design. To adapt a software application in our SystemC model, each task function is executed in a distinct `sc_module`'s process. We choose to implement the RTOS as a SystemC hierarchical channel, namely a `SC_channel`, which can then provides system services to software tasks through an interface offering the standard system API. In this way, system calls in task code are no more calls to standard `syscall` functions but calls to the corresponding RTOS object methods (Figure 2). Porting

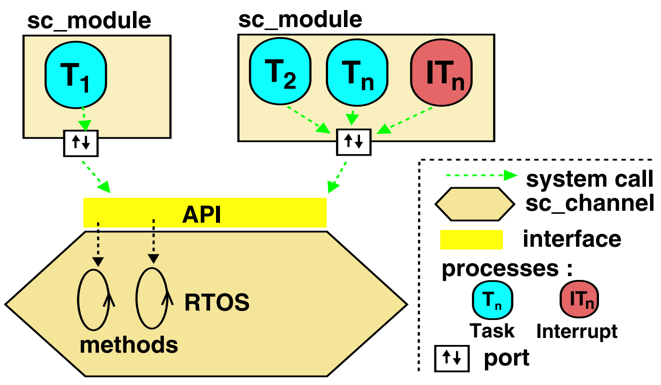

Fig. 2. Task running on a SystemC RTOS

an application onto our model only needs to adapt the syntax of the system call towards the virtual (modeled) RTOS API. This code transformation can easily be done thanks to the use of macroinstructions masking all OS calls.

Our OS model should be able to dynamically create a task, but the SystemC kernel does not allow dynamic creation of modules. We use the SystemC `sc_spawn()` function which allows to create a process after the beginning of the simulation. As the SystemC kernel does not allow dynamic creation of modules, we thus model task by pure C or C++ functions. Each task can call an OS service by calling the corresponding model method. Then dynamically created task are in fact SystemC processes (`sc_thread`) belonging to the RTOS module itself, as repreented in Figure 3.

In order to perform a timed simulation of the software application along with the operating system, this model implies to annotate each code portion between two system calls with `wait()` statements. The execution time depends actually on the given target architecture (memory hierarchy, CPU frequency, number and kind of instruction etc.), but we can easily be satisfied with rapid estimates for our modeling level.

This model allows to take a real application and simulates it without changing any instruction except adding the approximated simulation execution times.

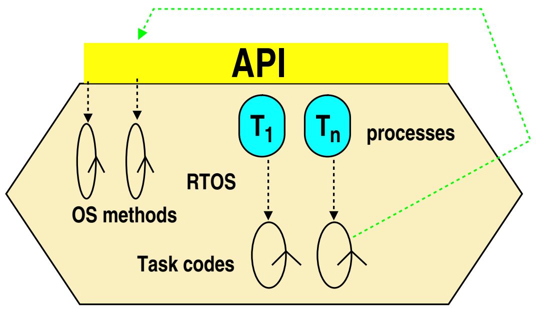

Fig. 3. RTOS model able to create tasks dynamically

C. Time management and scheduling

As discussed previously, it is necessary to introduce in the RTOS model a mechanism making it possible to circumvent artificially the execution engine of SystemC so that only one task occupies the execution unit at a given time. To cope with this, we encapsulate the `wait()` statements by a (pseudo) system call towards our model (the `os_wait()` call) which will manage this suitable behaviour. Each task is associated with an internal data structure describing its characteristics and OS parameters (as in a real OS). A specific internal SystemC event is included in each task's TCB. This event is used by the scheduling algorithm for synchronising tasks executions. Each task in our model is a SystemC thread having this event in its sensitivity list. In this manner, we can explicitly force nonparallelism, by making all tasks wait for their own event.

For modeling task preemption, OS listens for a periodic tick timer, or an interrupt event, which starts the OS scheduling algorithm. This scheduler can thus take the decision to awake the adequate process: it triggers the corresponding task event and then, the task starts. This task runs its functionnal code and calls `os_wait()` to simulate its execution time. The `os_wait()` waits for both the time given as argument and for a specific event called `stop`. If the `os_wait()` function loop is interrupted, then it calculates the remainning time and wait again, but now indefinitelly for the task event. When the scheduler decide to await a task, it trigger the coresponding task event, as for launching any task; and then the task can continue its `os_wait()` execution, for the remaining duration. Thus, we are able to accurately model the sequential execution of tasks and possible preemption at any time as shown in Figure 4.

An obvious limitation of task execution accuracy in our model consists in the fact that even if preemption could occur at any time, task blocks are already executed (in zero time) before the `os_wait()`. As long as a timed functional validation is concerned, this limitation has no consequences on the simulation results: the global application behaviour can be verified and potential deadlocks identified. If a more precise simulation is expected, it is possible to reduce the task block granularity until an instruction accurate model of the processor (Instruction Set Simulator), but is out of the scope of our work.

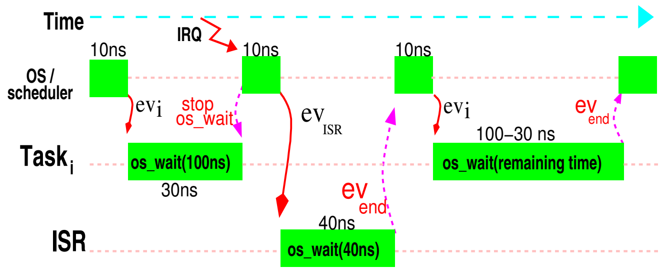

Fig. 4. A task runs its functionnal code in 0 time, and then performs an OS_wait() with a specific duration (100ns in this exemple). The preemption applies in fact on the `os_wait()` method, allowing an internal computation of the preemption duration (caused here by an interruption -ISR-).

D. Modularity and genericity

In order to build a generic model of a RTOS, the main services have been grouped into several categories. Locality of shared internal data has been the most important criterion for defining these categories. The actual services categories in the model presented in this paper are the following:

1) Process management and CPU time share (scheduling) : task create, suspend, ...
2) Synchronous and asynchronous communications: by ports, shared memory or FIFO
3) Synchronization : semaphore/Mutex, conditional variables and process joining
4) Memory management : allocate / free
5) Storage management : file read / write
6) Interruptions management
7) Distributed OS management
8) Timing : explicit simulation time wait statement

The last group (timing) has been added only for simulation and execution time modeling purpose and do not represents any real OS service.

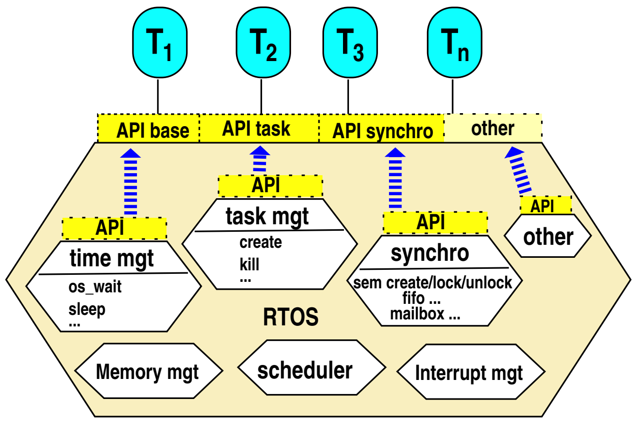

Fig. 5. Modular OS model: The RTOS is modeled by a hierarchical `sc_channel` containing several sub-channels which represent and model a service group. The main channel provides all the services to the application through a single API (the interface of the OS channel).

In order to offer these services to the application's tasks and render their implementation relatively independent one from the other, we choose to implement each group of service in the form of a hierarchical SystemC sub-channel (see Figure 5). Sub-channel inherits from the `sc_interface` class and thus offers the services defined in its interface. To offer the whole API, the RTOS exports each sub-channel interface, by using the `sc_export` mechanism.

The interface has been defined in a way that it is independent from the particular implementation of the services. This clear separation between the services interface (i.e. the definition of the system calls) and their implementations is an important feature of our model. It guarantees and eases the exploration of the services implementations by replacing a block by an other one assuming it implements the same interface.

The main drawback of decomposing the RTOS functions into separate blocks consists in insuring correct interoperability between separate SystemC objects. For example, the semaphore manager module obviously needs to communicate with the scheduling module in order to inform which task has to be put in sleeping state or has to be awaken. To solve this, we have implemented a second distinct interface in each service block. It consists in an internal interface through which other service groups can ask for services. The same mechanism (`sc_interface` inheritance) is used for the internal services. The `re-schedule()` method is a good example of an internal service offered by the scheduling block to other services. Each service block is thus added with inter-module communication ports and inter-module communications are realised by point-to-point connections (see Figure 6). We will

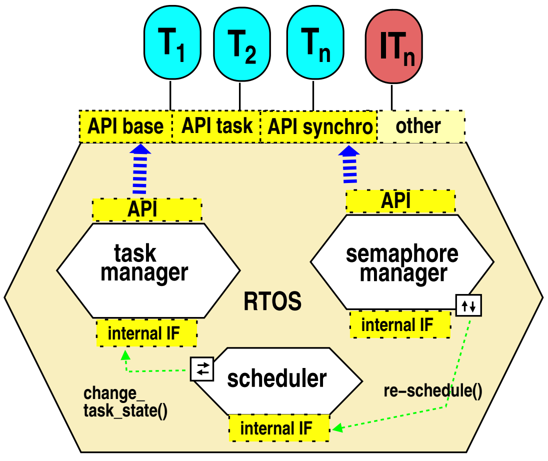

Fig. 6. Internal interface of inter module services

show in the next section that this modular structure of the RTOS model really eases both customisation and exploration of several implementations of basic or dedicated services. Several implementations of the same group of services can be explored and validated by replacing a service block by an other compliant with its two (internal and external) interfaces. One of the major feature that could be explored in this way is the hardware/software partitioning of an embedded RTOS.

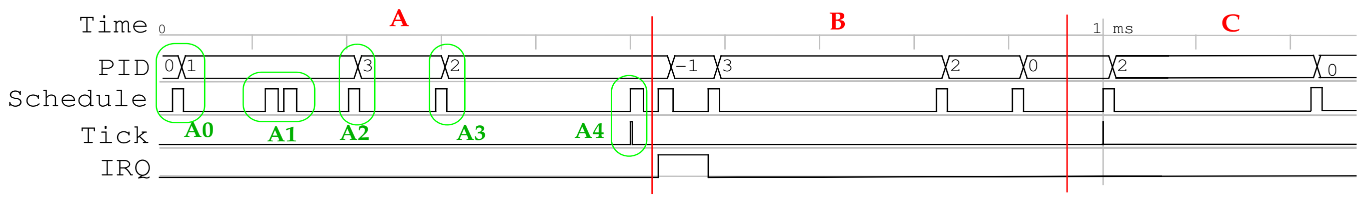

Fig. 8. Execution trace with current running PID and the scheduler activity

IV. Experiments

A. *The RTOS model implementation*

We have implemented a model of RTOS inspired from μC-OS-II[1]. As described in the previous section, services of the OS are encapsulated into SystemC modules. As each module is associated to a part of the system API, system calls toward modules follow those of μC-OS-II. In μC-OS-II, the scheduler is only based on user defined tasks priorities without round-robin since each task is associated a single priority. Inversely, our model can accept a large choice of scheduling strategies. The model obviously contains a real-time clock that allows the modeling and the simulation of fundamental RTOS behaviours such as periodic tasks, sleeping tasks and OS tick sensitivity. The tick period of the clock has been set in the following experiments to 500 μs (Figure 8). The OS tick periodically awakes the scheduler to simulate time quantums. We applied

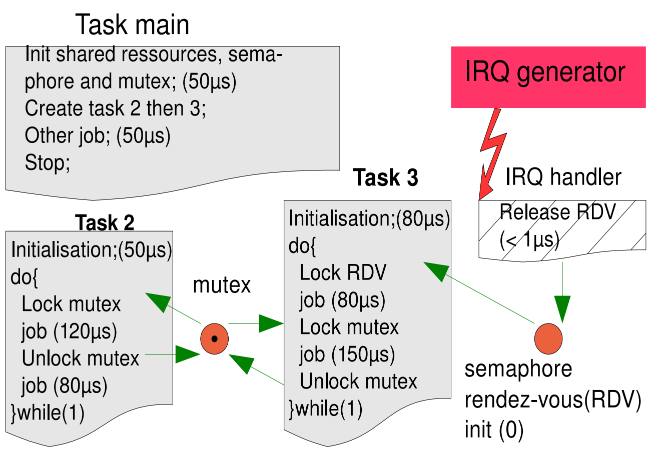

Fig. 7. Test application

our method to a small synthetic application (see Figure 7) to demonstrate the correct behaviour of this RTOS model.

This artificial application is composed of three tasks and an interruption handler. The main task only executes once with the highest priority, initialises global resources like a mutex and a ”Rendez-Vous” , launches task 2 and task 3 and does a little job after. The mutex (semaphore initialized to 1) protects a shared resource (like access to a video picture on memory) between task 2 and task 3. The rendez-vous (semaphore initialized to 0) semaphore is unlocked by the interruption handler when an interruption occurs. As shown in Figure 7, the OS model contains an interrupt handler able to be executed each time the corresponding interrupt is raised (virtually triggered by the simulation engine). Interrupts are modeled as particular processes with the highest task priority, considering the priority is not really relevant for interruptions. Task 2 has a period of 1 ms and has a lower priority than task 3. It periodically runs the same job, which consists in taking the mutex, doing some computation on the shared data, releasing the mutex, doing computation, and then sleeping until next period (figure 7). The third task runs each time an interrupt occurs except for the initialization phase where some initial treatments are done (initialising variables). It thus always waits for the rendez-vous semaphore (blocking call) in a loop, then it takes the mutex to perform a job with the shared resource, releases the mutex, and waits again for any new interrupt (via the rendez-vous semaphore).

B. *RTOS behaviour simulation*

We illustrate with Figure 8, the simulated RTOS activity. This Gantt diagram shows four fundamental RTOS mechanisms : tasks preemption, OS tick sensitivity, interruptions and tasks synchronization.
The scheduling obviously begins with the task PID (Process IDentifier) 0, which is the RTOS initialisation (and after, the idle task). As we can see in area A, when the OS finishes its initialization, it launches the scheduler (in A0) which elects the main task (PID 1 because it is the first task being created). We can see that the scheduler is launched each time a task is created (A1) or when a task ends (A2) or is blocked(A3). In A4, the OS Tick appears and relaunches the scheduler, and continues to run the only one ready task 2. In area B, an interrupt occurs, which launches the scheduler and runs the corresponding IRQ handler (PID -1). This one releases the RDV semaphore and stops. The scheduler then elects task 3 now ready, and not the interrupted task 2, because task 3 have a higher priority. Then when task 3 ends, task 2 can continue its job. In area C, we only focus her on the fact that the timer awakes correctly task 2 each millisecond as it is a periodic task. The area C will be more described in the following.

C. *RTOS services exploration*

Thanks to the modular nature of our OS kernel, by replacing a module by an other, we are able to explore different kind of implementation strategies for a given service. Figure 9 focuses on two different strategies for managing semaphores: awaking (Figure 9.B) or not (9.A) the scheduler upon semaphore release. This simple change can lead to different behaviours as now explained on the same application, where task 2 already owns the mutex when an interrupt occurs. The precedent

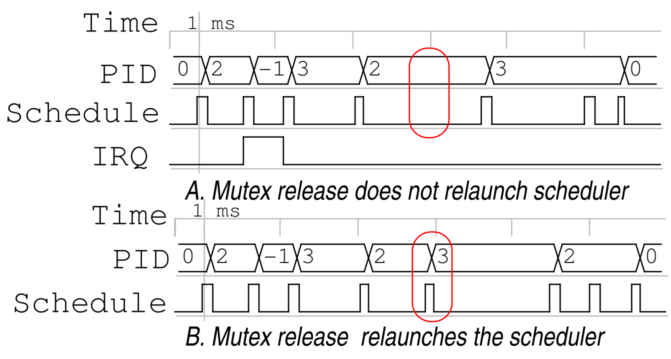
Time
1 ms
PID
0
2
-1
3
2
3
0
Schedule
IRQ
A. Mutex release does not relaunch scheduler
Time
1 ms
PID
0
2
-1
3
2
3
2
0
Schedule
B. Mutex release relaunches the scheduler

Fig. 9. New semaphore implementation : when released it restarts the scheduler

experiment implementation (Fig. 9.A) shows that task 3 runs partially and have to wait until the task 2 releases the mutex and finishes (except if the scheduler tick occurs) to treat the interrupt. This execution does not correspond to interruption subtreatments in real-time systems. Inversely, in the new experiment (Figure 9.B) we implement the mutex unlock so as to awake the scheduler which then searches the best task to run. Thus task 3 runs as soon as task 2 releases the mutex (surrounded area).

This example illustrates the exploration possibilities afforded by the structure of the RTOS model.

D. Current work and simulation costs

In practice, all these mechanisms are currently used on a more realistic application in the field of image processing for robotic vision. The application is used to learn object views or landscapes and extracts local visual features from the neighborhood. In this context we profiled this application on a hardware architecture composed of multiple Nios-II processors (MPSoC) prototyped onto an Altera Cyclone-II FPGA circuit. The timing data measured from this implementation are used to back-annotate the high-level model in order to simulate and explore different implementation strategies. The application runs without any change either on our abstract RTOS model or on μC-OS-II on a real embedded platform (except the necessary timing annotations).

TABLE I
SIMULATION COSTS

platform	simulation time	overcost in %
C under Linux	2m45s	0
only SystemC	2m53s	4.5
RTOS in SystemC	3m12s	16.4

We used a Intel Dualcore at 1.66GHz under Linux to perform simulation tests, on a sequence of 1000 images. The simulation does only use one core because SystemC does not work on multiprocessor. The application is fragmented into 31 tasks. As shown on Table I, we can conclude that our RTOS model does not dramatically impact the simulation time (only 9.8% more than the application running with the SystemC kernel primitives without our RTOS model) and is still faster than a real implementation execution (about 32 seconds to treat one image on the NIOS platform with μC-OS II). It allows to rapidly explore the application partitioning and the RTOS behaviour, whitout needing a real expensive platform.

V. Conclusion

We have presented in this paper the basic concepts of an abstract RTOS model in SystemC for system level design. The high level description of the model allows the designer to early simulate the dynamic behaviour of complex real-time applications. The model can be parameterized in order to represent the exact timing of the OS services on the future architecture. The designer can thus evaluate the impact of different service implementations on the real-time performances of the system. Moreover this model is built in an object oriented manner and each service of the OS can be added or deleted according to the application/architecture needs, updating automatically the operating system API. This modular structure will also permit to explore the deployment of the OS services onto multiple processing elements (processors and hardware reconfigurable units). In this direction, we are currently working on inter-OS communication mechanisms based on TLM 2.0 allowing the exploration of distributed OS implementations.

References

[1] *MicroC/OS-II: the real-time kernel.* USA: CMP Media, Inc., 2002, available at : http://www.ucos-ii.com.

[2] I. Benkermi, A. Benkhelifa, D. Chillet, S. Pillement, J. Prevotet, and F. Verdier, "System-Level Modelling for Reconfigurable SoCs," in *Design of Circuits and Integrated Systems (DCIS'05)*, Nov. 2005.

[3] J. Buck, S. Ha, E. A. Lee, and D. G. Messerschmitt, "Ptolemy: A Framework for Simulating and Prototyping Heterogenous Systems," *Int. Journal in Computer Simulation*, vol. 4, no. 2, pp. 155–182, Apr. 1994.

[4] J. Chevalier, O. Benny, M. Rondonneau, G. Bois, E. M. Aboulhamid, and F.-R. Boyer, "SPACE: a hardware/software SystemC modeling platform including an RTOS," in *Forum on Design Languages(FDL'03)*, Frankfurt, Germany, 2003.

[5] D. Desmet, D. Verkest, and H. D. Man, "Operating System based software generation for Systems-on-Chip," in *Conference on Design Automation(DAC'00)*, 2000, pp. 396–401.

[6] P. Hastono, S. Klaus, and S. Huss, "Real-Time Operating System Services for Realistic SystemC Simulation Models of Embedded Systems," in *Forum on specification and Design Languages(FDL'04)*, Sep. 2004.

[7] Z. He, A. Mok, and C. Peng, "Timed RTOS Modeling for Embedded System Design," in *IEEE Real Time on Embedded Technology and Applications Symposium (RTAS'05)*, 2005, pp. 448–457.

[8] F. Hessel, V. M. D. Rosa, C. E. Reif, C. Marcon, and T. G. S. D. Santos, "Scheduling Refinement in Abstract RTOS Models," *ACM Transactions on Embedded Computing Systems*, vol. 5, no. 2, pp. 342–354, 2006.

[9] J. Madsen, K. Virk, and M. Gonzalez, "Abstract RTOS Modeling for Multiprocessor System-on-Chip," in *Symposium on System-on-Chip (SOC'03)*, Nov. 2003, pp. 147–150.

[10] R. L. Moigne, O. Pasquier, and J.-P. Calvez, "A Generic RTOS Model for Real-time Systems Simulation with SystemC," in *Conference on Design, automation and test in Europe (DATE'04)*, 2004, p. 30082.

[11] OSCI, "IEEE 1666^{TM} Standard SystemC Language," available at : http://www.systemc.org.

[12] S. Ouadjaout and D. Houzet, "Generation of Embedded Hardware/Software from SystemC," *EURASIP Journal on Embedded Systems*, no. ID 18526, p. 11, 2006.

[13] H. Posadas, J. Adamez, E. Villar, F. Blasco, and F. Escuder, “RTOS modeling in SystemC for real-time embedded SW simulation: A POSIX model,” *Design Automation for Embedded Systems*, vol. 10, no. 4, pp. 209–227, Dec. 2005.

[14] L. Rioux, T. Saunier, S. Gerard, A. Radermacher, R. D. Simone, T. Gauthier, Y. Sorel, J. Forget, J.-L. Dekeyser, A. Cuccuru, C. Dumoulin, and C. Andrè, “MARTE: A New OMG Profile RFP for the Modeling and Analysis of Real-Time Embedded Systems,” Jun. 2005.

[15] H. Yu, A. Gerstlauer, and D. Gajski, “RTOS Scheduling in Transaction Level Models,” in *IEEE/ACM/IFIP int. conf. on Hardware/software codesign and system synthesis (CODES+ISSS’03)*, 2003, pp. 31–36.