A Smooth Refinement Flow for Co-designing HW and SW Threads [*]

Paolo Destro Franco Fummi Graziano Pravadelli
Dipartimento di Informatica - Università di Verona
Strada le Grazie 15, 37134 Verona, Italy
{destro, fummi, pravadelli}@sci.univr.it

Abstract

Separation of HW and SW design flows represents a critical aspect in the development of embedded systems. Co-verification becomes necessary, thus implying the development of complex co-simulation strategies. This paper presents a refinement flow that delays as much as possible the separation between HW and SW concurrent entities (threads), allowing their differentiation, but preserving an homogeneous simulation environment. The approach relies on SystemC as the unique reference language. However, SystemC threads, corresponding to the SW application, are simulated outside the control of the SystemC simulation kernel to exploit the typical features of multi-threading real-time operating systems running on embedded systems. On the contrary HW threads maintain the original simulation semantics of SystemC. This allows designers to effectively tune the SW application before HW/SW partitioning, leaving to an automatic procedure the SW generation, thus avoiding error-prone and time-consuming manual conversions.

1. Introduction

The always increasing complexity of embedded systems have induced designers to join HW and SW design phases by defining efficient methodologies relying on the concept of HW/SW co-design [1, 2, 3, 4, 5, 6, 7]. They propose techniques for simultaneous consideration of HW and SW parts within the design process. Co-design flows start generally at system level, where the embedded system is described by a set of untimed functional processes communicating by means of high-level transactions. Then, after functional validation, HW/SW partitioning takes place and HW/SW co-simulation strategies are adopted to allow the co-verification of the partitioned system [4, 6, 8, 9, 10]. Early co-simulation approaches require to set up complex heterogenous environments where HW and SW parts are executed by using different simulators [8, 9]. This heterogeneous style is sub-optimal in terms of simulation performance and easiness of integration, but it was the only possible choice when VHDL or Verilog simulation was the highest possible level of abstraction for HW simulation.

On the contrary, homogenous environments, like the ones proposed in [1, 2, 11], use a single engine for the simulation of both HW and SW components, thus simplifying the design modeling. The Ptolemy [1], Polis [2] and Statemate [11] environments are pioneering works in that direction. However, they are suitable only in a very initial phase of the design, since they are based on formal models of computation that do not provide designers with support for easily moving towards the commercial RTL and gate-level synthesis tools generally adopted in the HW refinement flow.

To overcome such a limitation, more recent approaches use system level description languages (SLDLs) such as SpecC [12] or SystemC [13], that can be adopted throughout the refinement steps from system level to RTL. In particular, the advent of design flows based on SystemC allowed the definition of efficient semi-homogeneous approaches [4, 10, 14, 15]. They are homogeneous from the language point of view, since both HW and SW are described by using C++, but still heterogenous from the simulation point of view, since HW and SW are generally executed by using different simulators: the SystemC simulation kernel for the HW components and an instruction set simulator (ISS), or a more/less detailed CPU model able to execute SW, for the SW programs. On the contrary, the adoption of SystemC as the reference language for modeling both HW and SW components definitely simplifies the implementation of the initial system-level model as well as the subsequent HW/SW partitioning. However, although the latest versions of SystemC (SystemC2.x) include some system-level oriented constructs (e.g., communication channels or process synchronization primitives), SystemC is still based on typical hardware design requirements like signals, clocks, registers, parallel synchronous and asynchronous processes (i.e., concepts known from hardware description languages like VHDL and Verilog). Thus, SystemC offers little support for the embedded system designer, who wants to include the dynamic real-time behavior, typical of embedded SW, in the system model. In particular, while HW refinement with SystemC is supported by commercial multi-level design tools [16], SystemC lacks the necessary constructs to provide SW refinement. Besides, SystemC does not support thread priority assignment, because its simulator does not offer all the necessary functions usually found in a real-time operating system (RTOS), such as preemption or priority scheduling. This represents a strong limitation for SystemC-based co-design approaches, considered that embedded software now routinely accounts for 80% of embedded system development costs [17].

Some works [3, 18, 19, 20, 21] have been proposed to overcome the previous limitation by integrating RTOS capability in the C++/SystemC design flow, to improve the support to SW refinement. In [3] the authors present SoCOS, a system-level design environment for modeling and simulating embedded system. In particular, a C++ library is proposed that offers the designers with services analogous to an operating system (OS) in SW design. In this way, real-time aspects can be gradually introduced and SW can be functionally tested in combination with HW. However, this approach requires to use a proprietary simulation engine and it needs manual refinement to get the SW code. In [18] a method is proposed for automatic generation of application-specific OSs and automatic targeting of application SW. The OS generation starts from a very small OS kernel. Then, OS services, which are specific to the application and deduced from dependencies created by the system specification, are added to the kernel to construct the whole OS. However, no methods for embedded SW code generation are described. In [19] the authors present a method for systematic embedded SW generation that reduces the SW generation cost in a platform-based HW/SW co-design methodology. In particular, C++ SW code is automatically generated from SystemC processes. Such an approach relies on the overloading of a subset of SystemC constructs. However, it imposes code modification when SystemC construct not supported are used in the original description. Automatic embedded SW generation is addressed also in [20], where a

[*]This work has been partially supported by the European project VERTIGO FP6-2005-IST-5-033709.

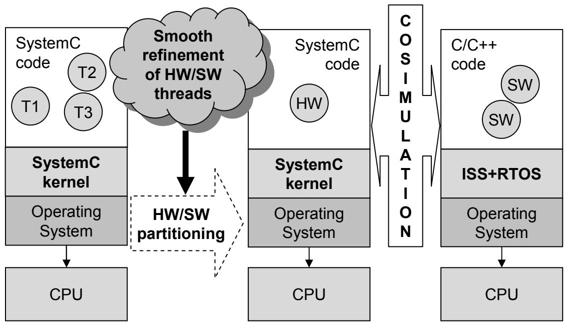


Figure 1: Position of the proposed refinement flow in a traditional SystemC-based HW/SW co-design approach.

method is proposed to refine an SLDL specification into C code, after HW/SW partitioning.

The previous works provide valuable methods for supporting SW refinement, but they must be applied after HW/SW partitioning has been decided. This prevents an efficient evaluation of different HW/SW configurations. To facilitate HW/SW partitioning, a different approach is presented in [21]. The authors propose a SystemC refinement methodology that focuses on using SW abstraction levels to enhance high-level embedded SW modeling support. This is achieved by encapsulating RTOS functionalities in a SystemC-RTOS interface. Besides, the methodology allows to easily move threads from HW to SW and vice versa without affecting intermodule communications. However, the high flexibility of such an approach is paid in term of simulation time overhead, since the simulation infrastructure is quite complex. In fact, to allow the RTOS scheduling of SW threads SystemC-specific function calls are translated into appropriate RTOS-specific function calls by an opportune interface and an ISS is encapsulated in SystemC.

In this paper, we propose a simpler SystemC-based co-design approach. Early system-level evaluation of different HW/SW configurations can be carried out in a homogeneous SystemC environment, without the need of introducing complex co-simulation frameworks relying on different simulators for HW and SW threads. Besides, it provides designers with support for embedded SW generation and tuning of SW threads by exploiting multi-threading RTOS features. This is obtained by linking SystemC modules corresponding to SW threads to a C++ library that overrides SystemC constructs to remove the SystemC simulation kernel, and let the SW threads to be handled by the underlying RTOS. In this way, HW threads maintain the original simulation semantics, while SW threads are simulated outside the control of the SystemC simulation kernel. This allows to refine and validate the SW threads synchronization without time-consuming and manual conversion from SystemC to C++ code.

It is worth noting that the proposed approach is not intended for performance evaluation, which is, indeed, the aim of co-simulation strategies that operate on detailed RTL HW descriptions and SW applications running on instruction set simulators or real boards. On the contrary, our refinement flow is intended to be used at system level before co-simulation takes place, and its main goals are:

- simplifying HW/SW partitioning without the need of changing the code of HW/SW modules;
- simplifying the embedded SW generation and tuning such that the obtained code could be as close as possible to the one that will run on the final embedded system.

Figure 1 shows the position of the proposed refinement flow in a traditional SystemC-based HW/SW co-design approach.

The paper is organized as follows. Section 2 summarizes the SystemC execution model highlighting why the SystemC simulation kernel is not suited for SW refinement. Section 3 describes the proposed refinement flow. Section 4 presents how SystemC constructs have been overloaded to replace the SystemC simulation kernel and allow the support of RTOS multi-threading features in the proposed refinement flow. Section 5 describes how the SystemC semantics of HW modules have been preserved after the removal of the SystemC simulation kernel. Section 6 explains how the proposed SystemC-RTOS mapping allows the automatic generation of embedded SW and its tuning. Section 7 presents experimental results. Finally, Section 8 is devoted to concluding remarks.

2. SystemC Execution Model

The SystemC library allows us to model the behavior of reactive systems by defining synchronous and asynchronous processes. A synchronous process is executed only at specific instances of time determined by the clock edge to which the process is sensitive. On the contrary, an asynchronous process is sensitive to generic events (statically declared in a sensitivity list or dynamically generated during the simulation) and its execution is resumed each time one of such events occurs. Independently from these observations, there are two kinds of processes in SystemC: *methods* and *threads* declared by using, respectively, the keyword `SC_METHOD` and `SC_THREAD`[1].

A method is triggered when the SystemC simulation kernel calls the function associated with the process instance according to its sensitivity list. When a method process is triggered, the associated function executes from beginning to end, then it returns control to the kernel. A method process cannot be terminated and cannot include calls to `wait()` functions for suspending its execution. On the contrary, a function associated with a thread process is called only once by the kernel (except when a clocked thread process is reset, in which case the associated function may be called again). A thread can call a `wait()` function for suspending its execution until the kernel resumes the thread, according to its sensitivity list.

The SystemC simulation kernel relies on a *co-routine execution model*, also known as co-operative multitasking[2]. Process instances (either methods or threads) execute without interruption, only a single process can be running at any one time, and no other process can execute until the currently executing process has yielded control to the kernel. Thus, a process cannot pre-empt or interrupt the execution of another process as an OS might. Because transfer of control between processes only happens when `SC_THREAD` processes call `wait()`, or, equivalently, when `SC_METHOD` processes return control to the simulator, SystemC models can be written without concern that a process may be pre-empted involuntarily. Specifically, the code within a method or delimited by two `wait()` statements in a thread can safely assume that no other processes have modified any variables which are also accessible to other threads.

Such a co-routine execution model is suited for the simulation of concurrent HW modules, but it is not for SW refinement and synchronization. Thus, using SystemC for describing mixed HW/SW embedded systems is profitable only at system level, when HW and SW parts are not defined yet. After HW/SW partitioning, SystemC should not be used for SW functionalities anymore. In fact, the SystemC co-routine execution model prevents, a priori, race conditions and synchronization problems. On the contrary, these may happen when the SW processes are directly executed on a multi-threading RTOS without the SystemC kernel. Moreover, the behavior of the

[1]`SC_CTHREAD`s can also be declared, but the only difference between them and `SC_THREAD`s is that an `SC_CTHREAD` is sensitive on a clock whereas an `SC_THREAD` process can be sensitive on any event.

[2]This is true for both the Quick Threads and the Posix Threads libraries supported by SystemC.

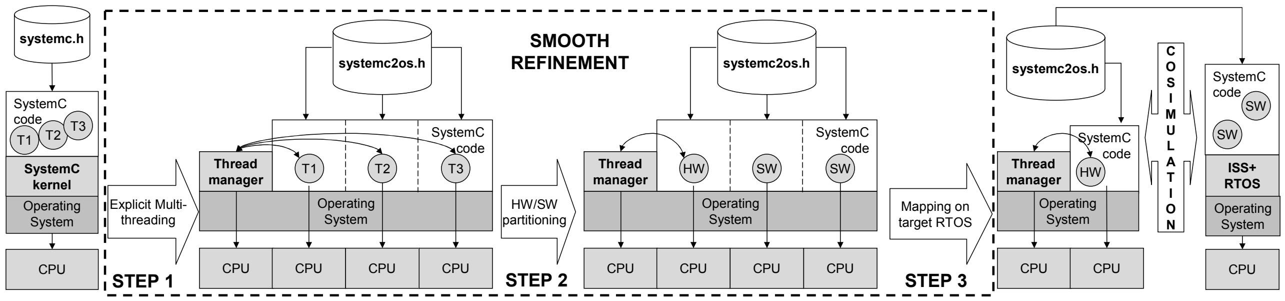


Figure 2: Proposed refinement flow for co-designing HW and SW threads.

RTOS priority-based pre-emption cannot be evaluated. Thus, by using SystemC, designers cannot evaluate neither the correctness of the synchronization mechanism, that they must introduce in the SW functionalities, nor the RTOS impact on the whole system. Besides, SystemC simulation cannot takes benefits from running on a multi-threading RTOS, possibly hosted by a multi-processor system, since SystemC processes are executed sequentially on a single processor. Thus, to use SystemC code for efficiently modeling and evaluating embedded SW, avoiding the previous drawbacks, SW threads must be executed out of the control of the SystemC simulation kernel as described in the next sections.

3. HW/SW Refinement Flow

Figure 1 shows a traditional SystemC-based HW/SW co-design flow. At system level, the system functionality is represented by a set of concurrent threads managed by the SystemC simulation kernel without distinction between HW and SW threads. Then, after the HW/SW partitioning, the SystemC code describing SW functionalities, is mapped in a pure SW program (e.g., C/C++) scheduled by the selected RTOS which runs on an ISS of the target processor. In fact, as reported in the previous section, SW refinement and synchronization evaluation cannot be profitable performed under the control of the co-routine execution model of the SystemC kernel. On the contrary, SystemC modules representing HW components are refined in synthesizable RTL descriptions and keep to be simulated by exploiting the SystemC kernel, thus proceeding along the traditional HW design flow. However, after partitioning, the SW and the HW flows cannot be totally disjoint, since embedded systems are target-specific systems where the HW/SW synergy is extremely strong. Thus, a co-simulation framework is generally used to allows a profitable cooperation between SW engineers and HW designers during the subsequent refinement steps and performance analysis. However, the introduction of co-simulation makes difficult changing the HW/SW partitioning if it results to be inadequate. In fact, moving threads from SW to HW may require heavy code modifications of the HW/SW interface, and manual conversion from C++ processes, that possibly call RTOS functions, to SystemC threads, or vice versa.

Thus, we propose the co-design approach shown in Figure 2 to provide designers with a smooth HW/SW refinement flow that aids an efficient exploration of different HW/SW configurations and allows to tune the SW synchronization mechanism exploiting the multi-threading feature of the underlying OS. The starting point is represented by a set of SystemC concurrent threads representing the system-level description of the design. The first step of the refinement flow is completely automatic. It consists of replacing the SystemC library (`systemc.h`) with a C++ library (`systemc2os.h`) that we have developed to override the SystemC constructs and exclude the SystemC kernel from the simulation environment. The `systemc2os.h` library exploits only C standard functions and system calls for synchronization provided by multi-threading RTOS. In this way, the SystemC code modeling the unpartitioned system-level design is unchanged, but SystemC processes become threads of the underlying OS. The simulation is driven by a simple thread manager that preserves the behavior of threads according to the SystemC semantics.

The first benefit deriving from substituting `systemc.h` with `systemc2os.h` is that the co-routine execution model of the SystemC simulation kernel is removed. Thus, threads can be concurrently executed, possibly on a multi-processor system. Concurrent execution is a mandatory condition for addressing SW refinement issues like mutual exclusion, deadlock avoidance, priority assignment for thread scheduling, etc. Moreover, it is worth noting that the removal of the co-routine execution model may reflect into a simulation speed-up, with respect to the original SystemC simulation. This is particularly evident when threads execute CPU-bound computations (e.g., for modeling multimedia systems that require complex signal processing). In fact, in this case, the OS scheduler can distribute the load on different CPUs, and the preemption mechanism guarantees that ready threads do not have to wait the resume of stopped threads (that happens in sequential execution). However, the simulation of systems composed of many I/O bound threads with little computation can be slowed down. This is due to the presence of the thread manger which exploits OS system calls causing many time-consuming switches between user and kernel execution modes. Thus, the overhead introduced by the thread manager is predominant with respect to the little computation load of running threads. Such considerations will be further examined in the experimental result section.

At step 2, HW/SW partitioning takes place. Different HW/SW configurations can be effortlessly evaluated. In fact, moving a thread from HW to SW (or vice versa) requires only to disconnect (connect) the thread from (to) the thread manager. Such an operation consists of modifying just one line of code in the thread declaration. Thus, HW threads keep to be controlled by the thread manager according to the SystemC semantics, while SW threads are handled by the underlying multi-threading OS. Thus, SystemC code corresponding to SW threads is pure C++ code, indeed. Then, time-consuming and error-prone manual conversions from SystemC module to C++ classes are completely avoided. At this point, the majority of SW refinement steps can be efficiently performed without the need of complex co-simulation strategies. In particular the synchronization mechanism can be tuned and evaluated, since SW threads are not executed in co-routine mode.

Once HW/SW partitioning is satisfactory, co-simulation can be introduced at step 3. Thus, HW threads keep to be simulated under the control of the thread manager, while SW threads can be mapped on the target ISS+RTOS environment. This allows SW engineers to complete SW refinement and tune the RTOS-dependent features. Moreover, drivers, implementing an ad-hoc HW/SW interface, can be introduced. After step 3, HW threads can be refined in an RTL description for the subsequent synthesis steps, while SW can be optimized according to traditional SW engineering flows.

SystemC	systemc2os.h
sc_main()	main()
SC_MODULE	Struct
SC_METHOD and SC_THREAD	OS thread, semaphore, conditional variable, mutex
sc_clock	OS thread
sc_port and sc_signal	Shared memory, mutex
wait()	Semaphore
Data types	Original SystemC data types

Figure 3: Mapping of SystemC constructs.

```
SC_MODULE(transmit){
  sc_in<sc_int<32> >  in_port;
  sc_out<sc_int<32> > out port;
  void run();
  SC_CTOR(transmit){
    SC_THREAD(run);
    sensitive << in_port;
  }
}
```

Figure 4: A simple `SC_MODULE`.

4. SystemC to OS Mapping

The mapping of SystemC constructs into C++ OS-dependent constructs has required to find a way for converting both syntax (SystemC keywords), and semantic aspects (process execution, signal updating, synchronization, communication, etc.). The syntax conversion has been performed by using the preprocessor directive `#define`, while the semantic issues have been addressed by defining new functions generally relying on primitives provided by multi-threading OSs. The resulting `systemc2os.h` library is independent from the underlying OS, since it uses symbolic name for defining generic functions that can be specialized by opportunely mapping them on OS-dependent functions through the preprocessor directive `#define`. The library includes:

- a set of preprocessor directive to re-define the SystemC keywords (e.g., `SC_MODULE`, `sc_signal`, etc.);
- a set of functions used by the thread manager to implement a synchronization and scheduling mechanism that preserves the SystemC semantics of HW threads and provides synchronization functions for the SW threads.

Figure 3 shows how the main SystemC constructs have been replaced in the `systemc2os.h` library.

4.1 Modules

The SystemC construct `SC_MODULE` (Figure 4) is used to define the interface of a module and the behavior of its processes. It declares input and output ports, internal signals, and methods. Moreover, it includes the declaration of the constructor (`SC_CTOR`) that defines which methods behave as processes (`SC_METHOD` or `SC_THREAD`) and the corresponding sensitivity lists. The SystemC keyword `SC_MODULE` is straightforward substituted by the C++ keyword `struct`. On the contrary, the replacement of the constructor is not simply syntactical, but it has required the definition of three objects, `sensitive`, `sensitive_pos` and `sensitive_neg` to manage the three kinds of sensitivity list that can be declared in SystemC. They implement the operator `<<`, typically used to define the sensitivity list of a process, as described in Section 4.5.

4.2 Processes

Processes, that in SystemC are executed sequentially, have been mapped into threads of the OS. Thus, they can be concurrently executed. The macro implemented to redefine `SC_THREAD`s and `SC_METHODS`s performs the following operations:

*/*The function waits for a broadcast signal from the thread manager to start the simulation by using a conditional variable. Then the body of the method is called inside the loop, and it is suspended each time the `wait()` function is reached to respect the SystemC semantics.*/*

```
template<class M>
void *sc_method(void *p){
  thread_info<M> *my_info;
  my_info = (thread_info<M> *)p;
  pthread_mutex_lock(&mutex_sched)
  pthread_cond_wait(&cond_sched, &mutex_sched);
  pthread_mutex_unlock(&mutex_sched);
  while(true){
    ((my_info->i_addr)->*(my_info->i_offset))();
    wait();
  }
  return 0;
}
```

Figure 5: Startup routine for the redefinition of the `SC_METHOD` semantics.

- creation of an OS-dependent thread where the process body will be executed;
- addition of the thread handler to the list used by the thread manager to implement the desired synchronization;
- instantiation of a semaphore associated to the thread for stopping and restarting its execution, respectively, when a `wait()` is executed inside the process, and when a signal included in the sensitivity list of the process changes its value;
- initialization of a structure for the management of the sensitivity list of the process.

The startup routine of the created thread waits for a broadcast signal from the thread manager before starting the execution of the process body. In this way, the execution of processes start concurrently when all the corresponding threads have been instantiated. Moreover, to replace an `SC_METHOD` respecting its semantics, the startup routine executes the body of the method inside an always-true loop as shown in Figure 5. Note that, the `wait()` function has been redefined as reported in Section 4.6.

4.3 Clocks

Each clock defined in the SystemC module is replaced by an OS thread. Its startup routine generates the clock waveform and it accordingly wakes up processes sensitive to the corresponding clock. Clocks are preserved by step 1 of the proposed refinement flow. However, at step 2, clocks are used only for HW threads while they are ignored by SW threads, since they are meaningless for SW application.

4.4 Ports, Signals and Data Types

Ports and signals have been re-implemented by means of an object that instantiates an area on the shared memory to memorize the port/signal value, and a mutex for preventing race conditions due to concurrent accesses. Moreover, the redefined version of input ports (`sc_in`) and signals (`sc_signal`) memorizes the list of threads corresponding to processes sensitive to the port/signal itself. Such a list is used by the thread manager to wake up sensitive processes when the port/signal value changes. Methods for reading, writing and binding ports and signals have been also redefined, as well as a buffering strategy that allows to delay the updating of signal values to the next clock cycle, as required by the SystemC semantics. On the contrary, SystemC data types do not exploit functionality defined in the simulation kernel, thus they have been completely reused inside `systemc2os.h`.

4.5 The Sensitivity List

SystemC processes are generally sensitive to ports and signals. When a port/signal changes its value, the sensitive processes

*/*The function finds the semaphore related to the process to be stopped and it calls a wait on it.*/*

```
void wait(){
  activation_list *is_me;
  is_me = find_thread(THREAD_SELF());
  SEM_WAIT(is_me->semaphore);
}
```

Figure 6: Redefinition of the `wait()` function.

are waked up and they restart the execution from the beginning of their body (SC_METHOD) or from the instruction subsequent to the wait() that stopped their previous execution (SC_THREAD). Sensitivity lists are declared in the SC_MODULE constructor by means of the SystemC keywords sensitive, sensitive_pos or sensitive_neg that have been re-defined in the systemc2os.h library. By using our library, when a sensitivity declaration is encountered in the SC_CTOR, every signal and port declared in the sensitivity list adds the corresponding process to the list of processes that must be waked up when the signal/port value changes.

4.6 Process Suspension

In SystemC, process synchronization is mainly obtained by means of the function wait(). It must be explicitly called inside the body of SC_THREADs each time a synchronization point is desired. When a wait() is reached, the calling SC_THREAD suspends itself until an event occur. Thus, the wait() function has been redefined to call the wait() primitive (SEM_WAIT() in Figure 6) of the semaphore associated to the thread. On the contrary, wait() functions cannot be used inside SC_METHODs. They stop themselves once the last instruction of their body has been executed. Thus, to stop the SC_METHOD execution, respecting the SystemC semantics, we explicitly call the wait() function redefined in systemc2os.h inside the loop implemented in the redefinition of the SC_METHOD keyword (Figure 5).

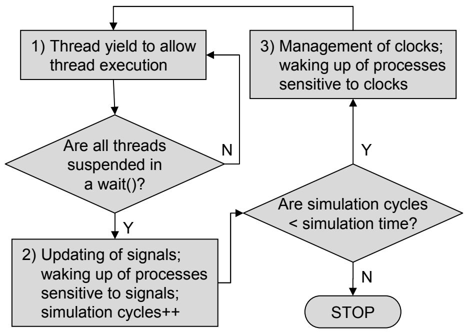
4.7 TLM Constructs

The SystemC transaction-level modeling (TLM) library is composed of a set of interfaces describing how models can communicate at transaction level. However, the implementation of such interfaces must be defined by the designers, thus they are independent from the SystemC simulation kernel. For this reason, the mapping of the TLM constructs is straightforward.

TLM is composed of three abstraction layers. At the highest layer (level 3 or message layer), the system is described by means of a set of function calls, as it was pure SW, without using SystemC active entities (i.e., threads). The communication is obtained by calling read() and write() functions whose implementation is a designer task. At level 2 (or transaction layer), the system is composed of a set of SC_THREADs communicating through hierarchical channels. The interface of these channels allows threads to communicate by calling blocking or non-blocking get() and put() functions. The redefinition of such functions inside systemc2os.h has been enough to allow the removal of the SystemC simulation kernel without affecting the simulation semantics. Finally, level 1 (or transfer layer) is very similar to RTL. Level 1 descriptions are clocked cycle-accurate models where the traditional RTL pin interface is abstracted away. Thus, level 1 models are handled by the thread manager without effort with respect to RTL models.

4.8 The Main Routine

The SystemC simulation starts by calling the sc_start() function inside the sc_main() routine. While sc_main() has been simply redefined as the traditional main() function of C programs, sc_start() has been completely rewritten. The new version starts, manages and stops the execution of OS threads corresponding to SystemC processes. In particular, sc_start() connects threads related to HW components to the thread manager, while it simply starts the execution of threads corresponding to SW functionality.



1) Thread yield to allow thread execution
3) Management of clocks; waking up of processes sensitive to clocks
Are all threads suspended in a wait()?
N
Y
Y
Are simulation cycles < simulation time?
2) Updating of signals; waking up of processes sensitive to signals; simulation cycles++
N
STOP

Figure 7: Thread manager execution.

5. HW Threads Simulation

After step 1 of the proposed refinement flow, the SystemC simulation kernel is removed, and all threads are connected to the thread manager. The same happens for HW threads after step 2. Such a manager preserves the standard SystemC behavior for connected threads, but it exploits a thread library to concurrently execute them, instead of using the co-routine execution mode of SystemC.

At the beginning of the simulation, all the threads are suspended on a conditional variable waiting for a broadcast notification from the thread manager. When the notification arrives, the threads are unlocked, and they concurrently compete for being assigned to the CPU(s). Once the simulation has started, the manager handles the connected threads as shown in Figure 7. Since the manager is a thread too, it yields the CPU, by means of the THREAD_YIELD() function, to allow the execution of other threads. Then, when the CPU is reassigned to the manager, it checks if all the threads related to SystemC processes have been executed and are suspended on the corresponding semaphore inside a wait() function. When such a condition is reached, the manager updates the values of signals and it wakes up processes sensitive to signals whose value has been changed. This is obtained by unlocking the semaphores associated to the thread. Then, the simulation time is advanced and a new clock edge is generated for each clock defined in the design. This wakes up processes sensitive to clocks starting a new simulation cycle. The simulation stops when the simulation time expires.

It is worth noting that during its execution, a thread can be preempted by the OS before it reaches a wait() function, while this cannot happen for a traditional SystemC process. However, this is not a problem, since if the thread manager is resumed, it continuously yields the CPU until all threads are suspended in a wait(). Thus, the SystemC semantics is preserved.

6. SW Threads Simulation

The thread manager is used to control only threads related to HW components. On the contrary, SW threads are simulated by exploiting the synchronization primitives provided by the underlying OS ignoring the semantics imposed for HW components by the thread manager. Thus, SystemC code related to SW functionalities execute like it was a pure C++ program.

The scheduling of SW threads is totally handled by the hosting OS. These threads ignore clocks and they instantaneously update signal values that are immediately available for other threads. Thus, SW threads that were sensitive to the clock are never suspended and they continuously run. Besides, asynchronous processes, waiting for generic events on a wait(), resume their execution immediately after the event happens without respecting the SystemC semantics. In this way, the behavior of a SystemC module rep-

Design	SystemC (s.)	Step1 (s.)	Step2 (s.)	Step3 (s.)
switch	120	4140	3900	5721
voip	320	116	109	10256
fr+bus	327	178	174	12658

Table 1: Simulation time.

resenting a SW functionality may be different when `systemc.h` is replaced by `systemc2os.h`. In particular, different behaviors are observed if designers rely on the SystemC synchronization semantics when they design the system. On the contrary, this does not happen if designers explicitly add synchronization mechanisms (e.g., semaphores, monitors, mutexes, conditional regions, etc.) inside the SystemC code. Thus, designers can easily refine SW threads and evaluate the effectiveness of the adopted synchronization mechanism.

7. Experimental Results

The proposed refinement flow has been evaluated, on an Intel Xeon Server equipped with four 3MHz CPUs and 4GB RAM running RedHat Linux 9.0, by using three different systems: an extended version of the Multicast Helix Packet Switch example distributed with SystemC (*switch*), a multimedia embedded system for transmitting voice over IP composed of a signal generator, an adaptive differential pulse code modulation coder and some filtering modules (*voip*), and, finally, two modules of a face recognition system connected by an AMBA bus (*fc+bus*). These examples have been selected since they have different characteristics: *switch* is an I/O-bound system with minimal computational effort, *voip* is a CPU-bound system with few communication exchanges, and *fc+bus* is a CPU-bound system with a medium communication rate with respect to *switch* and *voip*.

Table 1 shows the simulation time computed by adopting the proposed refinement flow. Column *SystemC* reports the simulation time corresponding to the initial SystemC descriptions before step1 takes place. Columns *Step1* and *Step2* refer to the proposed refinement flow. In particular, they show the simulation time computed, respectively, after the substitution of the co-routine-based SystemC kernel with our thread manager, and after the subsequent HW/SW partitioning. Finally, Column *Step3* refers to the simulation time computed after step3 by using the timing-accurate co-simulation methodology described in [4].

The comparison between the original SystemC simulation and the thread manager-based simulation (step1 and step2) shows conflicting results for the considered examples. As expected from the observations reported in Section 3, the overhead introduced for the thread synchronization by our thread manager is predominant with respect to the little computation time required by the I/O-bound *switch* example. Thus, the original SystemC simulation is faster than the multi-threading simulation performed after step1 and step2, even if the co-routine execution model of SystemC prevents the concurrent exploitation of the multi-processor hosting machine. On the contrary, the simulation of the CPU-bound examples (*voip* and *fc+bus*) takes benefit from the possibility of concurrently exploiting the multi-processor hosting machine guaranteed by our thread manager. Regarding the comparison between the thread manager-based simulation and the co-simulation proposed in [4], the simulation time computed after step 1 and step 2 is lower than the co-simulation time computed after step3 for all the examples. This emphasizes that exploring different HW/SW configurations according to the proposed refinement flow, before applying co-simulation, guarantees a sensible time saving, not only from the point of view of the coding effort, but also for the simulation of the design under refinement.

8. Concluding Remarks

In this paper, we proposed a smooth refinement flow for HW/SW co-design that should be applied before co-simulation. Co-simulation allows an accurate performance evaluation of the whole design after HW/SW partitioning, but it makes difficult the evaluation of different HW/SW configurations, since moving HW modules towards SW (or vice versa) generally requires time-consuming manual modifications. On the contrary, our approach allows designers to model HW/SW embedded systems at system level and evaluate different HW/SW configurations during partitioning by using an homogeneous SystemC-based environment. However, the co-routine execution model of SystemC, which is not suited for SW refinement, is substituted by a thread manager that respects the SystemC semantics of HW threads, leaving the control of SW threads to the underlying OS. In this way, SW threads can be refined and tuned by introducing opportune synchronization mechanisms before moving towards a co-simulation strategy.

References

[1] J. Buck, S. Ha, E. Lee, and D. Messerschmitt. *Ptolemy: A Framework for Simulating and Prototyping Heterogeneous Systems*. International Journal in Computer Simulation, vol. 4(2):pp. 155–182, 1994.

[2] F. Balarin, M. Chiodo, P.Giusto, H.Hsieh, A.Jurecska, L.Lavagno, C.Passerone, A.Sangiovanni-Vincentelli, E.Sentovich, K.Suzuki, and B.Tabbara. *Hardware-Software Co-Design of Embedded Systems: The Polis Approach*. Kluwer Academic Press, 1997.

[3] D. Desmet, D. Verkest, and H. D. Man. *Operating system based software generation for systems-on-chip*. In *Proc. of ACM/IEEE DAC*, pp. 396–401. 2000.

[4] L. Formaggio, F. Fummi, and G. Pravadelli. *A Timing-Accurate HW/SW Co-Simulation of an ISS with SystemC*. In *Proc. of IEEE CODES+ISSS*, pp. 152–157. 2004.

[5] S. Honda, T. Wakabayashi, H. Tomiyama, and H. Takada. *RTOS-centric hardware/software cosimulator for embedded system design*. In *Proc. of IEEE CODES+ISSS*, pp. 158–163. 2004.

[6] A. A. Jerraya and W. Wolf. *Hardware/Software Interface Codesing for Embedded Systems*. IEEE Computers, vol. 38(2):pp. 63–69, 2005.

[7] Z. Xiong, M. Zhang, S. Li, S. Liu, and Y. Chao. *Virtual embedded operating system for hardware/software co-design*. In *Proc. of IEEE ASICON*, pp. 939–943. 2005.

[8] P. Coste, F. Hessel, P. L. Marrec, Z. Sugar, M. Romdhani, R. Suescun, N. Zergainoh, and A. Jerraya. *Multilanguage Design of Heterogeneous Systems*. In *Proc. of IEEE CODES*, pp. 54–58. 1999.

[9] C. Valderrama, F. Nacabal, P. Paulin, and A. Jerraya. *Automatic VHDL-C Interface Generation for Distributed Co-Simulation: Application to Large Design Examples*. Design Automation for Embedded Systems, vol. 3(2/3):pp. 199–217, 1998.

[10] L. Semeria and A. Ghosh. *Methodology for Hardware/Software Co-verification in C/C++*. In *Proc. of IEEE ASP-DAC*, pp. 405–408. 2000.

[11] D. Harel, H. Lachover, A. Naamad, A. Pnueli, M. Politi, R. Sherman, A. Shtull-Trauring, and M. Trakhtenbrot. *STATEMATE: A working environment for the development of complex reactive systems*. IEEE Trans. on Software Engineering, vol. 16(4):pp. 403–414, 1990.

[12] D. Gajski, J. Zhu, R. Domer, A. Gerstlauer, and S. Zhao. *SpecC: Specification Language and Methodology*. Kluwer Academic Press, 2000.

[13] J. Aynsley and D. Long. *SystemC Language Reference Manual*. Open SystemC Initiative, 2005.

[14] I. Moussa, T. Grellier, and G. Nguyen. *Exploring SW Performance Using SoC Transaction-level Modelling*. In *Proc. of IEEE Design Automation and Test in Europe*, pp. 120–125. 2003.

[15] H. Patel and S. Shukla. *Towards a heterogeneous simulation kernel for system-level models: a SystemC kernel for synchronous data flow models*. IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems, vol. 24(8):pp. 1261–1271, 2005.

[16] http://www.synopsys.com. *Synopsys CoCentric System Studio*.

[17] D. Edenfeld, A. B. Kahng, M. Rodgers, and Y. Zorian. *2003 Technology Roadmap for Semiconductors*. IEEE Computer, vol. 37(1):pp. 47–56, 2004.

[18] L. Gauthier, S. Yoo, and A. A. Jerraya. *Automatic Generation and Targeting of Application-Specific Operating Systems and Embedded Systems Software*. IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems, vol. 20(11):pp. 1239–1301, 2001.

[19] F. Herrera, H. Posadas, P. Sanchez, and E. Villar. *Systematic Embedded Software Generation from SystemC*. In *Proc. of IEEE DATE*, pp. 10142–10147. 2003.

[20] H. Yu, R. Domer, and D. Gajski. *Embedded Software Generation from System Level Design Languages*. In *Proc. of IEEE ASP-DAC*, pp. 463–4468. 2004.

[21] J. Chevalier, M. de Nanclas, L. Filion, O. Benny, M. Rondonneau, G. Bois, and E. M. Aboulhamid. *A SystemC Refinement Methodology for Embedded Software*. IEEE Design and Test of Computers, vol. 23(2):pp. 148–158, 2006.