Interface-Centric Abstraction Level for Rapid Hardware/Software Integration

André C. Nácul[†], Marcello Lajolo[‡], Tony Givargis[†]

[†] Computer Science Department
University of California, Irvine
{nacul, givargis}@uci.edu

[‡] NEC Laboratories of America
Princeton, NJ
lajolo@nec-labs.com

Abstract

With the continuous advances of high-level synthesis and hardware/software codesign, engineers have now the luxury and the desire to explore very quickly multiple high-level architectures. System-level tools can enable trade-offs of architectures that rely on different combinations of memory access, resource sharing and multiplexing. A good system-level design flow must enable fast and accurate viewing of multiple solutions based on different design choices. In this paper we present a system-level API for text-based specifications that combines transaction-level modeling for the hardware interface and OS and device drivers levels for the software interface into a unified semantics. We also present a refinement process that allows to generate very rapidly a hardware/software integration.

1 Introduction

The design of embedded systems is growing in complexity at a fast rate. Devices are more feature-rich than ever, incorporating new functionalities, newer protocols, and more modes of operation. At the same time, designers must keep pace to deliver a new generation of products in an even tighter time-to-market. Coupled with the growth in chip capacity, the task is daunting, and calls for a simpler design flow.

Automation is one solution to cope with the growth in system design complexity. Automating the generation of code has long been used in different stages of embedded systems design, for instance, in hardware synthesis. More recently, software and operating systems have also been the focus of automation efforts [BB03] [HPSV03] [NG04].

Aside from automation, the design flow also needs to maximize design reuse and portability. Ideally, functional blocks should be easily migrated from hardware to software, and vice versa. A higher level of abstraction in describing the system's functionality facilitates this process. In such high abstraction, system specification can be carried in a portable fashion, independent of associated models of computation, hardware availability, communication architecture or specific operating system support.

The System Level Design community has tried to address these issues in different ways. We believe that a common programming interface is needed to allow designers to easily specify communication and iteration with an OS layer. SystemC [Theb] has tried to support a common interface. Nevertheless, in the current version, the support for software and OS is still not complete.

In this work, we propose a generic framework for system specification. Our framework is composed of a portable API, its corresponding semantics, and alternatives for hardware and software implementation for each entry of the API. The objective is to provide designers with a minimal set of high-level primitives that can be used to abstract and specify the system behavior. Our API is not dependent on any system-level design language. Rather, we are presenting a methodology to synthesize hardware, software and interface communication based on the proposed API. The API can be adapted or incorporated in the design language of choice. SystemC [Theb], for instance, includes support for some of the primitives we

present. Other primitives, however, are not present in SystemC, or have a behavior that is not the one we envision.

The API is partially based on the POSIX/Pthreads standard [TI04], and encompasses primitives for processes instantiation, communication with shared variables and message passing, process synchronization and timing behavior specification. The framework is integrated in our hardware/software codesign environment. When a functional block is mapped to a specific platform component, either hardware or software, we are able to automatically generate all the hardware descriptions, interconnections, software data structures and even device drivers that will effectively implement the semantics of the API entries. Therefore, with the use of an abstract, high-level system description, it is possible to automate hardware, software and communication interface generation, enhancing portability and reuse.

A description provided with this API can be easily mapped to different processors, various operating systems, and many communication architectures, without the need to modify any part of the original specification. The use of a portable specification also enables a rapid exploration of design alternatives. Since the proposed API is platform and implementation independent, one can easily test different communication architectures or different operating systems much quicker than when using platform specific code like the AMBA Bus API [ARM03] or VxWorks OS API [Win].

This paper is organized as follows. Section 2 discusses the previous works related to our proposal. We define the terminology we will use in this paper in Section 3. Section 4 describes our API, presenting possible mapping alternatives. In section 5, we present an example of hardware/software integration process starting from the proposed API. We present our conclusions in section 6.

2 Related Work

System level design has been the motivation for many publications in the literature. Most of the approaches address one of the components of synthesis, be it communication, hardware, or software synthesis. The synthesis of (real-time) operating system support has been studied more recently. However, none of the approaches integrate all the parts into one framework, as we propose in this work.

Cadence's Virtual Component Co-design Environment (VCC) was an earlier tool which tried to provide a design space exploration environment for SoCs. It used library components to synthesize the design. VCC lacks a complete path to implementation, though. In this sense, Dziri et al. [DSW+03] have combined VCC to other tools in order to provide a complete path to implementation. The main difficulty was integrating the multiple design tools, each of them using a different specification model.

Concerning interface synthesis and analysis, Meyerowitz has presented a tool [MPSV03] that can evaluate different bus architectures and arbitration protocols. He shows that response time and bandwidth utilization can improve by combining different arbitration protocols. Passerone et.al. [PRSV98] generates interface adapters, allowing IP blocks to communicate even with incompatible protocols. A transaction-level model for the AMBA bus in SystemC 2.0 is presented by Caldari et.al. [CCC+03]. They propose a set of high-abstraction classes to model communication interfaces. This is similar to the work presented by Coppola et al. [CCGM03] also on synthesis of communication interfaces.

There have also been proposals addressing automatic generation of RTOSes for SoC architectures. However, most of them are concerned only with the RTOS generation, and do not integrate hardware synthesis or custom communication infrastructures. Some examples of these works are the proposals of Le Moigne et.al. [LPC04], Besana et.al. [BB03], and Herrera et.al. [HPSV03]. Gerstlauer et.al. [GYG03] propose to model the RTOS functionality with System Level Language primitives, refining the RTOS with the specification. His work is integrated in the SpecC environment.

In terms of System Level Design Languages, the two most developed approaches are SystemC [Theb] and SpecC [GDPG01]. The transaction level modeling provided by SystemC supports modeling of hardware and software components, tied together with different communication interfaces. Software suppport, and specially RTOS support is not yet fully integrated in the language, though, and is the subject of the discussions for the next SystemC release. The same applies to SpecC, which still does not have a fully integrated RTOS interface.

3 Terminology

The terminology regarding operating systems, hardware and software design is way overloaded. Different terms are used to refer to the same concept, and the hardware and software communities do not have a common terminology. Since we are trying to define a System Level API suitable for hardware and software implementations, it is mandatory that we define the terms that will be used throughout this work. Two terms are specially of interest to us, be it process, tasks, and threads; and concurrency and parallelism.

In the software and OS community, threads refer to units of execution, while processes also include resource allocation. A thread is generally viewed as a light process, because there is no need to allocate a separate memory space, file tables, page tables and other structures that are common to processes. Usually, a process may contain many threads. In the RTOS community, the term task is used to represent an execution job, usually implemented by a thread. On the other hand, there is no thread concept in the hardware community. Instead, the term process is more common, and often refers to some sort of processing unit, like a processor or an ASIC. When describing two processes in hardware, we typically end up with two datapaths and two controllers. In this paper, we will use the term *process* to designate unit of execution, be it a hardware or software implementation.

Similarly, the terms parallel and concurrent are used interchangeably. In this work, *parallel* processes are those truly executing in parallel, i.e., at the same time, without multiplexing. Therefore, we need distinct hardware in order to be able to run processes in parallel. On the contrary, *concurrent* refers to processes that are competing for the same hardware, usually in a time-multiplexed fashion. To the user, they are seen as if they were executing in parallel, but in reality only one of them will be executing in one processor at any given clock cycle.

4 System Level API

The proposed generic API for design specification is presented in Table 1. It is partially based on the POSIX [TI04] standard, a well defined and accepted programming interface for Operating Systems, and includes extra primitives that are not part of POSIX. The API is divided in four parts: Process Management, Communication, Synchronization and Timing. Process management includes functions to control process creation and execution. The Communication part encompasses shared memory and message-passing based communication, both blocking and non-blocking style. Synchronization includes primitives for process synchronization, like mutexes, semaphores and condition variables. Finally, the Timing section allows some control over the timing behavior of the system, providing a timed-wait and controlling timeouts for blocking operations.

The API is thought to be integrable with any system-level specification language like, for instance, SystemC. The API represents the abstract functionality we believe is needed to facilitate the design of hardware devices and the specification and synthesis of OS-based software. Some design languages might already include an equivalent form of part of the API. SystemC, for example, has its own classes for mutexes (`sc_mutex`) and semaphores (`sc_semaphore`), that work very similarly to those presented here. In that case, the native classes can be used. Other entries of the API are not available in SystemC or any other design language, and therefore must be included.

The API functions are inspired by POSIX and Transaction Level Modeling (TLM). Process Management and Synchronization primitives are largely based on POSIX. There is a clear one-to-one mapping of the API entries to POSIX primitives. These are more likely to be used in software descriptions. Meanwhile, communication primitives are the highly abstract send and receive typical of a TLM description, along with shared memory access, useful for both hardware and software designs. The range of specification styles possible to target with the API is very broad. Hardware oriented specifications might use bit manipulation and low level constructs more intensively, while software oriented specifications could use pointers, memory allocation and stack manipulation more frequently. Nevertheless, the API we propose is neutral and can accommodate either style.

Table 1: The API functions

Process Management	`process_create(id, param, func, arg)`
	`process_delete(id)`
	`process_suspend(id)`
	`process_resume(id)`
Communication	`port_send(port, data, size, mode)`
	`port_receive(port, size, mode)`
	`shared_mem_read(mem, offset, size, mode)`
	`shared_mem_write(mem, offset, data, size, mode)`
Synchronization	`mutex_lock(mutex)`
	`mutex_unlock(mutex)`
	`sema_wait(sem)`
	`sema_post(sem)`
	`cond_var_wait(var, mutex)`
	`cond_var_signal(var)`
	`cond_var_broadcast(var)`
	`sched_yield()`
Timing	`time_wait(time)`
	`process_join(id)`
	`mutex_lock_tmo(mutex, time)`
	`sema_wait_tmo(sem, time)`
	`cond_var_wait_tmo(var, mutex, time)`

In the Process Management section of the API, four functions are defined. The function `process_create` is used to instantiate and start the execution of a new process. The argument `func` is the entry point function of the process. Note that the actual code of the process, be it hardware or software, must be already available. The API function will create a new context for the new process and start executing the initial function. Also note that in case of hardware processes, if more than one process share the same hardware implementation, there is a need to synthesize a scheduler within the hardware implementation, so that time sharing of the hardware is possible. `Process_delete` stops and removes a process from the scheduler list forever, freeing all the resources that were held by that process. Finally, `process_suspend` and `process_resume` are used to stop and resume the execution of a process, respectively. A process is suspended by a `process_suspend` call, and stays suspended until some other process executes `process_resume` for that specific process.

Two different communication models are supported in the API, message passing and shared memory. Message passing is abstracted by the concepts of ports, and provides the primitives `port_send` and `port_receive` to implement the communication. Blocking and non-blocking styles are supported, and are specified by the designer through the argument `mode`. A blocking send blocks the sender until the receiver reads the message. Similarly, a blocking receive blocks the receiver until a message is available in the corresponding port. Shared memory communication is modeled with the `shared_mem_read` and `shared_mem_write` primitives. Here, two styles are also possible, synchronous and asynchronous, specified in the `mode` parameter. In synchronous mode, a lock is associated with each shared memory block, and only one process can access the memory at one specific time. Meanwhile, the asynchronous mode does not have a lock associated with the memory, and therefore concurrent accesses can happen. It is up to the programmer to ensure the correct behavior of the accesses. In all communication primitives, the size of the data block to be transmitted or received is specified in the `size` parameter. In case the data size is larger than the specified width of the communication interface, a protocol will have to be implemented to ensure that the data is correctly partitioned in the sender, and received and reassembled in the receiver.

In the Synchronization section, three different synchronization mechanisms are defined by the API:

mutexes, semaphores and condition variables. A semaphore is a synchronization mechanism that controls access to shared devices or data structures. A semaphore is initialized to a specific count value *C*, representing the number of available devices or the number of concurrent accesses possible. A call to `sema_wait` will block the calling process if the semaphore value is zero, meaning that none of the shared resources are available, while a call to `sema_post` increments the value of the semaphore, and unblocks a possibly waiting process. Mutexes are similar to binary semaphores, i.e., semaphores initialized with the value of one. The process calling `mutex_lock` will block in case the mutex value is zero, and `mutex_unlock` will set the mutex value to one, allowing one of the possibly waiting processes to continue. Furthermore, condition variables allow processes to wait for some event or condition to happen. The process calling `cond_var_wait` will block until the condition is met and the corresponding `cond_var_signal` is invoked. Alternatively, `cond_var_broadcast` can be used to signal an event when multiple processes should resume execution as a result of one event. Finally, the last entry in the synchronization section is `sched_yield`, which is an explicit release of the processing unit. In SW implementations, it will result in a context switch, while in a HW implementation, it will be equivalent to forcing a clock boundary in the execution.

Lastly, the Timing section allows the specification of the timing behavior of processes. Processes can wait for a fixed amount of time using the API called `time_wait`. The waiting time is provided in the parameter `time`. Additionally, it is also possible to specify timeouts for each of the blocking synchronization primitives, with `sema_wait_tmo`, `mutex_lock_tmo` and `cond_var_wait_tmo`. These functions behave exactly like the corresponding non-timed-out versions, except that a maximum blocking time is provided as an additional parameter of the function, and an exception will occur in case the timeout is reached. In this case, the designer should handle the error appropriately.

4.1 Interface Synthesis

When the input design description contains communication primitives from the System Level API, there is a need to synthesize the communication interface between the processes. Depending on the design partitioning, the interface will need to connect two hardware modules, two software modules, or a hardware and a software module. In this section, we show examples of custom interface synthesis for different partitions. We refer to the process sending data as the producer, and the process receiving data as the consumer.

4.1.1 Hardware-to-Hardware communication

In the case where two processes that communicate through ports are mapped to a hardware implementation, there are different alternatives for interface synthesis. However, since this is a hardware to hardware communication, it is not necessary to generate RTOS code or software to handle this specific communication.

One possible architecture for a port-based hardware to hardware communication is shown in Figure 1. In this case, there is a direct data connection between producer and consumer. Additionally, control lines are synthesized according to the API usage. If the port is ever used for a blocking send, then an acknowledge line from the consumer to the producer is necessary. Therefore, the producer is suspended until it receives an acknowledge from the consumer in case of a blocking communication. For communications with multiple consumers, the producer wait for the acknowledge of all consumers. This behavior is implemented with a logic OR of the individual acknowledges of the consumers, as shown in Figure 1. Similarly, an event line is added from the producer to each consumer for the case when blocking receives are specified. Since the event and acknowledge control signals are only synthesized when needed, they are shown with dashed lines in Figure 1.

Other architectures are also possible from the same System Level API. For instance, it is possible to generate a Transaction Level Model with AMBA-bus transactions for each port primitive. In this case, the `port_send` and `port_receive` primitives are replaced by a set of calls to the AMBA Transaction-Level API [ARM03].

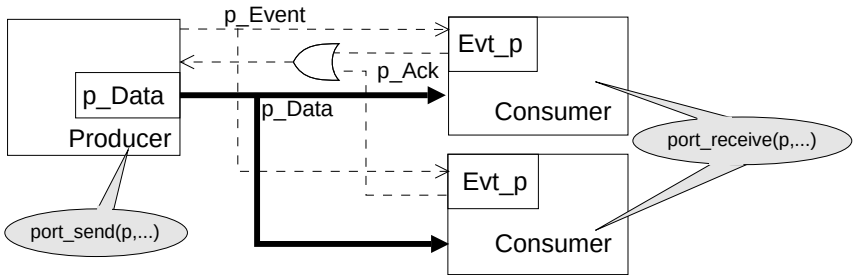

Figure 1: Interface Synthesis for HW-to-HW Communication

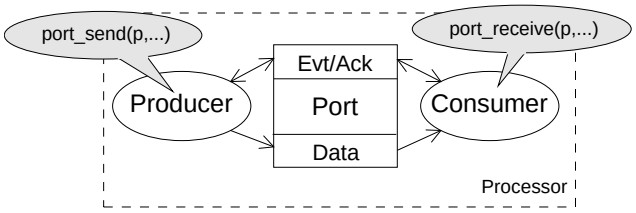

Figure 2: Interface Synthesis for SW-to-SW Communication

4.1.2 Software-to-Software communication

When two software processes are mapped to the same processor, the interface synthesis is simpler. Our framework will generate a software data structure in memory, shared between the processes, that will keep the data along with event and acknowledge control signals. All the producer has to do is to update two memory locations, with data and event signaling (in case of blocking receives), while the consumer will read the data memory and update the acknowledge bit of the same port. Figure 2 shows the interaction between the processes.

4.1.3 Hardware-to-Software communication

Hardware to software communications can be implemented by either interrupts or polling, using memory-mapped addresses in the latter case. In both cases, we will need some RTOS support in order to coordinate the processes. One possible solution is shown in Figure 3. Our framework will generate a bus adaptation layer for the hardware module, so that it can send and receive data from the bus. In the case of a memory-mapped communication, a device driver is also generated and runs inside the processor, monitoring the bus for activity in the memory-mapped region. The device driver is responsible for transferring data from the bus to the processor memory, to a port structure equivalent to the one shown in Figure 2. The software process will access the port data structure as it did in the software-software case, retrieving data and updating event flags. If instead an interrupt-based communication is specified, then an Interrupt Service Routine (ISR) needs to be synthesized. The ISR will be responsible for receiving the event signaling from the producer. In the interrupt-based communication, the actual data is still transferred through a memory-mapped location to the port structure.

4.1.4 Software-to-Hardware communication

In software to hardware communications, the producer is running in a processor, communicating with a hardware module. In our model, this kind of communication is always memory-mapped. The producer will update a *port* data structure, and a device driver propagates data and events to and from the bus. Events and acknowledge signals are generated for the receiver whenever necessary.

Note that the device driver can be unique for all the software-to-hardware and hardware-to-software communications. It has to monitor a set of software ports, transferring data to the bus, as well as monitor the bus for memory-mapped communications.

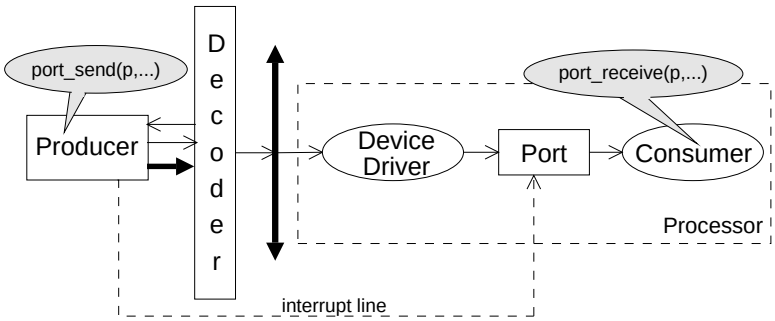

Figure 3: Interface Synthesis for HW-to-SW Communication

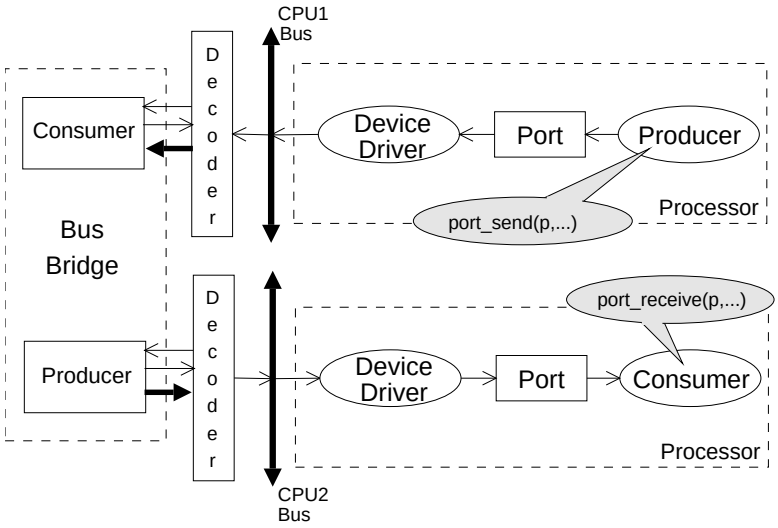

Figure 4: Interface Synthesis for Multiprocessor Communication

4.1.5 Multiprocessor communication

Finally, in case the processes are mapped to different processors, with different busses, a bridge will also be synthesized. Figure 4 shows the proposed architecture. In this scenario, the producer runs on processor 1, connected to System Bus 1, while the consumer runs on processor 2, connected to System Bus 2. The producer will see the bridge as the consumer, characterizing a software to hardware communication. Meanwhile, the consumer will see the bridge as the producer, therefore a hardware to software. The port will be accessed through a memory-mapped address. In addition to the bridge, device driver code is synthesized for both processors, linking the software process to the RTOS and to the bridge hardware.

For shared memory communication, two different architectures are possible, depending on synchronous or asynchronous communication. In the synchronous mode, a locking structure is generated for each shared memory, so that access is granted exclusively for each process. Every memory access has to obtain the lock first. In the asynchronous mode, only the memory is synthesized. The locking mechanism is implicit in the API call for shared memory access. Every shared memory will be directly connected to the system bus, accessible by the CPU. Additionally, a dedicated memory port will be available for each hardware module accessing the memory, so that using the bus is not necessary while accessing shared data. Therefore, there is less contention and higher parallelism in the implementation.

4.2 RTOS Synthesis

In addition to communication interface synthesis, the generation of RTOS support is required. In this case, our System Level API has to be mapped to OS-specific resources, adapting the generic API to

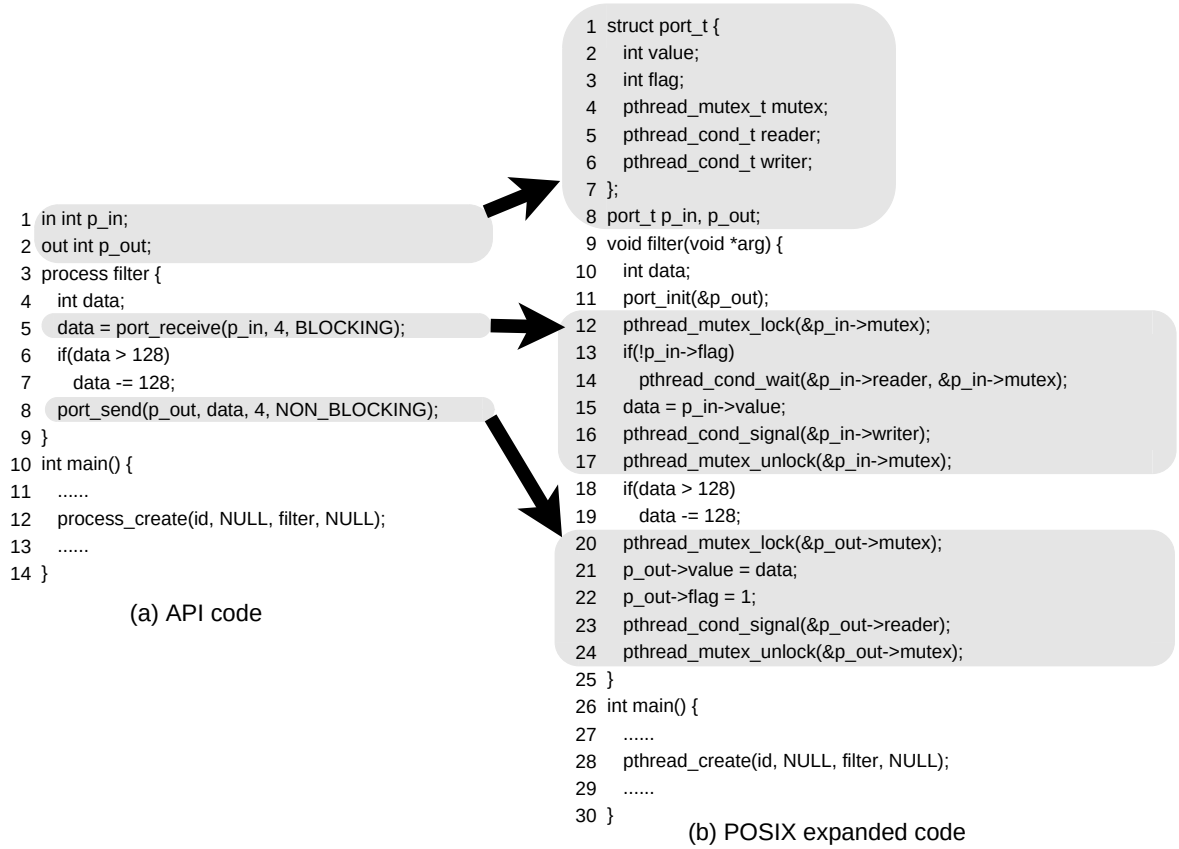

Figure 5: Code Example

the functionality available in the target RTOS. Since our API is based on POSIX, the mapping is trivial when targeting a POSIX-compliant OS, like Embedded Linux [Thea] or eCos [Mas02]. Alternatively, our tool is able to target non-POSIX RTOSes by mapping the API calls to the specific RTOS. Finally, the API-based description is used as input to tools that generate a customized OS infrastructure, like Polis [Bal97] and Phantom [NG04], in case a custom-generated RTOS was specified.

Figure 5 shows the code generation process for our System Level API. In Figure 5(a), the design is specified with the API primitives in a C-like specification. Figure 5(b) shows the generation of C code for a POSIX-compatible OS. The API primitives are expanded to POSIX code. Some primitives have a direct transformation to a POSIX call. Others need to be expanded into more than one POSIX instruction. This is the case with send and receives and synchronous shared memory access.

Note that a data structure is generated for the port-based communication (lines 1 to 7), as discussed earlier. The `port_send` (line 8) and `port_receive` (line 5) are expanded to POSIX/Pthread calls for mutexes and condition variables (lines 20 to 24 and lines 12 to 17), and updates the port data structure, reading data (line 15), sending data (line 21) and setting event flags (lines 16 and 22). The generated code also include the corresponding checks (line 13) and waits (line 14) in the case of the blocking port receive.

5 HW/SW Integration

5.1 Our HW/SW Codesign Environment

Our codesign framework provides an interface and a set of tools for synthesizing and simulating a design. Input to our codesign environment is a set of modules M_1, $M_2 \ldots M_n$ that implement a design. Modules are described in a C-based system level language, extended with the proposed API functions. Each

module represents a process. Next, the mapping step partitions the design into hardware and software implementations. The partitioning granularity is at the process level, i.e., once a process is mapped to hardware or software, all its functionality is synthesized to execute as a hardware block or a software task inside an RTOS. Currently, the partitioning process is manual. Once the design is partitioned, the designer specifies the communication parameters. In case of hardware to software communications, for instance, it is possible to determine the use of interrupts. Finally, hardware, software and interfaces are synthesized.

Hardware synthesis is handled by an in-house behavioral synthesizer, that produces synthesizable RTL for each module. Software modules are generated according to the operating system support desired by the designer. For each target software environment, we provide a library that implement the specified API for the referred environment. At the time, our codesign framework can generate software modules based on the POLIS [Bal97], the Phantom Compiler [NG04] and any POSIX-based operating system, like Embedded Linux [Thea] or eCos [Mas02] with the POSIX adaptation layer. In the later case, for instance, there is a one-to-one mapping of some of the API functions to the POSIX library of the operating system, while others require some expansions, as shown in Figure 5.

Software is compiled to a specific processor, which can be a NEC V850 or an ARM946. Finally, the interface is generated according to the partition and the communication style specified. We have simulators available that allow us to simulate the synthesized hardware, selected processor (cycle accurate in the case of V850 and instruction based on the case of ARM), software and communication interfaces.

5.2 Putting It All Together

The steps of our hardware-software integration process are depicted in Figure 6. The block diagram on top helps in visualizing the connections and processes. In this example, which implements a matrix multiplication algorithm consisting of three processes: the Index Control, that controls the execution of the algorithm, the Data Retriever, which fetches data from the shared memory, Mem, and passes them, two at a time, to the module MAC which multiplies them, accumulates intermediate results and finally writes the result back into the shared memory. So Data Retrieve only reads data from the shared memory, while MAC only writes into it.

An excerpt of the specification code is shown in the specification section of Figure 6. The specification contains some of the API calls proposed in this work incorporated into a C-based specification. This specification is the input to our codesign environment.

The process mapping is specified next. In the example of Figure 6, the Index Control and the Data Retrieve processes are mapped to SW, running in a single processor, while the Multiplier is mapped to a custom HW module. During the mapping stage, it is also necessary to specify the options for the communication interfaces. For the ports between Index Control and Data Retrieve, the communication will be internal to the processor, since this is a software to software communication. Therefore, a software port structure will be generated, like the one shown in Figure 5(b). For the case of the ports between Data Retrieve and Multiplier, we chose a memory-mapped communication to implement the hardware to software and software to hardware communications. Alternatively, interrupts could have been used for the communication that originates in the Multiplier.

In the Code Generation stage, the environment synthesizes the communication interface, along with the software targeted at the OS specified by the user and the hardware modules. In this stage, the RTOS is adapted to the application. Since there is a memory-mapped communication, the appropriate device driver for the communication will be incorporated into the software code. An address decoder is generated for the Multiplier process, so that it can access the bus. A locking structure is synthesized around the shared memory block. The lock is needed to support the synchronous operations in the memory access. Software-mapped processes are expanded to a POSIX specification, which can be compiled against POSIX-compliant OSes. The same POSIX code can still be used to generate application specific OS infrastructure, such as POLIS [Bal97] or Phantom [NG04].

Meanwhile, hardware-mapped processes result in the generation of a low-level SystemC description, to be synthesized with the appropriate tools. This detailed description includes the necessary extra

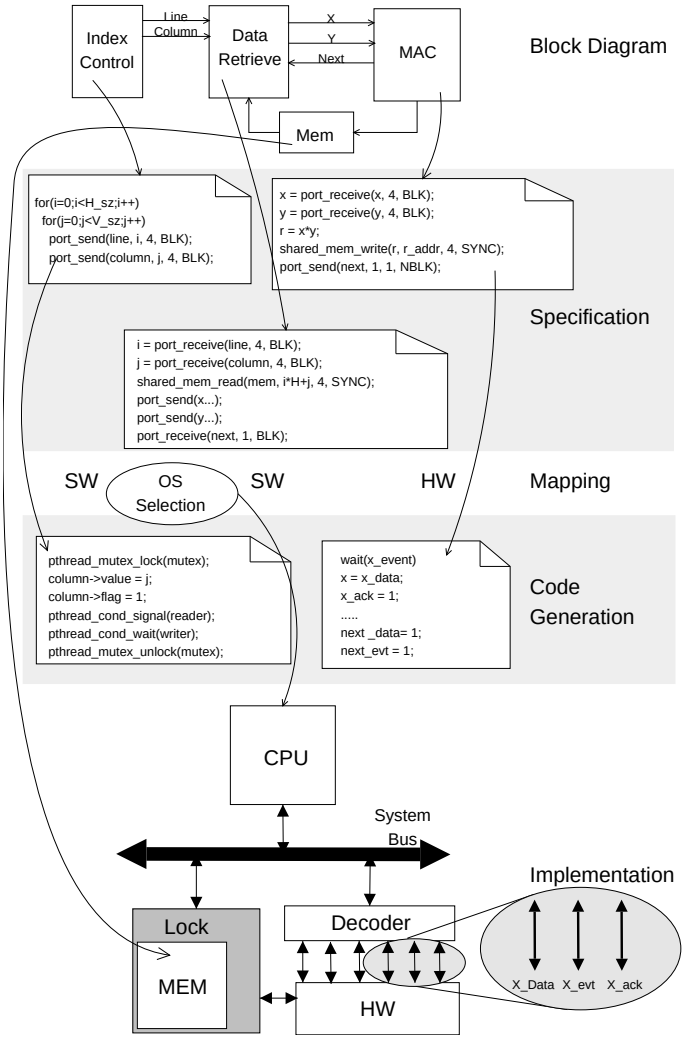

Figure 6: HW/SW Integration

hardware and interconnections that implement the hand-shaking control discussed in section 4.1.

The final architecture is shown in the Implementation section of Figure 6. The CPU will be executing the two processes mapped to software, supported by the RTOS specified in the mapping stage. The CPU is connected by a System Bus to the Hardware and Shared Memory modules. An address decoder connects the hardware to the bus. Finally, the shared memory incorporates the lock to support the synchronous communication specified earlier in the design, providing a dedicated access port to the hardware module. Note the expansion of the connections from the hardware module, with the inclusion of the connections for event and acknowledge for each port from software to hardware and hardware to software. The figure shows the expansion for the X port, and the other ports are expanded similarly.

The final generated hardware and software architecture is simulated an internally developed, cycle-accurate simulator. We are able to simulate single and multi-processor architectures, along with memory, buses, bridges, and reconfigurable logic to implement hardware-mapped modules. Currently, our simulator supports the NEC V850 processor and can provide cycle-accurate execution data. Additionally, we have a instruction-accurate model of the ARM946 processor.

6 Conclusions

Current complexity of embedded systems is driving a consensus toward the need for a higher abstraction level support for system specification. This will result in more opportunities for design reuse and better

design space exploration capabilities. In this context, synthesis of operating system and hardware/software interfaces is needed.

In this paper, we have introduced a System Level API that provides a specification support for rapid hardware/software integration by combining into a unified semantics both transaction level modeling for hardware specifications and OS and device drivers layers for software specifications. We have shown how this API can be easily integrated in any current System Level Design language and we have discussed its utilization into a hardware/software codesign flow.

References

[ARM03] ARM Limited. AMBA AHB Cycle Level Interface Specification, 2003.

[Bal97] F. Balarin et.al. *Hardware-Software Co-Design of Embedded Systems: The Polis Approach*. Kluwer Academic Publishers, 1997.

[BB03] Monica Besana and Michele Borgatti. Application Mapping to a Hardware Platform Through Automated Code Generation Targeting a RTOS. In *Proc. of DATE*, Feb. 2003.

[CCC+03] M. Caldari, M. Conti, M. Coppola, S. Curaba, L. Pieralisi, and C Turchetti. Transaction-Level Models for AMBA Bus Architecture Using SystemC 2.0. In *Proc. of DATE*, Feb. 2003.

[CCGM03] M. Coppola, Stephane Curaba, Miltos Grammatikakis, and Giuseppe Maruccia. IPSIM: SystemC 3.0 Enhancements for Communication Refinement. In *Proc. of DATE*, Feb. 2003.

[DSW+03] M. Dziri, F. Samet, F. Wagner, W. Cesario, and A. Jerraya. Combining Architecture Exploration and a Path to Implementation to Build a Complete SoC Design Flow from System Specification to RTL. In *Proc. of ASP-DAC*, Jan. 2003.

[GDPG01] Andreas Gerstlauer, Rainer Doemer, Junyu Peng, and Daniel Gajski. *System Design: A Practical Guide With SpecC*. Kluwer Academic Publishers, 2001.

[GYG03] A. Gerstlauer, H. Yu, and D. Gajski. RTOS Modeling for System Level Design. In *Proc. of DATE*, Feb. 2003.

[HPSV03] F. Herrera, H. Posadas, P. Sanchez, and E. Villar. Systematic Embedded Software Generation from SystemC. In *Proc. of DATE*, Feb. 2003.

[LPC04] R Le Moigne, O. Pasquier, and J-P. Calvez. A Generic RTOS Model for Real-time Systems Simulation with SystemC. In *Proc. of DATE*, Feb. 2004.

[Mas02] Anthony Massa. *Embedded Software Development with eCos*. Prentice Hall, 2002.

[MPSV03] Trevor Meyerowitz, Claudio Pinello, and Alberto Sangiovanni-Vincentelli. A Tool for Describing and Evaluating Hierarchical Real-Time Bus Scheduling Policies. In *Proc. of DAC*, Jun. 2003.

[NG04] A. Nacul and T. Givargis. Code Partitioning for Synthesis of Embedded Applications with Phantom. In *Proc. of ICCAD*, Nov. 2004.

[PRSV98] Roberto Passerone, James Rowson, and Alberto Sangiovanni-Vincentelli. Automatic Synthesis of Interfaces between Incompatible Protocols. In *Proc. of DAC*, Jun. 1998.

[Thea] The Embedded Linux Consortium. http://www.embedded-linux.org.

[Theb] The Open SystemC Initiative. http://www.systemc.org.

[TI04] The Open Group and IEEE. IEEE Std 1003.1, 2004. Available online at http://www.opengroup.org/onlinepubs/009695399/toc.htm.

[Win] Wind River Inc. http://www.windriver.com.