Development of Hybrid Components for Embedded Systems

Hugo Marcondes and Antônio Augusto M. Fröhlich
Federal University of Santa Catarina
Laboratory for Software and Hardware Integration
PO Box 476, 88049-900
Florianópolis, SC, Brazil
{hugom,guto}@lisha.ufsc.br

Abstract

Embedded Systems are increasing in complexity, requiring the use of an adequate design methodology in their conception. These methodologies must deal with several metrics associated with the design of embedded systems. In order to attend these metrics, several software engineering techniques are being applied in embedded system design, as component-based design. Moreover, a design based on higher-level abstraction enable a better design space exploration between several hardware and software compositions. We define a hybrid components as a development artifact that can be deployed by different combinations of hardware and software elements. Nevertheless, devising the proper interface for such component is certainly not a straightforward task. This paper presents a strategy to handle the construction of those hybrid components that delivers architectural transparency to clients, enabling the achievement of desired design metrics, through an effective design space exploration.

1. Introduction

Embedded systems are pervasive in our daily lives, from brake control systems in our vehicles to smart appliances in our homes. Hardware production technology advances and manufacturing costs reductions have promoted the development and popularization of very complex embedded applications. In this sense, the Application-Oriented System Design (AOSD) methodology [5] guides the development of embedded systems through a domain engineering process that aims at yielding components that can be promptly reused in a variety of application-specific systems. The portability of such components—and thus of applications using them—across distinct hardware platforms is achieved by means of a construct named *hardware mediator*, which defines a hardware/software interface contract between higher-level components and the hardware [10].

Hardware mediators are meant to be implemented using Generative Programming techniques [4] and, instead of building an ordinary *Hardware Abstraction Layer* (HAL), implicitly adapt existing hardware components to match the required interface by adding software to client components. For example, the hardware mediator for a hardware component that already presents the desired interface would be totally eliminated during the system generation process; while the hardware mediator for a hardware component that does not provide all the desired functionality could exceed the role of interface and include software elements to complement the hardware functionality.

Indirectly, the concept of hardware mediator defines a kind of *hybrid hardware/software component*, since different mediator implementations can exist for the same hardware component, each designed around a particular set of goals such as performance and energy efficiency. If the hardware platform can be itself synthesised—as is the case with IP-based platforms—then the notion of a hybrid component becomes even more appealing, since some hardware mediators could exist in different pre-validated combinations of hardware and software.

The idea of *hybrid hardware/software component* behind AOSD hardware mediators also constitutes an important tool for design of embedded systems. The fact that hybrid components in this sense exist in advance enables a scenery in which components can be tagged with information about required silicon area and features, energy consumption, performance, reliability, cost (e.g. associated royalties), and whatever other metric becomes convenient, thus sustaining an effective exploration of design space.

Nonetheless, devising the proper interface for a hybrid component—specially the more complex ones, such as CODECs, storage systems, and interconnects—and designing it in a way that is flexible enough to support implementations that freely combine hardware and software elements is certainly not a straightforward task. This paper presents a strategy to handle these issues through a well-defined hybrid hardware/software component *architecture*. This architecture emerged during the implementation of several hybrid components, such as the scheduling algorithm used in the case study of the proposed architecture. Those components was developed in the scope of the Embedded Systems Development Environment Project (PDSCE). [1].

The forthcoming sections are dedicated to explain the hardware mediator construct and its connotation of hybrid component in depth, to present the case studies that yielded the proposed hybrid component architecture, and to discuss the architecture itself.

2. Application-Oriented System Design

Application-Oriented System Design (AOSD) is a domain engineering methodology that elaborates on the well-known domain decomposition strategies behind Family-Based Design (FBD) and Object-Orientation (OO), i.e. *commonality* and *variability* analysis, to add the concept of *aspect* identification and separation yet at the early stages of design [5]. In this way, AOSD guides a domain engineering towards families of components, of which execution scenario dependencies are factored out as ”aspects” and external relationships are captured in a component framework. This domain engineering strategy consistently addresses some of the most relevant issues in component-based development:

Reusability: components tend to be highly reusable, for they are modeled as abstractions of real elements of a given domain and not as parts of a target system. Moreover, by factoring out execution scenario dependencies as aspects, components can be reused unmodified in a variety of scenarios simply by defining new aspect programs.

Complexity management: the identification and separation of execution scenario dependencies implicitly reduces the number of components in each family, since those components that would have been modeled to express a variation in the domain that originates from a scenario dependency are suppressed whenever the dependency can be modelled as an aspect. Simply stated, a set of 100 components could be modeled as a set of 10 components plus a set of 10 aspects and a mechanism to apply aspects to components. The overall complexity (and functionality) in the new set of 100 generated components is the same, but it is now confined in fewer constructs. This directly improves on maintainability.

[1]The PDSCE Project is funded by FINEP grant no. 01.04.0903.00.

Composability: by capturing component relationships in a component framework, AOSD enables components to be more easily combined while generating a system instance. It also put some limits to the misbehaviors that can arise from applying aspect programs to pre-validated components. *Feature-based models* are of great value at this point to capture configuration knowledge and thus make system generation a more predictable procedure.

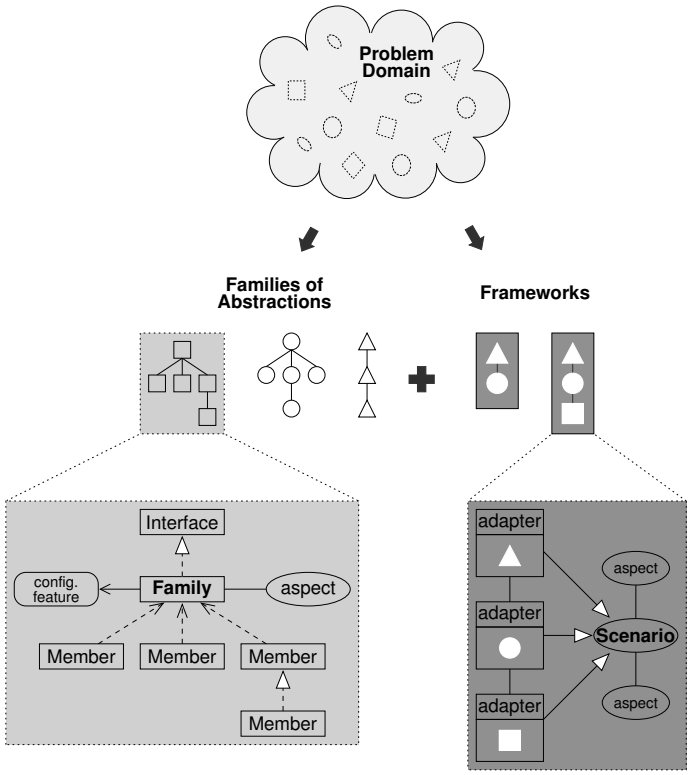

Figure 1. Overview of domain decomposition guided by AOSD.

Figure 1 illustrates the main elements of an AOSD domain decomposition, with domain entities being captured as abstractions that are organized in families and exported to users through comprehensive interfaces. Abstractions designate scenario independent components, since scenario dependencies are captured as aspects during design. Subsequent factorization captures configurable features as constructs that can be reused thorough the family. Relationships between families of abstractions delineate a component framework. Each of these elements are subsequently modelled according with the guidelines of Object-Oriented Design (OOD).

3. Hybrid Hardware/Software Components

As described in the introduction of this article, *hardware mediator* is the concept of AOSD responsible for preserving the architectural independence of high-level system components. Although the portability aspect of hardware mediators was consistently discussed in a previous paper [7], illustrating the role of a hardware mediator with a real case study will be of great use while trying to demonstrate how hybrid hardware/software components emerge from it. Therefore, the mediators involved in the management of processes in an experimental system will be revisited here.

In EPOS, processes management is delegated to the `Thread` and `Task` abstractions. `Task` abstractions corresponds to the activities specified in the application program, while `Threads` are the

entities that perform such activities. Some of the main requisites and dependencies of these system abstractions are deeply related with the architecture of the target processor, which is mediated by the `CPU` hardware mediator. For example, the execution context of a process comprises the values stored in user-visible registers of the processor, and the stack structure is determined by the Application Binary Interface (ABI) of the processor. Details like these are encapsulated in `CPU` and hidden from `Thread` and `Task`.

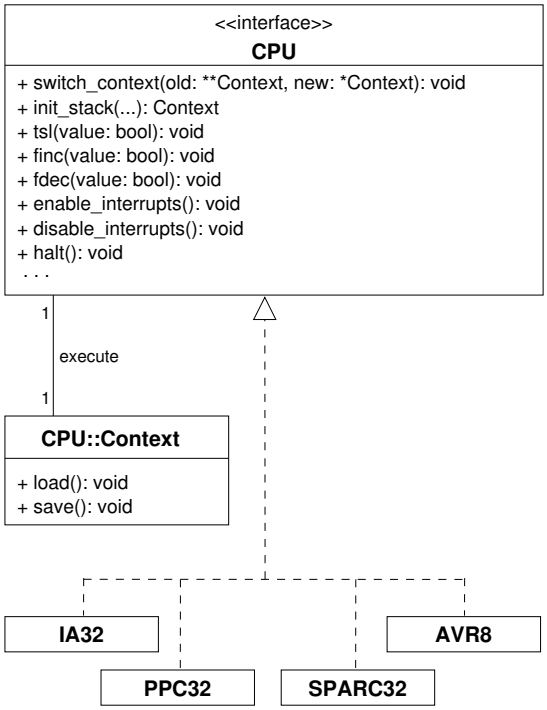

Figure 2. Overview of the CPU hardware mediator.

Figure 2 depicts some interface elements of the `CPU` mediator. The class `Context` defines all the internal data structures that must be stored for any given execution flow. As part of the `CPU` mediator, this class is redefined for every new architecture the system is intended to run on. `Thread` and `Task` simply use it as a black-box. The context of a thread is thus represented by an object that is dynamically stored on the thread's stack. A pointer to the location where the context is currently stored is maintained as an attribute of `Thread` that is implicitly updated by the method `CPU::switch_context()`.

Another architecture dependency in process management is related to stack initialization. In EPOS, a thread can be created to execute any ordinary function in the program (i.e., `Task`), regardless of the number of parameters it has and also regardless of the fact is encodes an explicit call to `Thread::exit()`. In order to sustain this programming model, the stack of a thread must be pre-initialized with the corresponding function parameters, as well as a return address that will properly guide the execution flow throughout `Thread::exit()`. However, compilers for different architectures use different function calling conventions, and having the `Thread` abstraction to manipulate the stack by its own would render undesirable architectural dependencies. The solution is to have a meta-program that is able to properly initialize the stack inside the `CPU` mediator. This interaction between `Thread` and `CPU` is illustrated by figure 3, which depicts the steps involved in creating (steps 1.*) and scheduling (steps 2.*) threads[2].

[2]If the *preemptive* feature is enabled then step 1.3 is also considered during thread creation (e.g., higher priority thread).

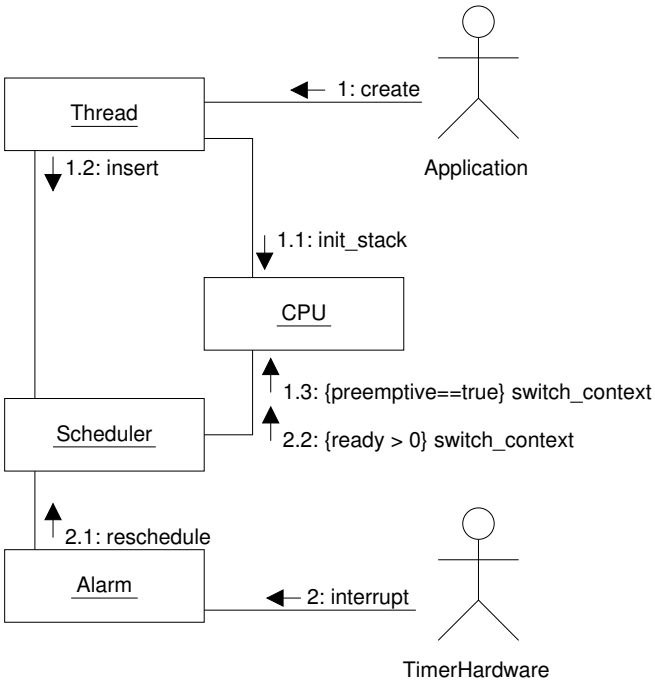

Figure 3. Thread creation and scheduling.

The `CPU` hardware mediator also implements some functionality for other system abstractions, such as bus-locked read-and-write transactions (i.e., *Test and Set Lock*), which is required by the `Synchronizer` family of abstractions, and endianness conversion (e.g. host to network and CPU to Little Endian) used by `Communicator`. The process scheduling algorithm is itself implemented by another abstraction that also uses `CPU`: the `Scheduler`.

With this example in mind, it is now easier to elaborate on the idea of hybrid hardware/software components emanating from hardware mediators. Consider, for instance, that a soft-core processor has bus-locked read-and-write memory transactions implemented as a configurable feature. Two members of the corresponding hardware mediator family could deliver the process synchronization mechanism alternatively in software or hardware and yet preserve the interface contract. These two mediators could thus be viewed by client components as a single hybrid component.

More sophisticated combinations could be devised for the `Scheduler` component, which could exist in a variety of shapes, including, for instance, a hardware-mostly implementation that features timers and queues; a software-mostly implementation that uses an external timer (`Alarm` in figure 3); a hardware/software implementation with caches, timers and policies in hardware, and queues in software.

The general form of such hybrid components is depicted in figure 4. Each hybrid component aggregates a hardware mediator that interfaces several software and hardware implementations. These implementations can be selected by the system developer in order to achieve the best compromise between performance, cost, energy consumption, silicon area, etc. The main challenge in the use of hardware mediators to construct a repository of hybrid hw/sw components is to design them in such a way that the interface with other components is preserved independently of the fact that sometimes the component will be implemented in hardware and other times in software.

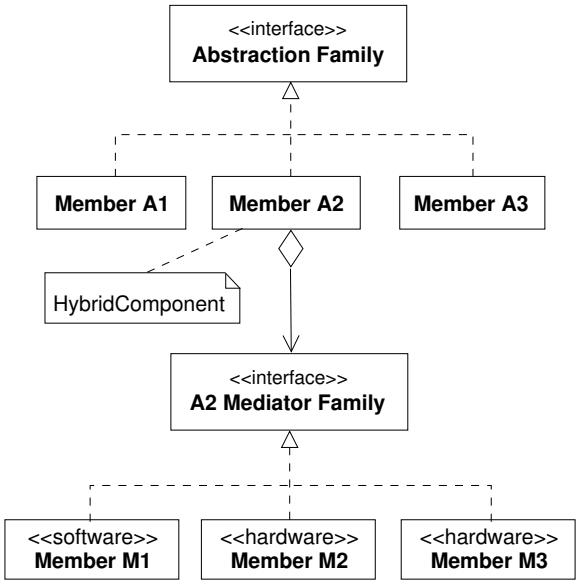

Figure 4. Hybrid hardware/software component organization.

4. Hybrid Components in PDSCE

During the PDSCE Project, some hybrid components were developed for EPOS as an answer to very distinct application requirements for a same hardware component. For instance, some digital television applications developed in the project, in particular the H.222 MPEG multiplex, called for strict real-time support and guided us toward the design of hardware-based scheduling and synchronizing components. Another application, the encoder, had far more trouble with high-performance algorithms and demanded advanced math support in hardware. Both applications coexisted on the same base architecture, the VirtexII-Pro, and used hybrid components selected and configured by EPOS system tools. The design and implementation of `Semaphore` and `Scheduler` hybrid components will be described next aiming at sustaining the reasoning about the systematization of the development of hybrid components presented in the next section.

4.1 Semaphore

A semaphore is a synchronization tool represented by a integer variable that can be accessed only by two *atomic* operations: `p` (from the Dutch *proberen*, to test) and `v` (from Dutch *verhogen*, to increment). In EPOS , this abstraction is realized by the `Semaphore` member of the `Synchronizer` family of components, which is outlined in figure 5.

Figure 6 illustrates the organization of the hardware semaphore IP. Basically, the IP has an internal memory that stores the semaphore's values and pointers to the blocked threads queues. The size of the internal memory is proportional to the maximum number of available semaphores, the maximum number of blocked threads and the number of bits that represent each semaphore value (e.g., 32-bits). The IP was integrated with the PLB bus using the IPIF interface provided by the XILINX [1].

When a semaphore is created, the IP performs a search within the internal memory to find out a free slot. If this operation succeeds, then the semaphore is set as valid and its `id` is returned to be referenced on subsequent commands (i.e., `p`, `v`, and `destroy`). For a `p` operation, the client must supply, in addition to the semaphore's `id` in the `Command_Register`, a pointer to the running thread in the `Thread_Pointer` register. These information is made available to the semaphore IP respectively via the `ID` and `DATA_IN` ports. If the operation causes the semaphore's value to become negative, then the running thread reference is inserted in the corresponding queue and a flag

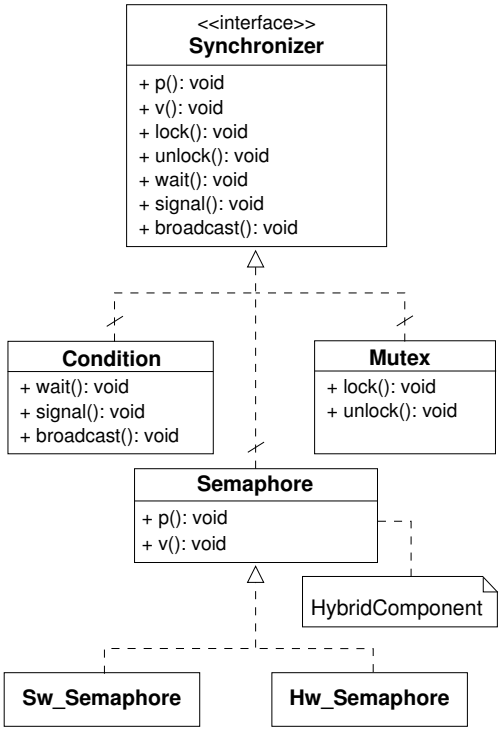

Figure 5. EPOS **family of synchronization components.**

in the `Status_Register` is set. The associated software routine can then invoke the scheduler. A `v` operation that causes a thread to be resumed reports the thread pointer via the same register and signalizes a flag on the `Status_Register`.

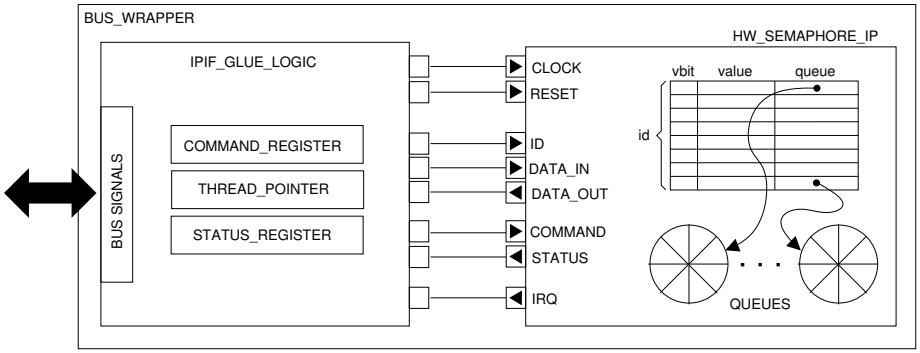

Figure 6. Overview of the hardware semaphore organization.

The final component was synthesised on the Virtex-4 ML 403 development board, using the Xilinx tools design flow. The table 1 present the FPGA area used by this component configured to allow the instantiation of 8 semaphores, each one holding a queue of 4 elements.

4.2. Thread Scheduler

For the PDSCE Project, an Priority-based scheduler was developed in the EPOS and gave rise to a hybrid component. Figure 7 illustrates the family of schedulers in EPOS. The `CPU_Scheduler` abstraction interacts with the `Threads` and `CPU` mediator described in section 3 to provide the mechanisms necessary to suspend and resume the execution of threads.

The hardware scheduler IP is depicted in figure 8. It has four main modules: scheduler, queue

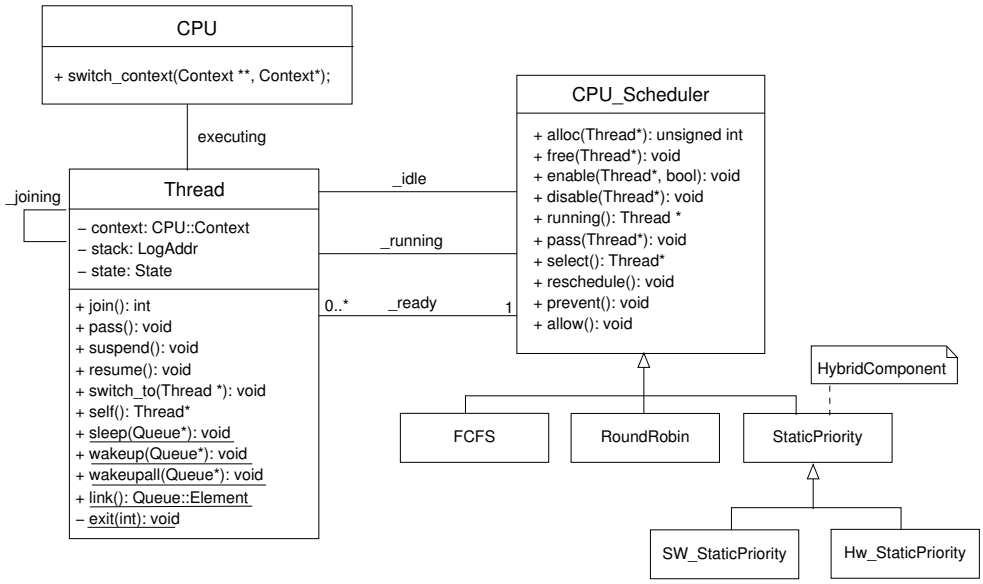

Figure 7. EPOS **family of scheduler components.**

control, time control and interface (IPIF). The *scheduler* module consists of two VHDL processes: alloc and main. The alloc processes is strictly related with the allocation table of *scheduler* as this process allocs new *threads* that are created in the system. In order to communicate with external components, i.e. threads, the *scheduler* must hold an word-size pointer to the instantiated ”object” in system memory. If these pointers was stored as the data on the *scheduler* queues, several word-size comparators will be needed to implement the scheduling and sorting logic of the queue. Insted of, a translator table is implemented to translate the word-size pointer address to a smaller pointer based on the maximum number of concurrent threads that the system will run. The main process is responsible for the management of threads on the queue, inserting them when ready, and removing when suspended. The *time control* module implements a hardware timer that generates *ticks* for the scheduler. The *queue control* module implements the scheduler's process list as a joint list of ready and suspended processes, thus saving FPGA area.

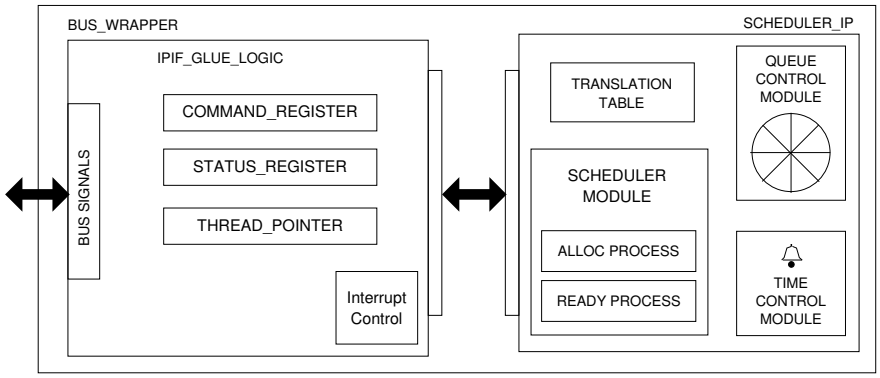

Figure 8. Overview of the hardware scheduler organization.

As the semaphore, the scheduler hardware component was synthesised on the Virtex-4 ML403 development board, using the Xilinx tools design flow. The table 1 presents the FPGA area used by the scheduler configured to handle at most 15 threads with 8 levels of Priority.

	Semaphore		Scheduler	
	w/ IPIF	w/o IPIF	w/ IPIF	w/o IPIF
Slices	1455	1327	1652	1520
FlipFlops	1455	1327	1652	1520
4-input LUTs	1455	1327	1652	1520

Table 1. Hybrid Components FPGA usage

5. An Architecture for Hybrid Components

The development of hybrid components for the PDSCE Project described in the previous section called our attention to several issues that seems to be inherent to the design of this kind of component. As mentioned earlier in this paper, the main issue is certainly architectural transparency for client components, which expect an hybrid component to behave identically whether a hardware, software, or mixed implementation is taken.

Analyzing how client components interact with their providers, we observed three distinct interaction patterns:

Synchronous: observed in components with sequential objects that only perform tasks when their methods are explicitly invoked; client components are blocked on the method call until service is completed.

Asynchronous: observed in components around active objects that perform tasks when their methods are explicitly invoked, but do not block the execution of the client component; some sort of call-back mechanism is used to notify the client about service completion.

Autonomous: components implemented as active objects that performs tasks independently of clients; the services provided by the component are either ubiquitous or generate events for clients.

The `Semaphore` component described in section 4 is an example of *synchronous* component, for any action derived from it originates on the calls to `p` or `v`. On the context of hybrid components, synchronous components can be easily shifted from software to hardware or vice-versa. When one such component is moved from software to hardware, the corresponding hardware mediator must block the clients until the hardware finish the requested service. This can be implemented on the hardware mediator either by polling a *status register* (a busy waiting mechanism) or by deploying a semaphore (an idle waiting alternative). ?For the opposite direction, that is, moving a synchronous component from hardware to software, the synchronicity is usually implicitly preserved by the method call mechanism on the processors.

Asynchronous components receive service requests via method calls just like synchronous ones, but differently from them, they do not block the calling client until the service is finished, allowing client and provider components to progress in parallel. Typical examples for this class of hybrid components are I/O related subsystems, such as file systems and communication systems. In order to get notifications about service completion, clients must register call-back functions or event handlers (e.g. UNIX signals). The moving of a asynchronous component from software to hardware is done by interrupts that activate the corresponding hardware mediator in order to trigger the original call-back mechanism. ?The opposite direction can be achieved with the use of concurrent programming techniques such as multithreading, with the call-back mechanism being triggered by software when the service is finished (i.e. thread exit).

Autonomous components execute their services independently of explicit client requests. The `Scheduler` component described in section 4, along with components such as garbage collectors and energy managers, is an example of autonomous component. The activity of this kind of component is usually driven by events. For instance, a scheduler is usually driven by a timer, a garbage collector is driven either by a timer or by the realization that the system is running out of memory, an energy manager is usually driven by activity counters in combination with power supply status. In this scenario, moving a hybrid component from software to hardware is feasible as long as the triggering events can be forward to the hardware component. The other way around is usually accomplished by having the hardware to generate interrupts to notify other components about general system status changes that might result from autonomous activities.

Hybrid components can act as client to other system components. When a hybrid component resides in software domain, the client communication is done through a method call to the desired component interface. When the hybrid component resides in hardware domain, the hardware generates an interrupt to request the desired service to the other component. This situation implies in the execution of the requested service within an CPU interrupt state, or the instantiation of a thread to execute the requested service. Both scenarios bring synchronization problems in the system execution, and this must be taken in account when selecting these components.

While dealing with these three categories of components in the PDSCE Project we realized that the hardware mediator concept of AOSD is indeed an underlying architecture for hybrid components. Components modeled according with AOSD guidelines were successfully transformed in hybrid components without major redesigns and based on well-know implementation techniques (i.e. status register polling, interrupts, and event handling). Furthermore, the interface contract inherent to hardware mediators is a major architectural transparency means when making a component hybrid.

6. Related Work

Several works in the field of hardware/software co-design have addressed issues pertaining architectural transparency and integration of components in heterogeneous environments.

Jerraya an Wolf analyses the evolution of Hw/Sw interface codesign techniques and defines a long-term roadmap for future success [6]. This work highlight several aspects presented by hardware mediator, as architectural transparency without incurring in excessive overhead.

Mooney proposed a framework to generate a partitioned hardware/software RTOS [8]. Independent of task requirements, this approach generates only one OS that is replicated on every processor. The designer does not have the flexibility to choose which components are implemented in software or hardware. Additionally, the designer cannot control the task mapping onto the target processors.

Nakano implemented a partitioned OS, called STRON (Silicon TRON) [9]. Nevertheless, the system does not allow choosing which components are going to be implemented in hardware and which ones are going to be developed in software.

Cesário et al. presents a high-level component-based design methodology that lets MPSoC designers handle hardware-software interfaces at a high abstraction level [3]. This approach integrates tools for hardware, software, and cosimulation wrapper generation that aids in the high-level modeling and generation of efficient hardware-software interfaces, helping on the development of hybrid components.

Anderson et al. presents the hthreads, a uniform programming model for specifying application threads running within a hybrid CPU/FPGA system [2]. Hthreads provides system service libraries that encapsulate platform specific operations under pthreads compatible API's, thus enabling the systematic migration of application functions through the software/hardware barrier providing an interesting way to implement asynchronous hybrid components.

7. Conclusions

This paper discusses a Hybrid Hw/Sw Components architecture based on the *hardware mediator* concept of Application-Oriented System Design. The concept was initially proposed by AOSD as a portability artifact, and was approached in this paper as a means to support design space exploration. A *hybrid component* defined around an AOSD hardware mediator can hide a series of implementations, some pure hardware, some pure software and some that are a combination of software and hardware. By tagging hybrid components with design constraints such as silicon area, energy consumption, and performance, effective exploration can be sustained.

The development of complex embedded applications in the PDSCE project corroborated such architecture, as components modeled within this project were successfully transformed in hybrid components without major redesigns and based on well-know implementation techniques. Moreover, this hybrid architecture can deliver architectural transparency for the client components, a key issue when dealing with design space exploration. This work is aiming now on the development of new hybrid components in order to achieve a consistent repository for further research on design space exploration tools and methods.

Acknowledgment

The work in this article is partially supported by FINEP - Financiadora de Estudos e Projetos under grant no. 01.04.0903.00.

References

[1] Xilinx logic core plb ipif. Technical report, Xilinx, 2005.

[2] E. Anderson, J. Agron, W. Peck, J. Stevens, F. Baijot, E. Komp, R. Sass, and D. Andrews. Enabling a uniform programming model across the software/hardware boundary. In *Field-Programmable Custom Computing Machines, 2006. FCCM '06. 14th Annual IEEE Symposium on*, pages 89–98, 2006.

[3] W. Cesario, D. Lyonnard, G. Nicolescu, Y. Paviot, S. Yoo, A. Jerraya, L. Gauthier, and M. Diaz-Nava. Multiprocessor soc platforms: a component-based design approach. *Design & Test of Computers, IEEE*, 19(6):52–63, 2002.

[4] K. Czarnecki and U. Eisenecker. *Generative Programming: Methods, Tools, and Applications*. Addison-Wesley, 2000.

[5] A. A. M. Fröhlich. *Application-Oriented Operating Systems*. GMD - Forschungszentrum Informationstechnik, 1 edition, 2001.

[6] A. A. Jerraya and W. Wolf. Hardware/software interface codesign for embedded systems. *IEEE Computer*, 38(2):63–69, 2005.

[7] H. Marcondes, A. S. H. Junior, L. F. Wanner, and A. A. Fröhlich. Operating systems portability: 8 bits and beyond. In *11th IEEE International Conference on Emerging Technologies and Factory Automation*, pages 124–130, 2006.

[8] I. Mooney, V.J. and D. Blough. A hardware-software real-time operating system framework for socs. *Design & Test of Computers, IEEE*, 19(6):44–51, 2002.

[9] T. Nakano, A. Utama, M. Itabashi, A. Shiomi, and M. Imai. Hardware implementation of a real-time operating system. In *TRON Project International Symposium, 1995., Proceedings of the 12th*, pages 34–42, 1995.

[10] F. V. Polpeta and A. A. Fröhlich. Hardware mediators: A portability artifact for component-based systems. In L. T. Yang, M. Guo, G. R. Gao, and N. K. Jha, editors, *EUC*, volume 3207 of *Lecture Notes in Computer Science*, pages 271–280. Springer, 2004.