A Hybrid Hardware and Software Component Architecture for Embedded System Design

Blind-Review

Abstract

Embedded systems are increasing in complexity, while several metrics such as time-to-market, reliability, safety and performance should be considered during the design of such systems. Frequently, the design of such systems imposes an integrated hardware/software design to cope with such metrics. In this sense, a component-based design methodology with components that can freely migrate through hardware and software domain benefits the design process of such systems. Moreover, a design based on higher-level abstraction enables a better design space exploration between several hardware and software compositions. We define hybrid hardware and software components as a development artifact that can be deployed by different combinations of hardware and software elements. In this paper, we will present an architecture for developing such components in order to construct a repository of components that can migrate between the hardware and software domains to meet the design system requirements.

1 Introduction

Several challenges arise on the design and implementation of current embedded systems. The applications themselves are becoming increasingly complex as the advances of the semiconductor industry enabled more sophisticated use of computational resources on a spread of market appliances. If at one side the applications are becoming more complex, on the other side the pressure of the market for rapid development of those systems makes the task of designing them a challenge.

The constraints imposed to such systems, in terms of functionality, performance, energy consumption, cost, reliability and time-to-market are getting tighter. Therefore, the task of designing such systems is becoming increasingly important and difficult at the same time [9]. Moreover, those systems could require an integrated hardware and software design that can be realized by a myriad of distinct computational architectures, ranging from simple 8-bit microcontrollers, digital signal processors (DSP), programmable logic devices (FPGA) to dedicated chips (ASIC) that provides the system functionality. In order to cope with these challenges, several methodologies were proposed by the hardware and software co-design community over the last decade. One approach to deal with these challenges is based on the concept of build a system based on the assembly of pre-validated components, like the *Platform-based design* [12]. However, designing such reusable artifacts to meet the requirements of several distinct applications should be as challenging as well [11]. The partition of the system between hardware and software also plays a key role in the design process. Usually, this mapping of system functionality into hardware implementation and software implementation is done in the initial phases of the specification of the system, enabling the development and implementation of the hardware and software occur concurrently. This approach however, is not ideal, as a mistake on this beginning phase of the project could lead to a re-engineering of the system, which can sometimes be too costly.

Our proposal to deal with these challenges is to use refined engineering techniques to build a repository of components that are flexible enough to provide components free of implementation domain. In this scenario, embedded systems could be built on such components that can be migrated to hardware or software domains without major redesigns to the system, according to the requirements of the application. To enable the construction of those flexible components, a set of engineering techniques was used. Domain Engineering was used to identify a set of representative entities within a domain. Such entities are modeled using Object-oriented design, Family-based design, and Aspect-orientation. A framework models the composition rules of such components, using advance techniques such as generative programming to ensure a low overhead to the composed system.

The next section will present the related work on hardware and software co-design. In section 3 the proposed architecture of hybrid hardware and software components is laid out. Three components built with this architecture are described and evaluated in section 4, followed by the conclusion of this paper.

2 Related Work

Several methodologies propose the integration of tools and design phases of embedded systems, to promote a rapid-prototyping and design of such systems. Metropolis [3] proposes the use of a unified framework, based on a metamodel with formal semantics that developers can use to capture designs, and an environment to support the simulation, formal analysis and synthesis of complex electronic systems, providing an adequate support to the design chain.

The Ptolemy project [7] focuses on the modeling design of heterogeneous systems, as mostly modern embedded computing systems are heterogeneous in the sense of being composed of subsystems with very different characteristics among their interactions as synchronous or asynchronous calls, buffered or unbuffered, etc. To deal with such heterogeneity, Ptolemy proposes a model structure and semantic framework that support several models of computations, such as *Communicating Sequential Processes*, *Continuous Time*, *Discrete Events*, *Process Network*, and *Synchronous Dataflow*.

While most of existent hardware-software co-design tools focus mainly on the hw-sw co-simulation to build a virtual prototyping environment for performing software design and system verification, PeaCE [8] appear as an extension to Ptolemy to provide a full-fledged co-design environment from functional simulation to system synthesis. It is targeted for multimedia applications with real-time constraints, specifying the system behavior with a heterogeneous composition of three models of computations and exploiting features of formal models maximally during the design process.

The use of a component-based design approach for multiprocessor SoC platforms are presented by [4]. This work proposes a unified methodology for automatic integration of heterogeneous pre-designed components effectively. A design flow called ROSES [6], uses this methodology to generate hardware, software, and functional interface sub-systems automatically starting from a system architectural model.

Another approach to deal with the component communication on multiprocessors SoC is based on the distributed system paradigm to provide a unified abstraction for both hardware and software components [10] that is deeply inspired by the concepts of communication objects standards such as CORBA. This approach uses the generation of a proxy-skeleton scheme to provide transparent communication architecture of the components in both domains (hardware and software).

HThreads [1], focus on specifying and unifying the programming model for hybrid CPU/FPGA systems, under the umbrella of multithreading programming. In this sense, they provide what they call *hardware thread interface* (HWTI) which supports the generalized pthreads API semantics, allowing for the passing of abstract data types between hardware and software. This approach enabled the migration of threads to the hardware domain, to be implemented as hardware accelerators. The HWTI interface provides access to the same system calls available to software threads, a globally distributed memory to support pointers, a generalized function call model including recursion, local variable declaration, dynamic memory allocation, and a remote procedural call model that enables hardware threads access to any library function [2].

3 Hybrid Hw/Sw Components

Hybrid Hw/Sw components can be realized as a mixture of hardware and software implementation that can vary from a component that realizes all your functionality in hardware to an implementation fully realized in software. The figure 1 depicts this concept, illustrating a full hardware implementation (A), a full software implementation (C) and a mixture of both (B).

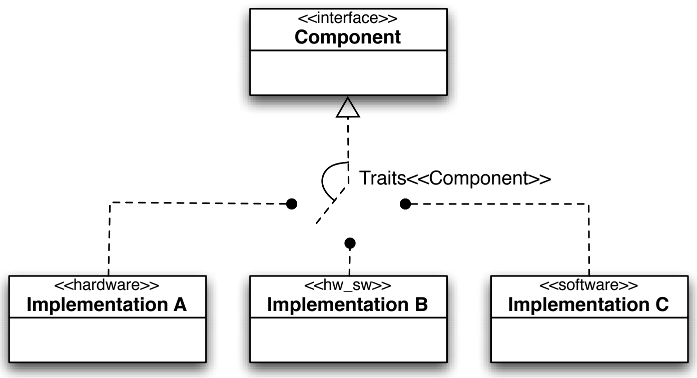


Figure 1: Hybrid components.

In this way, a system composed by such kind of components can adapt to its requirements according to the actual implementation selected to realize a specific interface, used by the application. For instance, a mobile application that requires an efficient use of energy could select an implementation that optimize such a metric to the detriment of others (i.e. cost), while applications that have an unlimited source of power could select implementations that benefit other metrics (i.e. performance, costs).

To illustrate such components, consider a very common component of most embedded system: a task scheduler. Such a component is mainly represented by a queue of elements that are ready to receive a resource from the system, usually the CPU, an ordering algorithm to establish the order in which the elements on the queue will receive the resource, and a timer responsible for managing the amount of time in which each element will receive of the resource (*quantum*). A hybrid hw/sw component could arise by pushing those elements to hardware or software domains in several combinations. As an example, the queue and the ordering algorithm could be realized in hardware to improve performance, at the expense of cost. The realization of the time management by the scheduler in hardware could also reduce the occurrence of interruptions of the CPU (to deal with time ticks that will not cause a rescheduling) that could lead to decreased energy consumption for instance.

Design reusable hybrid hw/sw components

Most of the methodologies in the design of embedded systems focus the design of each system independently. Although most of them consider the use and selection of pre-existent components already in the initial phases of the design process, most of them do not address how to guide the system development process to yield components that can be effectively reused on further projects. In fact, the construction of components that can be extremely reusable is one of the most challenging issues in *Platform-based design* [11]. Our proposal rests on the foundation of refined software engineering techniques to overcome such challenges and bring not only a flexible interface of components that can freely migrate between hardware and software domains, but also foster the reuse of the captured knowledge from previous projects in the form of reusable components.

To achieve such a degree of flexibility, it is essential to use a domain engineering methodology that elaborates on the well-known domain decomposition strategies, allied with Family-Based Design (FBD) and Object-Orientation (OO). In such an approach, the use of *commonality* and *variability* analysis captures the usage variations of the elements of the domain, than can be further factored out as aspects. In this sense, the use of such techniques guides the 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, addressing consistently some of the most relevant issues in component-based design, such as reusability, complexity management and composability.

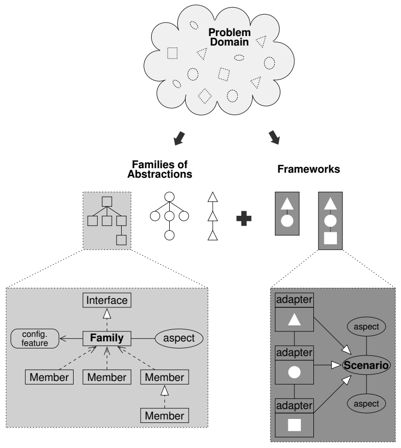


Figure 2: Overview of domain decomposition.

Figure 2 illustrate the main elements of 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 throughout the family. Relationships between families of abstractions delineate a component framework. Each of these elements is subsequently modeled according to the guidelines of Object-Oriented Design (OOD).

The portability of such components, and thus of applications that use them, across distinct hardware platforms is achieved by means of a construct called *"hardware mediator"*, which defines a hardware/software interface contract between higher-level components and the hardware. Hardware mediators are meant to be implemented using Generative Programming techniques [5] 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 eliminated totally 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 itself can be synthesized—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.

In fact, the flexibility that underlies the hardware mediator concepts is yielded from the domain decomposition processes that established a model that represents elements of the domain (concepts) and was not driven by a specific implementation of these concepts (no matter if they are hardware or software). In other words, this means that the interface provided by these components is free of implementation domain, and thus can be realized either as hardware or as software.

Hybrid Hw/Sw Component Architecture

In order to provide the seamless migration of the components between both implementation domains, not only should the interface be able to be realized in both domains, but also behave equally in both domains, avoiding the refactoring of the clients that use them. Analyzing how client components interact with their providers, we observed three distinct behaviors 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. Such behavior is intrinsic to software components, and can be preserved in hardware by means of its hardware mediator that can block client requests until the service is completed. The figure 3 illustrates an UML activity diagram of such behavior when the component is implemented on the hardware domain. The client requests the service to the component, which is executed while the client stands polling a register to be notified upon the finish of the service (busy waiting), or suspend itself until the hardware interrupts the CPU to resume the suspended client (idle waiting).

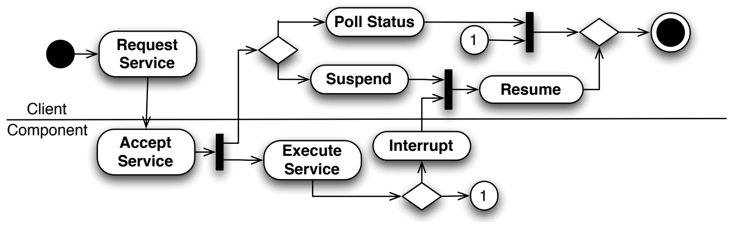


Figure 3: UML activity diagram of Synchronous Components.

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 callback mechanism is used to notify the client about service completion. Typical examples for this class of hybrid components are I/O related subsystems, such as file systems and communication systems. The figure 4 illustrates an UML activity diagram of such behavior. The client register a callback function if this is not already set (i.e. at initialization of the component) and then requests the service. Once the service is accepted by the component (i.e. the component is not servicing another request) the client continues it execution, while the component executes the service. When the requested service is finished, the component will call the registered call back function, which can be achieved by a simple call if the component is implemented in software or through an interrupt if the component is in software.

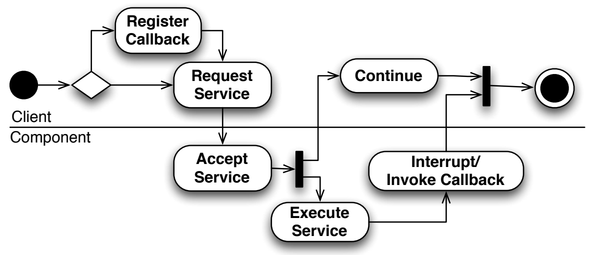


Figure 4: UML activity diagram of Asynchronous Components.

Autonomous: components implemented as active objects that perform tasks independently of clients; the services provided by the component are either ubiquitous or generate events for clients. Its behavior is depicted in figure 5, by a loop of service execution and event generation activities that could be interrupted by external events. 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 this 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.

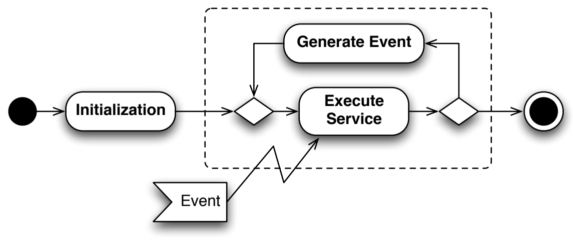


Figure 5: UML activity diagram of Autonomous Components.

The following section present three case studies that were designed according to the proposed architecture of components, and represent these three behavior patterns.

4 Case studies

To evaluate the proposed hybrid hw/sw component architecture, three components were developed in both implementation domains, each one representing a specific behavior. A *Semaphore* component, that behaves as a synchronous component, a *Scheduler* that behaves as an autonomous component and an *Alarm* component that behaves as an asynchronous component. The following sections describe the implementation of those components.

4.1 Semaphore

A semaphore is a synchronization tool represented by an 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). The software implementation of the component is realized by an object that aggregates the semaphore variable and a list of blocked threads that are waiting for the resource guarded by the semaphore abstraction. To guarantee the atomicity of its methods, the software implementation of the semaphore components uses the bus locking mechanisms of the underlying architecture, and when such a feature is not available, the atomicity is provided by masking the occurrence of interrupts.

The hardware implementation of the semaphore component, pushes each semaphore variable to a hardware implementation, and also manipulates the blocked threads queue on hardware. In this sense, four commands are implemented by the controller: `Create` and `Destroy`, responsible for allocation and deallocation of the internal resources (memory for the variable and the queue) and the other two traditional methods of semaphores `P` and `V`. For every `P` operation, the address of the caller of the method is passed through the input registers to the hardware, and if the caller has to be blocked its address is automatically inserted on the respective queue, and signalized by the status register. Once the resource becomes available (through a `V` operation) the address of the blocked thread waiting for the resource is removed from the queue, and put in the output registers. The need to resume the thread that is addressed on the output register is signalized by the status register.

4.2 Scheduler

The scheduler is responsible for organizing and defining the order that elements access a resource, when such a resource is shared among several elements. The most common use of a scheduler is to establish the order that tasks or process (elements) gain access to use the CPU to run (resource). Figure 6 depict the design of the process management family of components, where the `Scheduler` hybrid hw/sw components arise.

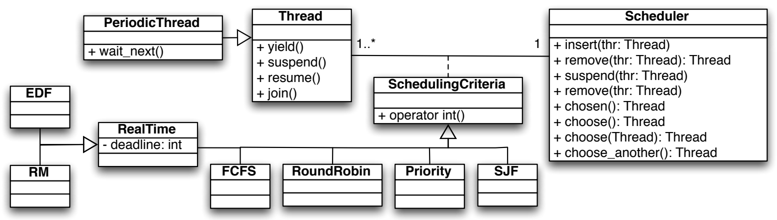


Figure 6: Process Management family of components

The `Scheduler` provides the basic implementation of methods to manipulate the queue of elements that are ready to use the resource managed by the scheduler, such as `insert()`, `remove()`, `resume()` and `suspend()`. A deeper explanation around the whole *Process management* family of components is beyond the scope of this paper. Let's focus here only on the component `Scheduler` that was implemented as a hybrid component. Such a component implements the fundamental structure of a scheduler, which consists of a queue of ready elements and time management mechanisms.

The implementation of the software scheduler follows the traditional design of lists. Such a list implementation could be configured to be realized as a conventional ordering list of its elements, as well as a relative list, where each element stores its ordering parameter relative to its predecessor. In this sense each element will hold the difference of its ordering parameter from the previous element, and so and on. In such kinds of implementations, it is particularly interesting when the scheduling policy has dynamic priority increases over time, such as the EDF policy, for example. In such a policy, as the absolute deadline is always a crescent value, the use of a conventional ordering, using the absolute deadline should lead to an overflow of the variable as the execution time is always growing (which can occur in a few hours on 8 bits microcontrollers). Instead of this, the use of a relative queue insures that the deadline is always stored relatively close to the current time, and in this way, the variable will not overflow.

The implementation of the scheduler in hardware has an internal memory that implements an ordered list. One module (`Controller`) is responsible for interpreting all the data received by the interface of the component in hardware and then, activates the process responsible for implementing the functionality requested by the user (through the `command` interface register). This implementation, as the software counterpart, realizes the insertion of its elements already in order, that is, the queue is always maintained in an orderly fashion, following the information that the `SchedulingCriteria` provides. In the memory of the component, a double-linked list is implemented.

It is worth highlighting two aspects of the implementation of this component regarding its implementation on hardware, especially for programmable logic devices. Both of these aspects are related to constraints in terms of the resources of such devices. Ideally, the hardware scheduler should exploit maximally the inherent parallelism of the hardware resources. However, such resources are very expensive, especially when the internal resources are used to implement several parallel bit comparators in order to search for elements in the queue, as well as, to find the insertion position of an element in queue.

Moreover, the use of 32 bits pointers, to reference the elements stored on the list (in this case `Threads`) becomes extremely costly, for implementing the comparators to search for those elements. On the other hand, the maximum number of tasks that a system will execute in an embedded system is usually known at design time, and for that reason, the resources usage of this component could be optimized implementing a mapping between the system pointer (32 bits) and an internal representation that uses only the number of bits necessary, taking into account, the maximum number of tasks running on the system.

4.3 Alarm

The `Alarm` component is responsible for providing the abstraction of an event generator to the system. This component behaves asynchronously, as its service (the event generation) occurs asynchronous from its request (Alarm instantiation). The figure 7 illustrate the design of the `Alarm` component. This component provides three types of event generation: call a function implemented by the user, resumes a blocked thread, or releases a semaphore (by calling it's `v()` operation). These events are supported by the `Handler` interface.

The software implementation of the `Alarm` component is implemented sharing a `Timer` used to manage the passage of time, and a relative queue of event requests. This queue is organized relative to the number of ticks missing for the occurrence of the event. At each interrupt of the `Timer`, the number of ticks is updated in the queue, and when this number is less than zero, the handler of the event is invoked. Its implementation on hardware implement dedicated counters for each supported `Alarm` component, implementing parallel countdown counters that generate interrupts to signalize events. The internal memory of the component is used to store a reference for each `Alarm` handler that is passed to the interrupt handler of the component to generate the respective event.

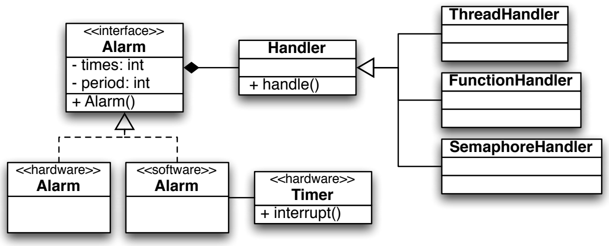
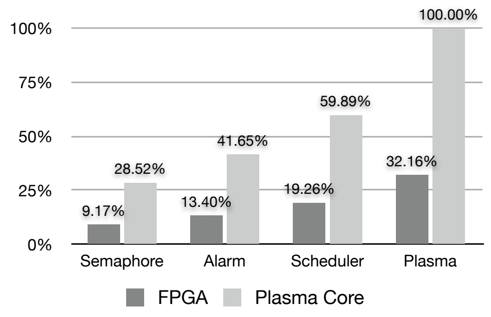


Figure 7: Alarm component design.

4.4 Results

An experimentation platform was used to develop, debug, and evaluate those components. It was used the XILINX development board `ML-403`, which has a VIRTEX 4 FPGA, that enable the instantiation of the hardware components on the configurable logic, while running the software components on its embedded POWERPC. The FPGA used on the platform was the `XC4VFX12`, which provides 5,412 slices of logic blocks for the implementation of the hardware accelerators.



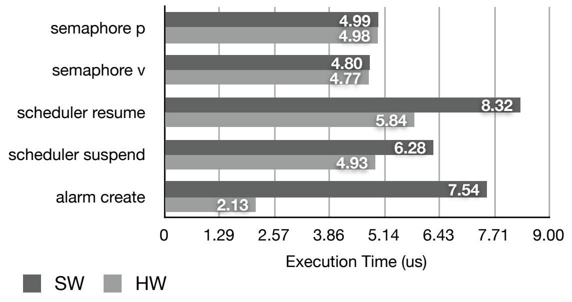


Figure 8: Logic usage and performance of hybrid hw/sw components

In order to evaluate the hybrid component implementations, the "The Dinning Philosophers" application was implemented using the three components. The alarm was used to wake-up the philosophers when their thinking period expired. The figure 8 shows the area consumed by the hardware implementation of the components, and the execution time of some methods of the components on both domains. The consumed logic of the hardware implementations is compared with the *Plasma* processor, an open implementation of the MIPS architecture. The execution times show a acceleration of hardware implementations for the Scheduler and the Alarm component, while the Semaphore component did not gain that much performance from the hardware implementation, mainly because the evaluated application did not push the usage of semaphores queues, whereas a hardware implementation could effectively bring benefits.

5 Conclusions

This paper presented the guidelines to build an architecture of hybrid hw/sw components. It highlighted the importance of the use of an adequate engineering technique in order to design components that are flexible enough to migrate from hardware to software, and vice-versa. Three hybrid hw/sw components that represent all the possible communication behavior of such components was developed, and several experiments were done building a benchmarking application using different combinations of hardware and software implementation of those components. Further research is directed to the migration of hybrid hw/sw components during runtime.

References

[1] 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.

[2] E. Anderson, W. Peck, J. Stevens, J. Agron, F. Baijot, S. Warn, and D. Andrews. Supporting high level language semantics within hardware resident threads. *Field Programmable Logic and Applications, 2007. FPL 2007. International Conference on*, pages 98–103, Aug. 2007.

[3] F. Balarin, Y. Watanabe, H. Hsieh, L. Lavagno, C. Passerone, and A. Sangiovanni-Vincentelli. Metropolis: an integrated electronic system design environment. *Computer*, 36(4):45–52, April 2003.

[4] W.O. Cesario, D. Lyonnard, G. Nicolescu, Y. Paviot, Sungjoo Yoo, A.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.

[5] Krysztof Czarnecki and Ulrich Eisenecker. *Generative Programming: Methods, Tools, and Applications*. Addison-Wesley, 2000.

[6] M.-A. Dziri, W. Cesario, F.R. Wagner, and A.A. Jerraya. Unified component integration flow for multi-processor soc design and validation. In *Design, Automation and Test in Europe Conference and Exhibition, 2004. Proceedings*, volume 2, pages 1132–1137 Vol.2, 2004.

[7] J. Eker, J.W. Janneck, E.A. Lee, Jie Liu, Xiaojun Liu, J. Ludvig, S. Neuendorffer, S. Sachs, and Yuhong Xiong. Taming heterogeneity - the ptolemy approach. *Proceedings of the IEEE*, 91(1):127–144, Jan 2003.

[8] Soonhoi Ha, Sungchan Kim, Choonseung Lee, Youngmin Yi, Seongnam Kwon, and Young-Pyo Joo. Peace: A hardware-software codesign environment for multimedia embedded systems. *ACM Trans. Des. Autom. Electron. Syst.*, 12(3):1–25, 2007.

[9] Paul Pop. Embedded systems design: Optimization challenges. In *CPAIOR*, pages 16–16, 2005.

[10] F. Rincon, J. Barba, F. Moya, F.J. Villanueva, D. Villa, J. Dondo, and J.C. Lopez. Unified inter-communication architecture for systems-on-chip. pages 17–26, May 2007.

[11] Alberto Sangiovanni-Vincentelli, Luca Carloni, Fernando De Bernardinis, and Marco Sgroi. Benefits and challenges for platform-based design. In *DAC '04: Proceedings of the 41st annual conference on Design automation*, pages 409–414, New York, NY, USA, 2004. ACM.

[12] Alberto L. Sangiovanni-Vincentelli and Grant Martin. Platform-based design and software design methodology for embedded systems. *IEEE Design & Test of Computers*, 18(6):23–33, 2001.