OOCE: Object-Oriented Communication Engine for SoC Design

Jesús Barba, Fernando Rincón, Francisco Moya, Félix J. Villanueva,
David Villa, Julio Dondo, Juan C. López
School of Computer Engineering
University of Castilla-La Mancha, Ciudad Real, Spain
Jesus.Barba@uclm.es

Abstract

For decades, software middlewares have tackled with the heterogeneity and interconnectivity roblems in computer networks with success. These problems are recurrent in the design of complex systems-on chip with a large number of components of different nature (including SW and HW modules). In this paper we present an object-oriented communication engine (OOCE) based on the architectural concepts used in software middlewares that unifies the inter-communication interface for both HW and SW elements. The infrastructure provided by the OOCE introduces a low-overhead abstraction layer that can be easily used to implement several parallel programming models. An implementation of this approach has been made for the Xilinx-V2Pro platform.

1. Introduction

The constant increase of the complexity in systems on a single chip (SoC) makes it mandatory the need of new approaches to help the designer to manage such systems. Due to the large number and the heterogeneity of the building blocks that a SoC contains, the election of the appropriate design flow is crucial to achieve a satisfactory result. The interest of the research community seems to move from interface-centric based solutions to high-level programming models proposals applied to system level design. The goodness of such high-level SoC platform programming models has been widely analyzed in the literature. Reutilization of models and components, component interchange, rapid prototyping and fast design space exploration are some of the multiple contributions of these techniques. It is worth to notice that, without the support of tools that makes automatic the design process from system-level specification to synthesis, the success of such solutions would be limited. One of the reasons that has driven this change is the appearance of the multiprocessor System-on-Chip (MPSoC) as the natural evolution in the design style to cope with the so called productivity and platform reuse problems. However, the software-centric design nature [1] of multiprocessor-based systems has shifted the research focus in topics such as concurrency, synchronization, control and effective mapping of applications. The classical hardware component integration and hardware/software interfacing issues seems to be placed in the background.

In this paper, we present a Hw/Sw platform called OOCE (Object Oriented Communication Engine) that supports the implementation of parallel programming models based on communicating distributed objects. Parallel programming models have demonstrated their suitability to be applied in MPSoC design and programming [2,3] whereas object-oriented modeling techniques promote new reuse opportunities through a clear separation between implementation and communication. Our work focuses in the definition and implementation of a lightweight infrastructure that provides an unified, abstract view of the components that conforms modern complex system (including Hw and Sw modules). The OOCE does not impose the use of a particular concurrency or synchronization model, programming model or API between the entities in the system. The designer is free to choice between a wide range of alternatives built on top of the remote method invocation semantics.

Following this introduction, we will discuss related work in section 2. In section 3, the principal features of the distributed object model and how they can be applied to SoC design are presented in depth. Section 4 surveys the proposed communication engine putting the focus on the different communication scenarios and how changes in system configuration are managed by the OOCE. Section 5 will provide a quick view of how different programming models can be easily supported by the OOCE as well as the problem of modelling concurrency. Finally, we conclude the paper with the implementation results and some conclusions.

2. Related work

Lately, there is a significant amount of research work in the area of multiprocessor systems development and Hw/Sw interface modeling. In [4] a unified Hw/Sw component model to describe the different parts of a Hw/Sw interface is presented. It covers different

abstraction levels at different steps of the design flow and it uses a service-based model to automatize the interface implementation. As in [4], we advocate for a unified model to achieve a true system level design methodology where hardware and software do not have different design flows. In our OOCE, the software version of the interface components also follows a two-layer approach. However, unlike [4], the presence of an operating system low-level service layer is optional which may lead to more efficient Sw to Hw communication.

The interface synthesis process proposed in our OOCE is based on the specialization of several interface templates as in [5]. The advantage of such approach in comparison with others is the absence of an interface definition phase and a refinement process avoiding the designer to perform a non negligible amount of work.

TTL [6] is a task level interface that can be used indistinctly for developing parallel application models and as a platform interface for integrating hardware and software tasks. In TTL the system is modelled as a set of communicating task (hardware or software) exchanging vector of tokens of some fixed type. On the contrary, OOCE uses an object-oriented model to specify the solution. Objects exposes a higher abstraction level in the communication since ports are replaced with method calls which may have arbitrary type signatures. These are some of the features that OOCE shared with the Multiflex [3] approach. Another common point between OOCE and Multiflex is the existence of an interface definition language and communication synthesis tools to guarantee a homogeneous representation of the data to be sent/received by the ends of the communication. This assures in-chip inter-component communication semantics. But we go one step forward. The OOCE data type system is 100% compatible with the ICE (*Internet Communication Engine*) data encoding rules. This feature enables OOCE to provide transparent off-chip communication with external components implementing the ICE protocol. To our knowledge, OOCE is the only platform that does not force to rewrite either applications or hardware components in order to make use of this advanced characteristic.

Blocking and non blocking communication is offered like in TTL and Multiflex. In Multiflex, one or more ORBs (*Object Request Brokers*) are responsible for the synchronization and load balancing of the multiple request between objects connected to them. In our view, the ORB concept implemented in Multiflex may turn into a bottleneck in the communication process and can expose scalability and fault tolerance problems due to its centric nature. By contrast, the OOCE ORB equivalent functionality is distributed among the architectural components of the Hw/Sw communication engine.

As in previous works by Paulin et al. [7] we also apply concepts from distributed object middlewares to SoC. We strongly believe that the application of the multiple forms of transparency, that a distributed object model (such as the proposed in Multiflex and OOCE) provides, will help the designers to manage the complexity and heterogeneity problems in complex SoCs. However, there are no clear references in [3,7] regarding this matter. The ORB, as the central functional component, may be considered the responsible for providing some kind of *location transparency* (matching services with requests), *replication transparency* (load balancing of the requests among objects) and *concurrency transparency* (scheduling the requests). But, besides the previously mentioned problems, no considerations are made about the “dynamic view of system”. Questions as how the migration of functionality is treated in a transparent way or even how the location transparency is assured if a service provider change its location (and therefore the ORB which is connected to) does not have a straight response. Our OOCE overcomes these limitations by a set of simple architectural elements.

The distributed heterogeneous programming model that the OOCE supports covers the System, Virtual Architecture and Transaction Accurate levels of abstraction as proposed in [2] to efficiently abstract Hw/Sw interfaces.

A MPSoC is a special case of System-on-Chip with multiple processors. Most of the research work in this area
concentrates on software-related problems such as synchronization between Sw entities running in different processors. The OOCE concepts also apply to this communication scenario and extends the interest to Hw to Hw and and Hw to Sw communication.

The application of object-oriented techniques to the design of hardware systems has mainly found three working lines: (a) the adoption of concepts from high level object-oriented programming languages by hardware description languages and vice versa, (b) system-level design and high level synthesis and (c) the development of hardware architectures that implements software objects.

The improvement of the productivity is achieved by the reuse of previous code making it use of concepts such as class, inheritance, encapsulation and polymorphism. However, there is not a clear correlation between the concept of object and its hardware implementation. This *conceptual gap* is one of the reasons that it has motivated a slow, though progressive, acceptance of these techniques in hardware design.

Regarding system-level design, SystemC provides hardware-oriented constructs packed into a class library, implemented in standard C++, to develop hardware systems. The use of UML for system level design has been proposed in [8,9] while [10] deals with the synthesis process from an object-oriented specification.

Several implementations of software objects as hardware modules have been proposed in the literature. Eventually, a high level specification may be easier to be synthesized into RTL. Radetzki [11] establishes a hardware object architecture based on a memory plus some control logic for state transition. Goudarzi and Hessabi [12] propose a hardware platform for dynamic objects with support for inheritance and polymorphism. Cheng and Wu [13] analyzes four implementation strategies for software objects in hardware based on a data-memory mapping analysis. Our work does not propose any hardware object implementation but a general hardware object model following a structural approach to easily embrace current and future object hardware implementation strategies.

3. The SoC as a distributed object system

During the nineties, the traditional remote procedure call concept evolved into sophisticated software architectures for distributed heterogeneous programming. Many of the goals achieved by these architectures are also main concerns for hardware designers when they face the growing complexity of coming systems. Moreover, an analogy between SoCs and distributed systems can be established: Hw components and software processors as network nodes, buses and NoCs as interconnection media, communication message translated into bus transactions and so on.

Considering a SoC as a set of objects that interact through message passing has a twofold benefit since (a) it keeps the system model invariable and (b) it establishes clear semantics for communication, enabling module interchange and reuse. Ideally, the designer will experience an important increase in his productivity and silicon companies could offer more flexible and cheaper platforms meeting the well-known cost and time-to-market constraints.

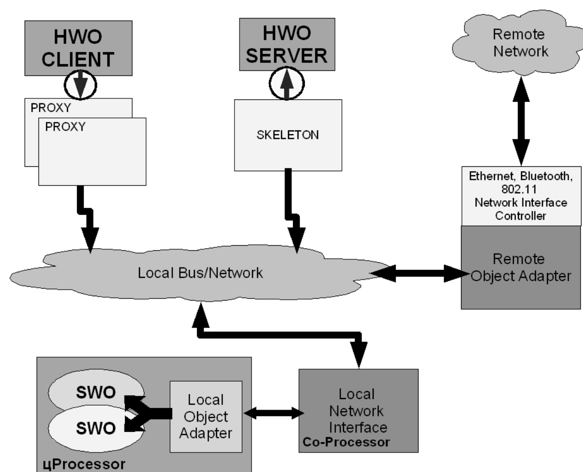

Figure 1. Proposed middleware-based object-oriented communication engine.

Our work is partially inspired in commercial *middlewares* such as ICE and CORBA. We adapt the solutions presented in them to the special requirements of hardware design: efficiency and low overhead. The OOCE implements hardware accelerators to boost the message passing process.

3.1. Middleware concepts for SoC design

A middleware is an abstraction layer containing a set of architectural elements that provide basic communication services. A middleware makes homogeneous the communication between the components hiding the implementation details of the network nodes to the application developer. Generally, a middleware bases its functionality on: (a) a client-server model of communication, (b) a common data type system and a set of data coding/encoding rules and (c) a simple protocol which defines the set of messages that the client and the server exchanges.

The *communication channel transparency* is one of the basis to increase the reuse opportunities in SoC design. Both hardware and software modules can work properly in other environments with minimal changes. A middleware provides another form of transparency: *access transparency.*The interface to a component remains invariable regardless of changes in its internal implementation. SoC design could take advantage of access transparency in many ways:

- *IP integration and interchange*. Classes of hardware components will expose the same module interface (i.e. write and read operations). The implementation decisions (such as the use a FIFO or a RAM to provide some kind of storage capability) can be postponed to the very end of the design process [14].
- *Co-design*. The access transparency principle implies a real homogeneous view of both software and hardware components.
- *Model reuse*. Since all the implementation details are hidden at system level, the model of the system keeps invariant from a design to another.

Location transparency relates to the ability of two components to communicate independently of their real location. The middleware provides to the actors in the communication process the illusion that they interact locally even if they reside in distant network places. This opens a wide range of possibilities in SoC design:

- Transparent Sw to Hw and Hw to Sw communication. The communication mechanism is exactly the same for all the communication scenarios since the system components (Hw or Sw) are not able to distinguish how the source/target of a message is implemented (again Hw or Sw).
- Transparent management of component migration. Functionality may cross software and hardware

boundaries on demand, depending on quality of service and performance criteria.

- Transparent management of component replacement. On module failure or bug detection a component could be substituted. It is not needed to halt the whole system. Reconfigurable logic support is then necessary.

4. The object-oriented communication engine

Figure 1 shows a high-level vision of the most important elements present in our OOCE. Table 1 summarizes the role of each component. As stated previously, our communication engine is object-oriented which implies that any component in the system is seen as an object. An object provides just some per-instance private state data (attributes) and an encapsulation mechanism for some functionality (methods). Objects communicate with other objects in the system invoking the corresponding method.
Therefore, the concept of object is the basis to offer (a) *access transparency* (through the set of methods that compose the object interface) and (b) semantics in the communication (through the method invocation mechanism).

4.1. In-chip (SoC) communication

In our OOCE, each method invocation must take place between a *proxy* and a *skeleton*. Actually, proxies and skeletons are not merely bus/network wrappers that isolate clients and servers from the bus/network. From the client point of view a proxy is seen as a private object implementation. It provides exactly the same interface.

Table 1. Principal OOCE features applied to SoC design.

OOCE	Middleware Concept	SoC application
Proxies/ Skeletons	- Channel access transparency - Component access transparency	- Component reuse - Bus/network independence - IP interchange
Local Object Adapter	- Location transparency	- Transparent Sw and Hw integration - Migration
Local Network Interface	- Channel access transparency	- Bus/network independence - Migration
Remote Object Adapter	- Location transparency	- Off-chip communication

Consequently, the *client* (the object that initiates the communication) starts the invocation of a method as if it was physically connected to the real server. The proxy deals with the particularities of the underlying communication channel and forwards the request following a particular protocol and message format. In the other side, the skeleton acts as the server (the object that implements the required service). It receives the request and interprets it. Finally, the actual method invocation is performed to the actual server.
A method call is decomposed into write and read primitives. Read and write operations are basic services offered by any of the buses and on-chip networks, so that the complete process can be easily targeted to a particular bus/network technology (OCP, OPB, AMBA, etc.). Also, some data encoding/decoding rules have to be defined in order to homogenize how data types are sent "on the wire". Our OOCE provides support for synchronous invocations (blocking) and asynchronous invocations (non-blocking) for high latency operations.

4.1.1. Hw to Hw communication. To implement logical objects as physical hardware components, we define: (1) a standardized interface in order to make automatic the generation of wrappers and (2) a "local" method invocation protocol. By "local" we mean two components connected directly. A hardware object exhibits a characteristic physical interface. One of the main features of our hardware object model is the flexibility to define how values are passed/retrieved to/from the IP. This makes it easier to fit the final implementation to the particular design constraints or to adapt existing IPs.
However, "local" invocations are not the main scenario in current SoC: hardware components communicate through a system bus or network. Proxies and skeletons are in charge of implementing the "remote" invocation of methods as described in section 4.1. By "remote", we mean an invocation that takes place between two hardware objects connected to a system bus or network. We propose an interface synthesis process based on object-oriented models to generate proxies and skeletons in an automatic way [15].
We use UML (*Unified Modelling Language*) to model the static and dynamic view of the system. For each object, a language-neutral interface definition using SLICE (The interface definition language used in ICE) is derived. Before the generation of RTL code, the designer must select a particular scheme of communication (bus, network, etc.) from a set of predefined mechanisms. Finally, a wrapper generation process specializes and optimizes a wrapper template that depends on the communication method previously selected, the number and type of operations that it must support, etc. The last step in our interface synthesis process is the generation of the hardware version for proxies and skeletons. We have defined two interface templates which will be specialized

iterator.cpp

```
iter::RGB operator *() {
  RGB rgb;
  int data;
 // synchronous invocation
 // number of parameters and model
  putfsl(0<<16||MSYNC,M_FSL);
  putfsl(ITEROBJ_ID,M_FSL);
  putfsl(ITEROBJ_GET,M_FSL);
  // retrieve result
  getfsl(data,S_FSL);
   //unpack values, 3 8-bit compo-
nents
  rgb.r = data & 0x00FF0000 >> 16;
  rgb.g = data & 0x0000FF00 >> 8;
  rgb.b = data & 0x000000FF;
  return rgb;
}
```

filter.cpp

```
filter::void run() {
   ...
   current_pixel = *it;
   ...
}
```

main.cpp

```
int main() {
   filter f;
   f.run();
   return 0;
}
```

(a) SW version

filter.cpp

```
filter:: void run {
  putfsl(0<<16||MSYNC,S_FSL);
  putfsl(FIITEROBJ_ID,S_FSL);
  putfsl(FITER_RUN,S_FSL);
}
```

main.cpp

```
int main() {
   filter f;
   f.run();
   return 0;
}
```

(b) HW version

Figure 2. Mixed mode filtering implementation.

according to the bus/network selected by the designer and the number and kind of methods to be interpreted.

Based on the relationship information provided by the UML model, a skeleton is produced for each hardware object exporting at least one method. Only for each method that has to be remotely invoked by the client, a proxy to the target is generated. This reduces the logic used in the proxy since there is no implementation for unused methods. Another optimization introduced in this process is the reutilization of logic when two method definitions and their corresponding physical hardware object interfaces are identical.

4.1.2 Hw to Sw and Sw to Hw Communication. In order to perform Hw/Sw communication transparently, neither software objects nor hardware objects must change the way they interface depending on which is the target/source of a method invocation. Proxies and skeletons, in the hardware side, do not need to distinguish if the callee/calling object is running in software or not. For this reason, invocations involving hardware and software objects result in exactly the same messages than the ones generated in a hardware to hardware method call.

The Local Network Interface (LNI) and the *Local Object Adapter* (LOA) are the OOCE components that are in charge of the management of Hw/Sw communication. The LNI is completely implemented in hardware as a co-processor which is also a peripheral connected to the local bus/network. Usually, the processor has a limited control over advanced bus features such as burst transactions, error control, etc. To implement an efficient Sw to Hw communication all the communication capabilities must be exposed and accessible.

Also, the system processor usually has only a master interface to the system bus. This restricts the way other peripherals can communicate with it, for example through an interruption mechanism. The LNI simulates the processor slave interface we need to remain invariable the behaviour of the hardware proxies if the destination of the communication is a object running in the processor.

Figure 2 shows the automatically generated software proxies code for two versions of a simple filtering algorithm implemented in a Xilinx-V2 Pro Platform . In (a) the iterator is implemented in hardware [14] whereas in (b) the filter and the iterator are both implemented in hardware. The reader should notice that although the non-negligible changes in the underlying platform, the implementation of the main program does not change.

The LNI holds a table containing only the identifiers of he objects and theirs corresponding system base address. The LNI is continuously monitoring the bus/network traffic. Once the LNI detects a transaction addresses to a software object, it issues an interruption. The LOA (implemented as the interruption service routine) communicates with the coprocessor to get the data associated with the invocation. Finally, the LOA acts as the skeleton of the server object and invokes the software method.

The migration of functionality from Hw to Sw only requires the addition of a new entry in the LNI object's table. In addition, the Hw object module has to be notified so that any bus/network transaction is now unattended. The reverse path (migration from Sw to Hw) comprises the opposites steps.

4.2. Off-chip (SoC) communication

Transparent off-chip communication is one of the main contributions of this work. In other approaches, since a common communication infrastructure is missing, on-chip functionality may only be accessed from off-chip components using an ad-hoc interface that exists only if it has been foreseen by the designer.

The *Remote Object Adapter* (ROA) receives the invocation from a external client object as a TCP encapsulated ICE (the commercial middleware used in our prototype) message, and translates it into a message for the skeleton as if it was a local proxy. The ROA internally caches some parts of the ICE message (client external network address, client OID, request identifier) in order to build a valid response message. Invocations from inside the chip to an outer server object are also possible. The ROA maintains a table with the OIDs of the external accessible objects and remote network addressing information.

The number and type of applications that can make use of this feature ranges from debugging, simulation and

verification to collaborative development tools, remote control and ubiquitous computing.

4.3. Dynamic Reconfiguration Support in OOCE

The OOCE implements a reconfiguration service to manage the use of dynamic reconfiguration logic from an object-oriented point of view [16]. The presence of the reconfiguration logic will be unavoidable if we want to provide our system with migration and component replacement transparency. The use of the concept of object, easies the utilization of the partial reconfiguration technology by the applications. The reconfiguration service is logically structured in a four layer stack:

- *Layer 1. Dynamically reconfigurable objects.* At the lower level we find the dynamically reconfigurable objects. These objects, in addition to their normal functionality, include a very simple extra interface for controlling the stop, activation, the reconfiguration or the storage or loading of their state.
- *Layer 2: Hardware and Software Activators.* The activation layer manages the whole process of reconfiguration for an object, which may include making the object persistent.
- *Layer 3: Dynamic Objects Management.* The third layer provides high-level reconfiguration services, such as the scheduling, the location or the migration of the dynamically reconfigurable objects. The last layer corresponds to the application or operating system, that can make use of the reconfiguration services implicitly or explicitly.

5. OOCE programming models and concurrency support

As discussed in section 2, the OOCE is mainly based on a message-passing model typical of heterogeneous distributed computing environments. However the OOCE does not limit the choice to a single programming model. Due to the low-overhead introduced by the OOCE platform and the high degree of efficiency achieved by the hardware accelerators, more alternatives can be built on top of the communication engine.

For example, let us consider a shared memory communication framework. The TTL interface [6] can be considered a good example of this kind of inter operation. Tasks correspond with objects in the OOCE (to be more precise *control or active objects*). Channels can be also modelled as objects in our object-based platform. Each object will be responsible for the channel administration and will implement the necessary methods to support TTL basic primitives interface. Different interface types (CB, RB, RN, DBI and DNI in TTL terminology) will be available through specialization of the channel base class.

The management of the resources and the concurrent aspects of the system must be taken into account in parallel execution environments such as MPSoC, with several tasks or objects running simultaneously on different software processors. Once again, the use of the OOCE does not impose a fixed interface to get all the system components synchronized. What the OOCE provides is just a set of basic communication services that can be used at higher abstraction levels. Within the OOCE framework, the synchronization and concurrency problems are considered design problems. This contrasts with other interface-centric approaches where these matters have an important impact on the final API limiting the alternatives presented to the designers.

We have developed a library of hardware, software and mixed (Hw/Sw) components to incorporate concurrency administration to the OOCE. The OOCE concurrency and synchronization support relies on the use of these library components (mutexes, locks, semaphores etc.) that enable the implementation of concurrency and resource design patterns.

6. Experimental results

All the OOCE concepts presented in this work have been implemented on the Xilinx XUP-V2Pro platform. The XUP-V2Pro platform uses a OPB system bus to connect all the processing elements in the SoC working at 100Mhz. Software objects will run on a Microblaze software processor just as the local object adapter. The LNI has been implemented as a coprocessor attached to the system processor using two FSL (*Fast Serial Link*) interfaces. The slave link is used to invoke methods from SW to HW and the master link can be used by: (a) a calling SW object to retrieve the result from a previous synchronous invocation or (b) the LOA component to manage HW to SW invocations or to retrieve the result from a previous asynchronous invocation.

The presence of the LNI component reduces considerably the overhead for a Sw/Hw, Hw/Sw and Sw/Sw (invocations between objects running on different processors) communication. Both client-side and server-side code (software version of the proxy and skeleton) generated by our tools take less than 20 instructions each. The delay introduced by the LNI component is of 6 cycles for a incoming invocation and 3 cycles for a outgoing invocation (the former needs to translate the target bus address).

The ROA is also completely implemented in hardware which provides faster processing times of ICEP messages. Less than 90 microseconds are necessary to parse an external invocation including: Ethernet, TCP and ICE header checking to validate the request, translation of the object and method identification strings to internal bus addresses. This represents a reduction of two orders of magnitude if we compare with its counterpart in software.

Table 2. OOCE components synthesis results.

	ROA	LNI	Proxy	Skeleton
Slices	912	30	25	24
FF	825	113	6	0
LUTs	2161	167	27	43
Critical path (ns)	6.615	7.140	1.07	2.65

Table 2 shows the synthesis results for the OOCE components implemented in hardware. Both ROA and LNI figures do not included the resources regarding the implementation of the translation tables.

7. Conclusions and future work

We have described an implementation strategy of a lightweight communication infrastructure for systems that are modelled as communicating objects. This infrastructure is independent of the underlying bus/network that relates all the components in the system. We have implemented all the presented concepts in a FPGA-based system. The principal features of our OOCE are: (a) most of its components are generated in an automatic way, with a minimum participation of the designer, (b) it is flexible since it is extremely easy to adapt it to new target technologies and (c) it enables the use of parallel programming models that help to abstract both hardware and software interfaces.Future work is being focused on developing a real co-design object-oriented design methodology thanks to the introduction of concepts such as locations transparency and access transparency.

8. Acknowledgement

This work has been funded by the Spanish Ministry of Education and Science (TIN2005-08719) and the Regional Government of Castilla-La Mancha (PBI-05-0049).

9. References

[1] Grant Martin, "Overview of the MPSoC Design Challenge", In Proc. of the *43th Design Automation Conference,* San Francisco, California, 2006.

[2] A.A. Jerraya, A. Bouchhima, F. Pétrot, "Programming models and Hw-Sw interfaces abstraction for Multi-Processor SoC", In Proc. of the *43th Design Automation Conference*, San Francisco, California, 2006.

[3] P.G. Paulin et al. *Parallel Programming Models for a Multiprocessor SoC Platform Applied to Networking and Multimedia*, IEEE Transactions on VLSI systems, vol. 14, 17, July 2006.

[4] A. Bouchhima et al., *A unified Hw/Sw interface model to remove discontinuities between Hw and Sw design*, In Proc. of EMSOFT 2005, Jersey City NJ USA, September 2005.

[5] Yin-Tsung Hwang and Sung-Chung Lin. Automatic protocol translation and template based interface synthesis for IP reuse in SoC. In *Proc. Of the 2004 IEEE Asia-Pacific Conference on Circuits and Systems*, 2004.

[6] P. Van de Wolf, E. De Kock, T. Hendrikson, W. Kruijtzer, and G. Essink, "*Design and Programming of Emebedded Multiprocessors: An Interface-centric approach*". In Proc. of the *CODES+ISS'04*, September 2004.

[7] P.G. Paulin et al. *Distributed Object Models for Multi-Processor SoC's, with Application to Low-Power Multimedia Wireless Systems*. In Proc. of Design Automation Conference, Mar 2006.

[8] R. Chen, M. Sgroi, L. Lavagno, G. Martin, A. Sangiovanni-Vicentelli, and J. Rabaey. Embedded system design using UML and platforms. In *Forum on Specification and Design Languages* (FDL), Marseille, France, September 2002.

[9] Q. Zhu, A. Matsuda, S. Kuwamura, T. Nakata, and M. Shoji. An object-oriented design process for system-on-chip using UML. In *International Symposium on System Synthesis* (ISSS), pages 249-254, Kyoto, Japan, September 2002.

[10] C. Schulz-Key, M. Winterholer, T. Schweizer, T. Kuhn, and W. Rossenstiel. Object-oriented modeling and synthesis of system specifications. In *Asia South Pacific Design Automation Conference* (ASPDAC), pages 238-243, Yokohama, Japan, 2004.

[11] M. Radetzki. Synthesis of digital circuits from object-oriented specifications, *Ph.D. Thesis*, Univerisität Oldenburg, Germany, 2000.

[12] M. Goudarzi, and S. Hessabi, Synthesis of Object-Oriented Descriptions Modeled at Functional-Level. *World Scientific and Engineering Academy and Societ Transactions on Computers*, Athens, 2003.

[13] Fu-Chiung Cheng, Hung-Chi Wu. Design and Implementation of Software Objects in Hardware. In *International Conference on Computer Design* (ICCD), Oct 2006.

[14] F. Rincón, J. Barba and J.C. López, "Generic Programming with Abstract Parametrized Components", In Proc. Of the *DCIS Conference*, Bourdeaux, France, 2004.

[15] J. Barba, F. Rincón, F. Moya, F.J. Villanueva, D. Villa, and J.C. López, "Lightweight Communication Infrastructure for IP integration", In Proc. Of the IPSOC Conference, Grenoble, France, December 2006.

[16] J. Dondo, F. Rincón, J. Barba, F.Moya, F.J. Villanueva, D. Villa, and J.C. López, "Dynamic reconfiguration management based on a distributed object model", In Proc. Of the *17th FPL International Conference*, Amsterdam, Netherlands, August 2007.