Unified Inter-Communication Architecture for Systems-on-Chip

F. Rincón, J. Barba, F.Moya, F.J. Villanueva, D. Villa, J. Dondo, J.C. López
University of Castilla-La Mancha
fernando.rincon@uclm.es

Abstract

System-On-Chip (SoC) architectures are called to be the platform for an ever increasing number of interactive applications. One of the most time-consuming tasks is to define communication interfaces between the different components through a number of scattered heterogeneous processing nodes. That is not only a complex task, but also very specific to a certain implementation, which may limit the flexibility of the system, and makes the solutions difficult to reuse. In this paper, we describe how the distributed systems paradigm can be extended to provide a unified abstraction for both hardware and software components. Moreover, based on that abstraction, we define a low-overhead system-wide communication architecture that offers communication transparency between all kinds of components. Since the architecture is highly compatible with standard distributed object software systems, it also allows seamless interaction with any other kind of external network.

1. Introduction

The latest consumer applications (e.g. multimedia processing or 3D games) demand complex designs to meet their real-time requirements while respecting other design constraints, such as low power or short time-to-market [10]. In this context, SYSTEMS ON CHIP (MPSoCs) have been proposed as a promising solution. Nevertheless, one major challenge in such systems is the integration in the platform of the multiple communication and APPLICATION PROGRAMMING INTERFACES (API) that each component (e.g. memory, buses, cores, etc.) is designed for. Moreover, another important problem in SoC design is the knowledge of the position of each component in the final system so as to be able to efficiently communicate with it (i.e. is it local, remote etc.), which makes the correct design of the aforementioned SoC an even more complex task. Thus, new methods that allow designers to achieve unified inter-communication methods on SoC architectures in the system integration flow are urgently needed.

Some concepts taken from distributed object platforms such as CORBA or Java RMI have already been applied to SoC design in order to obtain a unified view of Hw and Sw modules [13]. In this paper we present an approach which inherits most of these previous achievements enriched with a strong focus on location transparency and network transparency. The resulting architecture provides a unified view of the whole system and also enables the designer to seamlessly develop multi-SoC systems with different network technologies.

This paper is organized as follows. In Section 2 we discuss why SoCs are valid platforms for distributed objects systems. In Section 3 we present the overall hardware SoC architecture proposed for an efficient unified interconnection of components. In Section 4 we describe the synthesis process of the communication architecture. Section 5 provides an overall illustrative example. In Section 6, we describe different SoCs used as case studies and present our experimental results. In Section 7, we give an overview of work relating to our approach . Finally, in section 8, we summarize the contributions of the paper and present possible directions for future research.

2. SoC as a Distributed Object System

Generally speaking, a distributed system consists of a set of heterogeneous computing and storage resources connected to each other via an interconnected network. This normally refers to computers interconnected through local or wide area networks, but the definition may also include other kinds of systems, using different types of networks, transport protocols, etc.

Most of these systems are programmed using an object-oriented approach. The use of objects enforces modularity and provides flexibility and reusability. Objects also rely on a simple communication model: method invocation. This same mechanism is used for remote communication (REMOTE METHOD INVOCATION or RMI), where invocations are translated into synchronous messages passed though a certain communication infrastructure. The main advantage of RMI is that it provides a neat separation between functionality and communication. That makes Distributed Object Systems specially well-suited to deal with heterogeneity and scalability of applications.

But communication in distributed systems means more than simply routing messages between remote objects. It requires that we handle concepts such as access transparency, where an object must provide the same interface for an object regardless of its target architecture or language implementation; location transparency, which means that the destination of the message is not explicitly known by the caller; or migration transparency, where an object may migrate to another node without the caller knowing it or requiring any special action for it. These concepts are all built into a common infrastructure, the middleware, which will handle all communication tasks, and will provide a common high-level abstraction to the communicating objects.

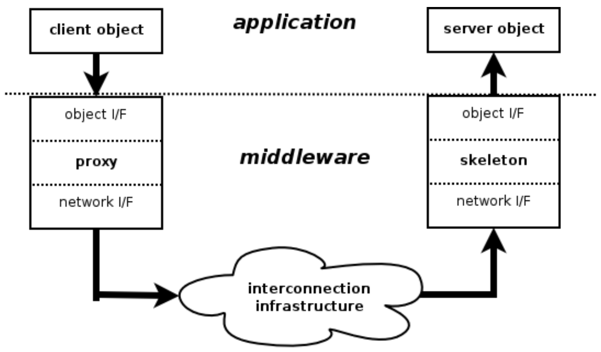

Figure 1. Actors in a Remote Method Invocation

Figure 1 shows the actors in remote communication, and the domain they belong to (application or middleware). Any method invocation must take place between a Proxy and a Skeleton. From the client point of view, the proxy is the requested object itself, since it provides exactly the same physical interface. On the other hand, server objects do not need to care about the location of client objects. They just provide an object interface which is exported through a skeleton. Communication between clients and servers then flows in the following way: 1) client requests an operation from the local proxy of the object. 2) The proxy translates the request into a set of signal assignments that will depend on the underlying communication architecture, 3) The skeleton receives proxy requests and translates them back to the operation provided by the server, 4) the server completes operation execution.

As in many standard software middlewares the approach described above relies on the automatic generation of the proper proxies and skeletons depending on the kind of communication that must be established between objects.

2.1 Global Communication Mechanism

The Distributed Object paradigm can be extended to multiprocessor NoCs. Messages passing between objects can naturally be adapted to the packet switching transmission mechanism of embedded networks, while software distributed objects may be deployed over the different software processors in the NoC, simply by adapting the embedded versions of standard middlewares.

Any hardware component can also be considered a hardware object. Such a component doesn’t have exactly the same characteristics as its software counterparts, mainly due to the static nature of hardware. However, they have a well known interface, provide encapsulation, have a state and include the logic to perform computations over that state, which is enough for using them in a distributed objects environment.

Here the challenge consists of how to define a global communication mechanism (system-wide middleware) that is able to provide communication transparency between any objects in the system, and support a unified programming model through the use of the same abstractions, regardless of the hardware or software implementation of the objects.

3. SoC Hardware Inter-Communication Architecture

In the following paragraphs, we describe our proposal for a system middleware that has been conceived with 3 objectives in mind. First, to define the basic communication mechanisms to allow RMI between any node inside the SoC, in spite of their hardware or software implementations. Second, it must be interoperable with other middlewares. That means that it must give external objects of the SoC the possibility to invoke methods from internal objects, and vice versa. And finally, it must be light-weight in cost and performance.

3.1 Local Inter-Communication Mechanism

One of the duties of the communication system is to cover the gap between high-level object method invocation and low-level protocol signaling of NoC network interfaces or standard buses. Invocations represent messages with an object and method name, plus a certain number of parameters of simple or complex types that may or may not have a return value. Low-level interfaces deal with data words that can be securely routed from/to other NIs or transmitted through a bus. To bridge this gap, the communication system defines a standard message format and a transport independent message passing mechanism (the proxies and skeletons described in section 2).

But the communication system must also provide interoperability between hardware and software objects. That means that proxies must be able to communicate to their corresponding skeletons, no matter what their mutual hardware or software implementations. Even the same proxy might communicate sometimes to a software implementation, sometimes to a hardware one of the same object. In the proposed architecture, all objects are interconnected to a common physical communication infrastructure (a bus, a hierarchy of buses or a NoC). So, interoperability is guaranteed by translating equal invocations to equal low-level messages. There is absolutely no difference then between an invocation message generated by a hardware or software object.

Hardware objects implement their functional interfaces using signals for method invocation and buses for their parameters and return values. Thus, Hw to Hw communication requires that the proxy pack the invocation into a set of words and translate it into a read or write transaction on the bus or NI (normally a burst). On the other side the skeleton unpacks the received data and regenerates the signals for exactly the same signals interface, giving the illusion of a point to point connection for both objects. In fact, the only difference between both approaches is the time the invocation takes to reach the destination. It may seem that proxies and skeletons are then simply bus wrappers between two cores. In part they are, since they provide bus adaptation, but they also provide the semantics of RMI, so there is no need to know anything about an object except its logical interface to use it. That is not the case of current IP cores, where communication and functionality are not clearly separated. Moreover, the overheads of this approach are almost negligible, as will be discussed in section 5.

From a distributed object environment point of view, we may say that the main difference between hardware and software objects is that while the former execute in their own node, the latter share the node (the processor) with many other objects. For that reason objects are not directly connected to the communication system, but a software layer exists between them. This layer may be as complicated as an operating system, or simply be a collection of routines that provides secure (and multiplexed) access to the system bus or network interface. We will refer to this layer as the communication API. In any case, the messages routed from and to software objects through the low-level communication infrastructure are exactly the same as if they were routed to their hardware versions.

Invocations from software to hardware objects use again the same flow described before. There is a local software proxy which provides the same interface as the hardware object. The proxy is almost hollow, and simply translates each invocation to a remote invocation, using the communications API (see RMI invocation in figure 4a). The invocation is serialized by the communication software and converted into a low-level transaction, identical to that generated from the hardware version of the proxy. Hardware to software invocations are symmetrical from the logical point of view, and messages reaching software skeletons are translated into local invocations.

The sceneries described before show how proxies and skeletons are used to provide communication transparency between all kinds of object (hardware or software). However, there are some other facilities that can be easily provided by the proposed architecture, such as location transparency. In that case a specialized object (locator) includes a routing table, and the hard-coded addresses of the target objects in the proxies are replaced by an indirect invocation through the locator object. Of course, this has some negative effects on communication performance, however location transparency should be compelling to support dynamic object creation in reconfigurable architectures, or for patching buggy hardware after manufacturing, for example.

3.2 Remote Inter-Communication Mechanism

The proposed architecture also considers remote communication with objects external to the SoC, and that can be accessed through an ethernet network interface, for example. Two special services provide remote invocation capabilities to and from the SoC, but no modifications are necessary for any of the local objects involved. Invocation from external object methods is handled by the *object adapter*. This adapter includes the proxies for those local objects that are going to be accessed from the outside. It is also responsible for receiving the messages from the outside (a UDP frame from an ethernet controller, for example), and translating them into invocations of the equivalent local proxy. On the other hand, invocation of external objects methods is the responsibility of the *remote server*. The server provides local skeletons for the caller proxies of all the remote objects to be contacted, translates invocations received by every proxy to the corresponding message, and routes it accordingly. Since not all objects in the system will require remote interaction, those two services may be optimally generated for the needs of the application.

4. Unified Design Flow

One of the main features of the proposed global communication architecture is that it can be fully automatically generated. This prevents the designer from dealing with complicated APIs or concrete communication protocols, and eases design exploration tasks.

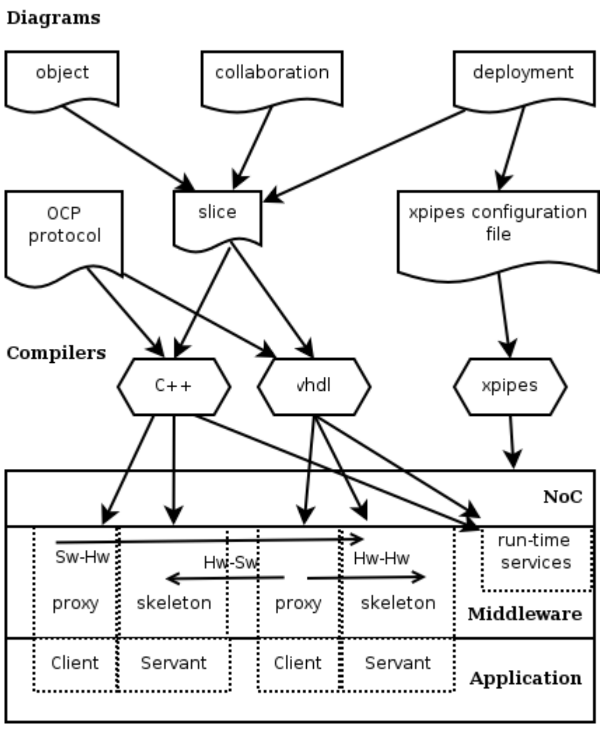

Figure 2. Communication Architecture Synthesis Flow

The synthesis flow is depicted in figure 2. Initially we should model the application with an object oriented approach. This model will only define a set of entities (objects) and relationships (collaborations) in the domain of the problem, and thus will not be related to any kind of implementation.

The inputs to the flow will be the object, collaboration and deployment diagrams that are typically obtained using UML for the modeling phase. The first one describes the interaction between objects (of a certain class) at run time. The collaboration diagram describes interaction between objects in terms of sequenced messages for concrete use cases. Finally, the deployment diagram should define how objects are to be implemented (Hw or Sw), and even additional information for Hw objects such as the relationship between parameters and ports, the number of concurrent requests a method may support, etc. All these 3 diagrams will be used to generate a single slice file. Slice is an Interface Description Language (IDL) for the Internet Communication Protocol [8] (ICE), a CORBA-like middleware platform. The slice description will contain user-defined data types and the object interfaces with only those methods that are really used in the application. The file will also contain some metadata to capture the information in the deployment diagram. The deployment diagram will also be used to generate a configuration file for the synthesis of the low-level communication infrastructure. That will be the XPipes compiler [9] in our case, and it will define the number of processing elements in the system and possible routes between them. The Slice description is still implementation technology (Hw or Sw) and language independent.

The synthesis of the communication architecture is performed by three kinds of compilers. The XPipes compiler will synthesize the lower layer of the NoC, that will provide message transport. The Slice-to-C++ and Slice-to-VHDL are responsible for the generation of the second layer, the system middleware, which is in fact a collection of proxies and skeletons of the different objects plus additional logic (and code) to provide run-time communication services, such as a directory service for indirect location of the objects, remote communication outside the NoC, etc. On the application domain, client and server objects will simply connect (or instantiate in the Sw domain) to the corresponding proxies and skeletons.

In order to obtain a light-weight implementation of the middleware, every component is optimally synthesized. That means that only the necessary hardware or software versions of proxies and skeletons are generated (depending on the type of invocation: Hw-Hw, Hw-Sw or Sw-Hw). Even for the internal architecture of these components, the interface, data type marshalling logic and temporary data storage is only included for the methods explicitly used in the collaboration diagrams. That makes it possible to reuse components with more functionality than needed by the application, but without paying an extra cost in the communication architecture for it. The same policy applies to run-time services such as indirect invocation or remote (to the outside of the NoC) communication. The Object adapter, for example, will only include the logic for a message passing between those objects marked as invocable from outside the NoC, while the Remote adapter will provide remote proxies for the remote objects really used from the NoC.

5. Illustrative Example

We will illustrate the concepts discussed previously with a little image filtering application. Despite the simplicity of the example, it is straightforward to extend it to much more complex systems.

There are only three objects involved: the one requiring the filtering, the filter object itself and an iterator object used by the latter in order to get the pixels of a certain image. The collaboration diagram (figure 3) captures the sequence of invocations between the objects.

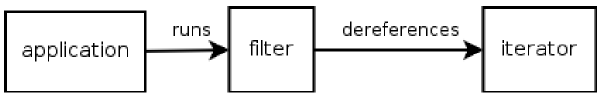

Figure 3. Collaboration diagram of the example

We will consider three versions of the application: 1) purely software, 2) mixed, with a hardware iterator and 3) mixed with a hardware filter and iterator. The first of them does not require the services of the communication system defined, since all objects are local to the same processor, so we will concentrate the description on the last two.

Figure 4 shows a piece of code for each of the two mixed-mode implementations. In 4a, the iterator is implemented in hardware, so the software iterator class in the listing is in fact the proxy view of the hardware object. The invocation of the indirect operator (*) of the iterator results in an RMI call from the communication API provided as part of the middleware. But from the caller point of view (the filter object) there is no difference from using the proxy to the hardware iterator from a real software iterator. They have exactly the same interface.

In figure 4b the software filter has also been implemented in hardware. For that reason, the code in the run method has been replaced by another RMI call. The software filter is then again a proxy to the hardware real object. The filter still uses the iterator to get the data of the image, but now it uses a hardware version of the iterator proxy. Invocations from the hardware filter to the indirect operator of the iterator result in exactly the same messages as the ones generated by the software proxy in figure 4a.

Iterator.cpp (iterator proxy)

```
Iter::RGB operator *() {
   RGB rgb;
   RMI(ITEROBJ_ID,
          DEREF_ID,
         &rgb,null);
   return rgb;
}
  …
```

filter.cpp (Sw object)

```
filter::void run() {
   …
   pixel = *it;
   …
}
```

filter.cpp (filter proxy)

```
filter::void run() {
   RMI(FILTEROBJ_ID,
       RUN_ID,
       void, null);
}
```

main.cpp

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

main.cpp

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

Figure 4. Mixed mode filtering implementation

It is worth noting in this example that, from the main program point of view, there is no way to distinguish if the filter object is implemented in hardware or software, since both provide exactly the same interface, and therefore are interchangeable. Also, Sw-Sw invocations are normal procedure calls that take place inside the processor. On the other hand, Sw-Hw invocations can be divided into a Sw-Sw invocation from the caller to the proxy of the callee, and a Hw-Hw invocation from the proxy to the callee. The first one belongs to the application domain, while the second one is performed by the communication middleware. Finally, Hw distributed objects (or servants) can be simultaneously and transparently invoked from different objects.

6. Experimental Results

The SoC described on the previous section has been implemented on the Xilinx XUP-V2Pro platform, following the approach in section 3.1. Each of the main programs has been assigned to a microblaze software processor as a stand-alone application, while hardware versions of the filter and the iterator objects have been synthesized from a VHDL description. We have also tested two different transport infrastructures: 1) the OPB bus which is available on the board, 2) a 3-

switch NoC synthesized from an xpipes description. From this experiment we can conclude that: 1) the effort for migrating a software object to a hardware implementation is only related to the coding of the functionality of the object. No changes to the rest of the objects are required, but a regeneration of the proxies of the new object is necessary. 2) The transport architecture can also be replaced transparently. To do this it is only necessary to resynthesize the corresponding proxies and skeletons using another transport protocol. 3) This change, however, will affect the performance of the communication. As shown in table 1, for example, for such a simple test case, the bus is more appropriate than the NoC, since the latency of the messages is considerably lower for the same kind of generated adapters. That will not be the case for typical SoCs, where the number of objects and communicating threads will be more suited to the NoC transport alternative.

Table 1. Execution time and message latencies

Latencies (cycles)	Total	Per Message
Sw	170 + 338 * #pixels	-
Hw it / Bus	170 + 298 * #pixels	2
Hw filter / Bus	170 + 12 * #pixels	2
Hw it / NoC	170 + 328 * #pixels	32
Hw filter / NoC	170 + 42 * #pixels	32

In order to evaluate the overhead of the communication architecture, we have defined the simplest possible situation, where a very simple hardware object with an internal register with two methods (1) set_value , 2) get_value), is accessed from another client hardware object. The objects have been deployed over the XUP-V2Pro board, following the flow in section 4. Alternatively,, we have repeated the implementation, keeping the same functionality on both objects, but this time using the IPIF wrapper generator from Xilinx tools. The synthesis results are shown in table 2. Each pair of columns corresponds to one block: the data in the first one is related to the whole component (in terms of FPGA resources consumed and critical path), while the second relates only to the functional part of the block (without the skeleton or proxy logic). The numbers show how the resources consumed in both approaches keep almost constant for the core functionality of the blocks. However, there is an important deviation when considering the whole block, specially for the client. This is due to the use of the IPIF parametric core as the base of the OPB wrappers. Although the IPIF core is supposed to be highly optimized, it is clear from the numbers that a specifically customized proxy can be far more efficient, since it only includes the logic that it really needs. This difference should decrease as the core requires more advanced communication services (such as DMA, interrupts), but it gives a hint on the adaptability of the proposed approach.

Remote inter-communication has also been experimentally verified. In this case the same get/set object from the second experiment has been invoked from an external client object running ICE (a commercial middleware) on a PC. Thus, an object adapter has been generated for the get/set object. The adapter receives the invocation from the client as a TCP encapsulated ICE message, and translates it into a message for the skeleton of the get/set, as if it was a local proxy. Currently, the adapter has been prototyped in software using the embedded version of ICE compiled on top of a microblaze processor. That provided automatic interoperability with the client, and made use of the software TCP stack.

Table 2. Proxy and Skeleton area results

	IPIF client	Proxy	IPIF server	Proxy
Slices	208	5	59	5
FF	189	13	97	2
LUTs	346	8	18	13
Critical path	0.37	0.01	3.25	2.65

7. Related Work

The ideas presented in this paper complement previous work on system-level abstractions. Orthogonalization of concerns in system-level design as proposed by [11], and more recently by [1] and [5], provide an object model similar to what this paper assumes, but most actual implementations focus on a structural view of the system and do not consider location transparency. In [12] a uniform communication mechanism for Hw and Sw resources is proposed, based on a central Hw-Sw OS and a Hw abstraction layer to provide task abstractions for Hw components. Previous work by Paulin et al [13] already applies concepts from distributed object middlewares to SoCs but they do not even consider one of the key features, location transparency. Some early ideas on how reconfigurable computing may benefit from these concepts are found in [7]. Previous results on automated generation of communication infrastructure for SoC design in [3, 4] are also applicable to the proxy/skeleton generation.

Object based and object oriented approaches [6, 2, 14] have also been used extensively to reduce the effort of translating some software components into hardware components or to improve the co-simulation of the system. Our hardware objects require a subset of what is provided by these extensions. Therefore we remain compatible with their approaches and we also keep full compatibility with standard IP based methodologies.

8. Conclusions

The communication architecture presented in this paper extends the distributed object paradigm to SoC platforms. The proxy and skeleton abstractions plus the use of the RMI semantics, provide a simple way to uncouple component functionality from communication implementation. From the designer perspective, this provides a homogeneous view of the system as a collection of communicating objects. From the point of view of implementation, the model presented provides communication and location transparency for any kind of local interaction between hardware and software components, blurring the hardware and software interface barrier. But it also provides the possibility of remote (may be off-chip) interaction with other objects.

Moreover, all the services and components that make up the middleware can automatically be generated, based on a few descriptions of? the interfaces of the objects, and on the deployment over a certain platform. This enhances the possibility of future reuse and eases design space exploration tasks. And, as the experimental results show, the communication architecture does not incur high overheads.

9. Acknowledgement

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

10. References

[1] W. Cesario, L. Gauthier, D. Lyonnard, G. Nicolescu, and A.A. Jerraya, "Object based hardware/software component interconnection model for interface design in system-on-a-chip circuits", *The Journal of Systems and Software*, 2004.

[2] R. Damasevicius and V. Stuikys, "Application of the object-oriented principles for hardware and embedded system design", *Integration, the VLSI Journal*, 2004.

[3] V. D'silva, S. Ramesh, and A Sowmya, "Bridge over troubled wrappers: Automated interface synthesis", *In Proceedings of the Intl. Conf. on VLSI Design*, 2004.

[4] A. Gerstlauer, "Communication abstractions for system-level design and synthesis", *Technical Report CECS-TR-03-30*, UC Irvine, 2003.

[5] A. Gerstlauer, D. Shin, R. Dmer, and D. D. Gajski, "System-level communication modeling for Network-on-Chip synthesis", *In Proceedings of theASP-DAC*, 2004.

[6] E. Grimpe and F. Oppenheimer, "Extending the SystemC synthesis subset by object-oriented features", *In Proceedings of CODES+ISSS*, October 2003.

[7] Ronald Hecht, Stepah Kubish, Harald Michelsen, Elmar Zeeb, and Dirk Timmermann, "A distributed object system approach for dynamic reconfiguration", *In Reconfigurable Architectures Workshop (RAW 06)*, Rhodos, Greece, April 2006.

[8] Michi Henning, "A new approach to object-oriented middleware", *IEEE Internet Computer*, vol 8, pp. 66–75, 2004.

[9] Antoine Jalabert, Srinivasan Murali, Luca Benini, and Giovanni De Micheli, "XPipes compiler: A tool for instantiating application specific Networks on Chip", *In Proceedings of Design, Automation and Test in Europe Conference (DATE'04)*, volume 4, pp. 1 – 6, February 2004.

[10] Ahmed Jerraya and Wayne Wolf, *Multiprocessor Systems-on-Chips*, Morgan Kaufmann, Elsevier, 2005.

[11] K. Keutzer, A. R. Newton, J.M. Rabaey, and A. Sangiovanni-Vicentelli, "System-level design: orthogonalization of concerns and platform-based design" *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, December 2000.

[12] J-Y. Mignolet, V. Nollet, P. Coene, D. Verkest, S. Vernalde, and R. Lauwereins, "Infrastructure for design and management of relocatable tasks in a heterogeneous reconfigurable System-on-Chip", *In Proceedings of the DATE '03 Conference*, 2003.

[13] P.G. Paulin, C. Pilkington, M. Langevin, E. Bensoudane, O. Benny, D. Lyonnard, B. Lavigueur, and D. Lo, "Distributed object models for multi-processor soc's, with application to low-power multimedia wireless systems", *In Proceedings of the Design and Test in Europe (DATE '06) Conference*, Munich, Germany, 2006.

[14] C. Schulz-Key, M. Winterholer, T. Schweizer, T. Kuhn, and W. Rosenstiel, *In Proceedings of theASP-DAC*, 2004.