Chapter 14

TRANSPARENT IP CORES INTEGRATION BASED ON THE DISTRIBUTED OBJECT PARADIGM

Fernando Rincón, Jesús Barba, Francisco Moya, Félix J. Villanueva, David Villa, Julio Dondo, Juan Carlos López
University of Castilla-La Mancha, Ciudad Real, Spain

Abstract: Heterogeneous system architectures are currently the main platform on which an ever increasing number of innovative applications (i.e. smart home or ambient intelligence applications) rely. When designing these complex systems, one of the most time-consuming tasks is the definition of the 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 for a particular implementation, which may limit the flexibility of the system, and makes the solutions difficult to reuse. In this chapter, we describe how to provide a unified abstraction for both hardware and software components that have to cooperate with each other, independently of their implementation and their location. Based on this abstraction, we define a low-overhead system-wide communication architecture that offers total communication transparency between any kind of components. Since the architecture is highly compatible with standard object-oriented distributed software systems, it also enables seamless interaction with any other kind of external network.

Key words: System-on-Chip, IP CORE, Distribuited Object, HW/SW codesign

1. INTRODUCTION

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. In this context, Systems-on-Chips (MPSoCs) have been proposed as a promising solution. Nevertheless, one major challenge in such systems is the integration in the platform of the multiple 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 to be able to efficiently communicate with it (e.g. local, remote), which makes the correct design of a SoC even more complex. Thus, new methods that allow designers to get unified inter-communication methods on SoCs architectures in the system integration flow are in great need.

Some concepts taken from distributed object platforms such as CORBA or Java RMI have already been applied to SoC design in order to get a unified view of HW and SW modules. 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 seamless develop multi-SoC systems with different network technologies.

This paper is organized as follows. In Section 2 we present a motivational example that will serve to guide us through our proposed approach to homogeneous hardware and software modeling. In section 3 we present the overall hardware SoC architecture of a system level middleware. In section 4 we revisit the motivational example under the distributed object approach described in section 3. In section 5 we show some experimental results. In section 6 we overview some related work. Finally, section 7, summarizes the contributions of the paper and presents possible future research directions.

2. MOTIVATIONAL EXAMPLE

Let's consider the design of a System on Chip for applications with some cryptographic requirements. For such purpose, the system will include a third party DES IP core that is able to provide encryption an decryption of certain blocks of data. We consider three different usage scenarios of the DES core. In the following paragraphs we will first describe each of the situations and the problems found in typical approaches, while in the next section we will analyze an alternative solution.

For the example we will use one of the DES cores provided by Opencores[1], that will also be used to illustrate the experimental results in section 5.

2.1 HW-HW Integration

The first scenario will be the use of the DES IP core from another hardware component. The DES core obtained from Opencores has a very simple interface with an encrypt and decrypt signal, a bus for providing the key, and the input and output data buffers.

Let's suppose that the SoC uses an OCP[2] bus for cores and processors interconnection. The first task would be to adapt it to that concrete bus. For such purpose we could use the CoreCreator tools from the OCP suite, and automatically generate the OCP wrapper out of a simple description of the DES interface. However, there is still some work to do, since we need to serialize the reception of the key and data input block, and the transmission of the data output block, since they do not fit in the bus word size.

On the other side, for the core using the DES, we should follow a similar but inverse procedure. Once known the bus interface, and the transaction-level protocol for providing the data and obtaining the results, we could write the functionality of the client core, next include some logic for the serialization of the transmission, and wrap it automatically to the OCP bus. This is a very normal procedure for IP integration, where both components are attached to the bus through some bus adapters (wrappers).

One of the advantages of using bus standards such as OCP is that they ease reusability of previous designs. However, interoperability at the level of operations is not guaranteed by the standard. The definition of a certain order of the key and data arguments is hardcoded inside the wrapper. Also the way arguments are divided into bus-size words is completely implementation dependent. That is the reason why cores with the same interface may not be interoperable.

All those problems are due to the loose coupling between functionality and communication. Therefore, any change in the component designed will also affect all of its clients, and may also imply their redesign.

2.2 HW-SW Integration

As a second scenario, we will consider how the DES could be used from a software client. To do so, in the classical approach we would need some kind of driver or API interface. These APIs are very dependent on the concrete core, and not easy to generate automatically. Even one minor change such as the modification of the address of the target component, or the order in which arguments are transmitted would require major modifications in the code.

Moreover, the APIs can be influenced by the specific transport architecture. For example, we could consider to add to the DES core the possibility to receive a set of words for batch processing with the same key. Transmissions from the CPU are performed word by word, and therefore can not take advantage of high-performance facilities such as bursting. Even, since a hardware client will probably transfer the block as a burst, it may be necessary to provide two different interfaces for both hardware and software clients.

2.3 Remote Communication

The third scenario will correspond to the request of the cyphering from outside the SoC. This may be the case for a pervasive computing application that needs some cyphering, for example. This interaction may be carried out though a wireless interface, for example.

This scenario is not very common, mainly due to the difficulty of multiplexing the ethernet between several components plus the microprocessors, which normally act as the masters of the device. Even, it is not clear how to translate network packets into the required bus transactions for the arguments and results of the operations. It would, however, be very useful to have remote communication to and from external clients, to make special computational resources accesible, for debugging purposes, for remote configuration, or even for remote reconfiguration of the SoC.

3. THE SYSTEM-LEVEL MIDDLEWARE

Most of the problems that SoC designers face nowadays are recurrent, and they have been tackled for decades in heterogeneous distributed computer networks environments. Since the 90’s, the use of a system middleware has been the satisfactory solution in this field. Although it can be established a correlation between the existing problems in computer networks and SoCs, the extension to the latter is not straightforward since they have their own special requirements, such as low power consumption, or low execution overhead, for example.

A middleware is an abstraction layer whose main objective is to provide an homogeneous communication mechanism between the components of a distributed system. 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 defining the set of messages client and server exchange. The objective is to provide orthogonalization between behavior and communication.

Applications using the middleware are usually based on the object-oriented programming model. 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 suited to deal with heterogeneity and scalability of applications.

In RMI any method invocation must take place between certain adapters, a Proxy (the client adapter) and a Skeleton (the server adapter). From the client's 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. Thus, proxies and skeletons completely hide the real communication process. Also, in most 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.

We could consider the SoC just as another type of distributed system. Like such systems, a SoC is composed of a set of heterogeneous computing and storage resources linked through some interconnection infrastructure, and suffers the same kind of problems: scalability, heterogeneity, different communication technologies, etc. Hence, it seems reasonable to apply the same kind of solutions, and concepts, although not necessarily the same implementation.

In the following paragraphs we briefly describe the main components of the system middleware (Object Oriented Communication Engine - OOCE).

3.1 The Communication broker

This layer of the middleware distributes remote invocations from the clients to the servers. In software systems it is normally a layer built on top of the operating system.

One of the main differences in OOCE with respect to the communication broker is that all components in the system share a physical communication infrastructure, which can be a bus or an on-chip network. The bus (or network) is already able to route the messages from one object to another, so there is no need of an extra layer for such purpose. Even for software objects, there is no extra layer, but remote invocation is a communication

primitive. Thus there is no need of an operating system to provide remote communication.

3.2 Proxies and Skeletons

Proxies and skeletons provide transparency, in 3 different aspects:

1. In the location of the target, which is normally coded in the proxy, and not hardcoded in the object (the functionality)
2. In the implementation technology of the objects. SW or HW proxies will generate exactly the same transactions in the bus. That makes it impossible to know if the invocation came from a HW or SW object, as it also happens with the response.
3. In the communication technology employed. This relates to how addresses for bus transactions are built from the target object and the operation requested; how the arguments are ordered, so all requests for the same operation are always performed the same way, with independence of the source; and how data types are serialized for their transmission through the bus.

Finally, we should highlight that proxies and skeletons can be generated automatically from the object interface description, so objects can be reused under any other different context (another bus protocol, for example) just regenerating the corresponding adapters.

3.3 Hardware Cores

A hardware core in the SoC will be the combination of three parts: 1) the hardware object, which contains pure functionality; 2) one skeleton, as an adapter for those operations that the object is able to serve; 3) as many proxies as the object uses as a client.

From all three parts, only the object is meant to be reusable, while skeletons and proxies should be efficiently generated depending on each particular case.

But even cores not been designed with this approach in mind may be used in the system middleware. For example, any RAM memory can be seen as an object providing read and write operation for bytes, words, double words, or even larger data blocks. The only thing required is a proxy that translates such operations into the proper transactions (DMA access for a block transfer, for example).

3.4 CPU Adapter

The main difference between hardware and software objects (in the OOCE context) is that software objects share a common processing element, while hardware objects execute in their own. This makes it necessary some multiplexing mechanism for SW clients to have access to the bus. This multiplexer is called the Object Adapter, and consist in a set of SW routines with a standard API that must be linked with the object code of the SW clients. For every object to be able to have access to the bus, first it must be registered in an Object Adapter.

Another problem with HW to SW invocations is that objects inside the CPU are not visible out of it. CPUs are usually just masters of the bus, and are not addressable. Here the solution adopted has been to insert a bus interface between the CPU and the bus. In SW to HW invocations, the interface simply buffers the invocation and translates it into a bus transaction. In HW to SW invocations, the interface holds a translation table with bus addresses and object identities. If any of these addresses is detected in the bus, the interface buffers the transaction and notifies the Object Adapter in the CPU through an interruption. The OA then routes the invocation to the proper object, and the response back to the interface, if there is one. The interface then provides the server capabilities to the objects inside the CPU.

3.5 Remote Bridge

The aim of the remote bridge is to translate internal (to the SoC) invocations to external ones through some kind of network interface. The information that must be transmitted on both sides of the communication has already been serialized, so the main task of the bridge is to pack it into the messages for a certain network transport protocol.

On the SoC bus side the bridge listens for transactions addressed to external objects. Those are recognized through an internal translation table, where some internal addresses are mapped to the network addresses of the referred objects.

On the network interface it performs the opposite task. In any case, messages coming in and out of the interface have always exactly the same format as internal interactions.

4. THE DES EXAMPLE REVISITED

The distributed object paradigm establishes a clear separation between the programming model and the arquitecture supporting it. Also, the OOCE platform allows the transparent integration of either hardware or software components. Thus, we can distinguish three different roles during the implementation of the system. On one side the typical hardware and software engineer roles. On the other side an integration specialist is required for the design and integration of the communication platform, as the backbone of the subsystems.

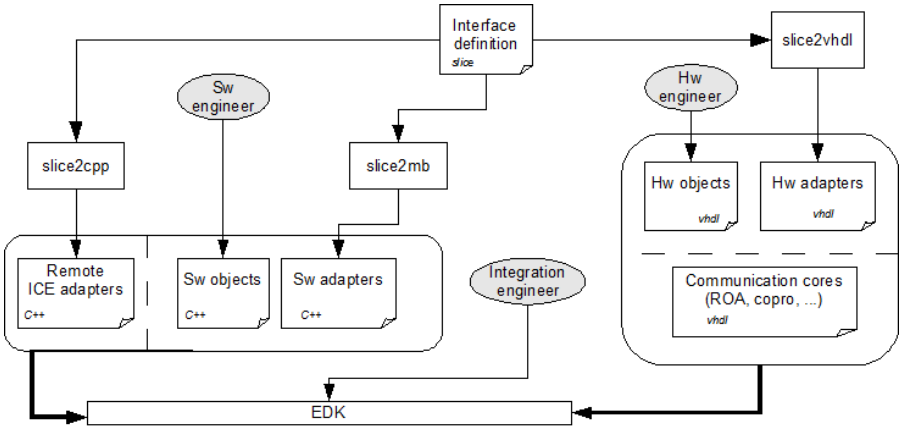

Figure 14-1. System Design Flow

Figure 14-1 shows the relationship between the three roles, as well as the flow for the automatic generation of the architecture, that once integrated with the rest of hardware and software entities becomes the final system. This system is not limited to one chip, but can also include components deployed on other types of computation nodes linked through a communication network.

```
module slice_example {
   ["hw:bus:plb", "hw:bus:args:64"]
   interface DES {
      long int encrypt(long int key, long int data_in);
      long int decrypt(long int key, long int data_in);
   };
};
```

Figure 14-2. Slice definition file for the DES core

The starting point of the flow is the interface specification file. This file includes the description of the interfaces for each object in the system. They are specified using an interface definition language (IDL) which is implementation neutral. Since OOCE is inspired in the ICE[3] middleware, the IDL is expressed using the Slice language. Figure 14-2 shows the slice interface definition for the DES example.

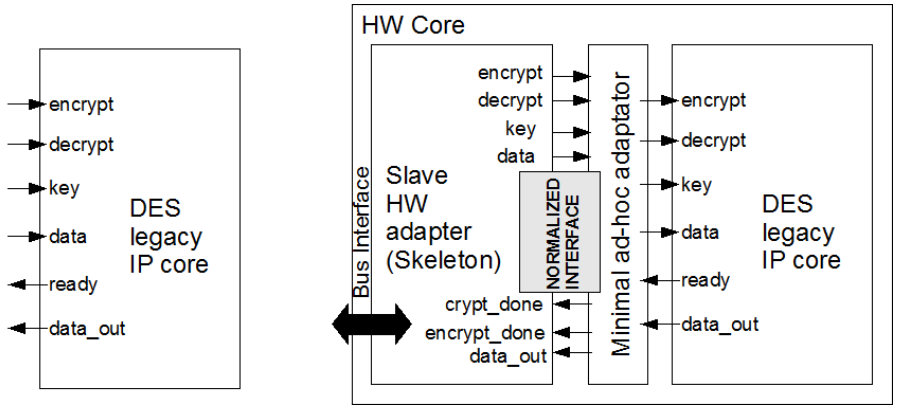

Figure 14-3. DES Hardware Component. a) DES legacy IP core. b) DES object + adapter

Although the interface description is implementation neutral, and since the platform must be generated from this file, it may include some metadata to guide the synthesis tools. It may provide extra information such as the implementation technology (hw or sw), the bus protocol (OPB, PLB, OCP, ...), the communication type (asynchronous or synchronous, blocking or non-blocking, ...), etc. This is in fact one of the tasks of the system integrator, to annotate the interface definition file, generally through iterative refinement to provide the platform that best suits a certain system.

The Slice file is parsed by different code generators. Each of them, depending on the metadata, will generate specialized adapters for every object and context. The VHDL code generator (*slice2vhdl*), for example, will write synthesizable vhdl models for the adapters that will be appended to the clients or servers developed by the hardware engineers, obtaining the different cores of the system.

In case reusing cores are not designed with the distributed object approach, some extra logic is required to adapt their legacy interface to the one derived from the slice method signatures (figure 14-3b). The overhead for the DES core is almost negligible due to the simplicity of the normalized interface proposed. However, writing the hardware object from scratch will

not incur in such overhead. The object will only include the functional code for implementing the operations, and will leave the communication responsibilities to the generated adapters.

From the same slice definition, the *slice2cpp* generator will produce the equivalent adapters (proxies and skeletons). Those adapters are based on a function library which implements the link between the CPU and the coprocessor for the communication. A device driver is no longer a collection of low-level reads and writes to a register bank interface. Now, the programmers deal with software objects that are instances of the proxies to the hardware models.

```
class DES {
   public:
      long int encrypt(long int key, long int data_in) {
         // the object identity and operation identity are mapped onto an address
         putfsl(OBJ_ID<<16 + ENC_ID);
         // the bus interface needs to know the number of arguments
         putfsl(NON_VOID|ENC_ARGS);
         // arguments and return values are serialized as two 32 bit words
         putfsl(key & 0xFFFFFFFF);
         putfsl(key >> 32);
         putfsl(data_in & 0xFFFFFFFF);
         putfsl(data_in >> 32);
        getfsl(data_out_low);
        getfsl(data_out_high);
         return data_out_high << 32 + data_out_low;
    }
}
```

Figure 14-4. C++ code for the DES SW client

Figure 14-4 shows how the distributed object model eases the task of the software developer. Here the use of the DES core is concerned with the invocation of the methods provided in the proxy, which completely hides all the implementation details of the communication. The proxy behavior with respect to the blocking or not of the executing thread may be configured in the slice definition file, providing a high degree of control to the programmer. This is completely orthogonal to the way communication is implemented through the bus.

The final task for the system integrator is the combination of the different adapters with hardware and software objects to build the hardware cores and

software components, and the inclusion of the rest of components of the OOCE engine.

To illustrate the robustness of the approach, let's suppose that the target platform is modified and the bus protocol and the bus size are now different. Those changes will not affect either the program using the DES component, or the adaptation of the DES core to the normalized interface. No modification will also be required to any of the other hardware clients (hardware components using the DES). It will only be a question of automatically regenerating the corresponding adapters.

In the proposed approach, remote communication does not imply nothing but a special bridge connected to the ethernet adapter. As it happens with SW to HW communication, interoperation is guaranteed by the use of the same bus transactions for the same operation requests. Thus, once a network packet coming from the outside reaches the bridge, it is injected in the SoC bus just if it was generated locally. The target core will recognize the address and perform the required operation putting back the results in the bus. These results are translated in the bridge back into network packages, and sent back to the remote client.

It is also possible to execute operations from remote servers from both hardware and software clients. They simply need the corresponding adapter (with the target server interface) that will translate operation requests into bus transactions. However, this transactions will not correspond to any address in the SoC address space, but will be mapped to the bridge. So the bridge will pack them into network packets, after a translation of the local SoC address to a network address (protocol and port), and will route them through the ethernet device.

5. EXPERIMENTAL RESULTS

As a proof of concept, the OOCE has been fully prototyped on the Xilinx XUP-V2Pro platform. We have performed a set of experiments, where we have considered all the different communication mechanisms, and tested all possible interactions between components. We have also characterized the results in terms of latency (table 14-1) and area (table 14-2). The types of interfaces refer to: I) simple synchronous and blocking read and write operations (without bursts), II) simple asynchronous and non-blocking read and write operations, and III) same as II using bursts.

Table 14-1. Communication latency for all types of OOCE interations

Iface	Invocation type	Latency (cycles) write/read
Type I	Hw –> Hw	3 / 2
	Sw –> Hw	N.A. / 10
	Hw –> Sw	N.A. / 11
	Sw –> Sw	50 / 21
Type II	Hw –> Hw	4 / 2
	Sw –> Hw	21 / 10
	Hw –> Sw	21 / 11
	Sw –> Sw	42 / 21
Type III	Hw –> Hw	19 / 17
	Sw –> Hw	56 / 27
	Hw –> Sw	56 / 29
	Sw –> Sw	108 / 56

Table 14-2. Area cost for the hardware adapters

Interface type	Resource	Area
Simple R/W	Hw proxy	4 FFs
		7 LUTs
Simple R/W	Hw skeleton	2FFs
		153 LUTs
Async R/W + burst support	Hw proxy	102 FFs
		208 LUTs
Async R/W + burst support	Hw skeleton	102 FFs
		208 LUTs

Also a completely SW version of the DES algorithm was implemented on the Microblaze 32 bit processor, to use it as the software reference model. Next, all the middleware infrastructure was generated for a SoC with HW, SW and remote clients for the DES model from Opencores.

Results measured for the DES encryption of a 2KB data block with a 55-bit key where the following: 102 microseconds for the fully SW version, 7 microseconds for the encryption using the DES core and a software client, and 5 for the completely hardware solution.

Also communication times for an off-chip invocation through an ethernet interface were measured. The reception of the packet took 218 cycles. The remote bridge translated the message into a bus transaction in 76 cycles. The execution of the invocation took 16 cycles. Finally, the result was packed into an Ethernet frame in 72 cycles and transmited back in 218 additional cycles.

6. 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 Keutzer[4], and more recently by Cesario[5] and Gertslauer[6], 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 care about location transparency. In Mignolet[7] a uniform communication mechanism for HW and SW resources is proposed, based on a central HW-SW Operating System and a HW abstraction layer to provide task abstractions for HW components. Previous works by Paulin et al.[8] already apply 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 Hetch[9]. Previous results on automated generation of communication infrastructure for SoC design[10,11] are also applicable to adapters generation.

Object-based and object-oriented approaches[12,13] 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.

7. CONCLUSIONS

The communication architecture presented in this paper extends the distributed object paradigm to SoC platforms. The proxy and skeleton abstractions plus RMI semantics, provide a simple way to decouple component functionality from communication implementation. From the designer perspective, this provides an homogeneous view of the system as a collection of communicating objects. From the implementation point of view, 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 are part of the middleware can automatically be generated based on a few descriptions on the interfaces of the objects. 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 in high overheads.

8. REFERENCES

1. Opencores; http://www.opencores.org; last visited June, 27, 2008.
2. Open Core Protocol (OCP); http://www.ocpip.org, last visited June, 27, 2008.
3. Internet Communication Engine (ICE); http://zeroc.com, las t visited June, 27, 2008.
4. Keutzer, K., Newton, A.R., Rabaey, J.M., Sangiovanni-Vincentelli, A. System-level design: orthogonalization of concerns and platform-based design. IEEE Trans. Computer-Aided Design of Integrated Circuits and Systems, 19, 12 (Dec. 2000).
5. 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, (70), 2004.
6. 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.
7. 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.
8. 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 DATE ’06 Conference, Munich, Germany, 2006.
9. 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.
10. 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.
11. A. Gerstlauer. Communication abstractions for system-level design and synthesis. Technical Report CECS-TR-03-30, UC Irvine, 2003.
12. Grimpe, E., Oppenheimer, F., Extending the SystemC Synthesis Subset by Object-Oriented Features. In Proceedings of CODES+ISSS, Oct. 2003.
13. Schulz-Key, C., Winterholer, M., Schweizer, T., Kuhn, T., Rosenstiel, W. Object-Oriented Modeling and Synthesis of SystemC Specifications. In Proceedings of theASP-DAC, 2004.

ACKNOWLEDGMENTS

This work has been funded by the Spanish National and the Castilla-La Mancha Regional Goverments under grants TIN2005-08719 and PAI 08-0234-8083, respectively.