Automatic HW/SW Interface Generation for Seamless Integration from Object-Oriented Models

J. Barba[1], F. Rincón[1], F. Moya[1], D. Villa[1], F.J. Villanueva[1] and J.C. López[1]
[1]Dept. of Technology and Information Systems. School of Computer Science, University of Castilla-La Mancha (Spain)

Abstract - *Hardware-dependent Software (HdS) synthesis is becoming a key aspect in the development of mixed software and hardware computation platforms due to the increasing software content in such systems. Uniform and high-level interface models for both HW and SW are required in order to perform HdS development in parallel with the platform prototyping.In this paper, an optimal hardware and software infrastructure that enables the HW/SW interfacing from an object-oriented perspective is presented. The generation of such infrastructure is performed semi-automatically from UML models. Using this approach, the resulting high-level programming interface enables the reuse of the embedded software.To evaluate the efficiency of our proposal a prototype implementation on the Xilinx-V2 Pro platform has been made.*

Keywords: HW/SW interfacing, HdS generation, Object-Oriented design methodology, SoC design, automatic synthesis.

1 Introduction

Nowadays, it is widely assumed that a higher productivity in the design of complex state-of-the-art heterogeneoteus systems (e.g. SoCs or MPSoCs) is achieved raising the abstraction level of the models used in the specification phase. Ideally, hardware architects and software engineers could use a common system specification from the beginning of the design process enabling concurrent workflows and overcoming the drawbacks that exhibits traditional design methodologies [1] (i.e. low productivity of the teams involved).

Although there have been so many proposals from the academia, the acceptance of such ideas and their incorporation to commercial CAD tools has been limited. The reasons in the root of such as skepticism include: (a) models, techniques and languages that do not satisfy the demands of the hardware and software developers; (b) a complete and automatic workflow, from design to synthesis and implementation is difficult to develop; and (c) solutions with a satisfactory degree of quality are almost impossible to obtain.

Only the tremendous pressure exerted by the embedded systems market has force the IC industry to adopt significant contributions in fields like modeling and simulation (e.g. *Transaction-level Modeling*).

Among the many aspects that should be considered in the design of complex heterogeneous systems, this work will be focused on HW/SW interfacing, key in the synthesis process of Hardware-dependent Software (HdS). HdS elaboration has become one of the principal matters of concern in embedded systems due to the increasing amount of software such systems contain.

Our proposal is based on an object-oriented view of the platform and its functional elements. We have chosen the concept of *object* and the *Remote Method Invocation* (RMI) semantics as the framework to unify the communication interface for both HW and SW components. Of course, we do not attempt to establish objects as the unique and valid modeling concept, since it depends on the nature of the target application. But *objects* provide important advantages when applied to reconfigurable computing [2] (i.e. component state management) and fit very well in multiple programming models that efficiently could be used to abstract the HW/SW interface [3][4].

The object model has also demonstrated to be suitable for the specification and architecture of multimedia and streaming applications (i.e. OpenMax [5]), two of the main research areas on which this approach will be proven.

So, we advocate for the adoption of the object-oriented paradigm to improve the design process of heterogeneous embedded systems. In the case of HdS, the objects provide a more stable development scenario, at the same time they provide the basis of *design for reuse* practices [6]. In other approaches, the interfaces offered to the programmers are low-level and very sensitive to variations in the hardware cores (i.e. the access through a register interface). Software developers may then invest most of their time and effort in rewriting a small part of the software, leading to a more unproductive work.

The automatic generation of software drivers and hardware adapters is accomplished by means of a template-based mechanism, similar to those most of the commercial software middlewares for networked systems have. This approach frees the designer of (re)writing and (re)defining the system interfaces in every design phase, avoiding a non negligible amount of work.

By means of reusing code and models and the automation of the software (and also hardware) adapters generation, our approach aims to be a catalyst of the design process, boosting the productivity of the development teams.

The paper is organized in the following sections. Section 2 offers an overview of the related work. In section 3, we present a general view of our HW/SW interface synthesis process. The hardware platform which supports the proposed embedded software development framework is analyzed in section 4. Then, the software services that will be used by the high-level generated primitives are explained in depth in section 5. Finally, the paper is closed with sections 6 and 7, presenting the experimental results, conclusions and future work.

2 Related work

Lately, there is a significant amount of research work in the area of HW/SW interface modeling and HdS generation. For example, Mignolet et al [7] provide a uniform communication scheme for hardware and software tasks within the OS4RS operating system. This approach is limited to applications modeled as a set of concurrent threads (tasks) as in TTL [8]. TTL 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.

The concept of task as a modeling concept exhibits many drawbacks when the migration of the functionality is considered. The election of the synchronization points is not easy and the concept of *state* in tasks is not clearly defined. On the contrary, objects have an associated semantic that makes them more suitable for such dynamic scenarios.

Besides, threads are also not well considered to be used in parallel programming models since their non-determinism [9], whereas distributed object models have proven their viability in this area (i.e. Multiflex [10]).

Other approaches must be also considered. For example, the BORPH operating system [11] offers a homogeneous UNIX interface for both software and hardware processes in the form of file access primitives. Although the kernel interface easies the development of applications (since it is familiar to programmers), it is not clear that such interface facilitates the reuse of hardware. The *IOREG* interface models the access to the hardware as a memory which does not really raise the abstraction level of the hardware part.

In [12] 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 automate the interface implementation.

COSY [13] and ROSES [14] use a library-based approach to generate both hardware and software wrappers. The communication model presented in COSY is based on channels which are implemented as FIFOs. In practice, the interaction with the hardware cores turns into low-level read/write operations in COSY. Regarding ROSES, once again the available API functions for a software task that needs to communicate with the generated hardware wrappers are low-level, in the form of *read/write put/get* primitives.

Schirner et al. present in [15] an automatic method to synthesize HdS from TLM models. SpecC channels are modeled using the TLM concepts and SW to HW communication (the reverse scenario is not considered) is implemented using the ISO/OSI layering model. Although TLM was originally conceived for simulation and verification purposes, it has been rapidly adopted by synthesis techniques. The abstract, high-level concept of transaction can efficiently model the HW/SW interface. In this line, it is worth mentioning the work of Klingauf et al. It describes how the concept of *Hardware Procedure Call* [16], on top of TLM concepts, offers a truly high-level access mechanism to HW functions in a service oriented manner. There are no references to the architecture of the resulting hardware/software supporting platform for HPCs and its efficiency.

3 Interface Specification

As previously mentioned, our approach is based on the RMI semantics as the enabling concept to achieve a path to the HW/SW interface synthesis from high-level object-oriented models. This work is based on the *Object-Oriented Communication Engine* (OOCE) [17], a hardware/software middleware that implements the concept of RMI for SoCs in an efficient manner.

The distributed object model behinds OOCE specifies how method invocations are translated to technology dependent operations. Basically, a method invocation becomes a transaction over the on-chip communication infrastructure to transmit a request message from the source to the target (*client/server* in the OOCE terminology). A request message is composed by a header, which is codified in the address lines, and a body which results in a byte sequence that codifies the arguments in the data lines. The way the method parameters are codified within the data bitstream and the number and type of messages interchanged by clients and servers is standardized in OOCE, so that component interoperability is assured. In [17], the reader can find an overview of the superstructure and main objectives pursued by OOCE.

The proposed workflow (see Figure 1) of the HW/SW interface synthesis process makes use of the UML (*Unified Modeling Language*) notation to represent the structural aspects (*Object Diagram*) of the system and the relationships

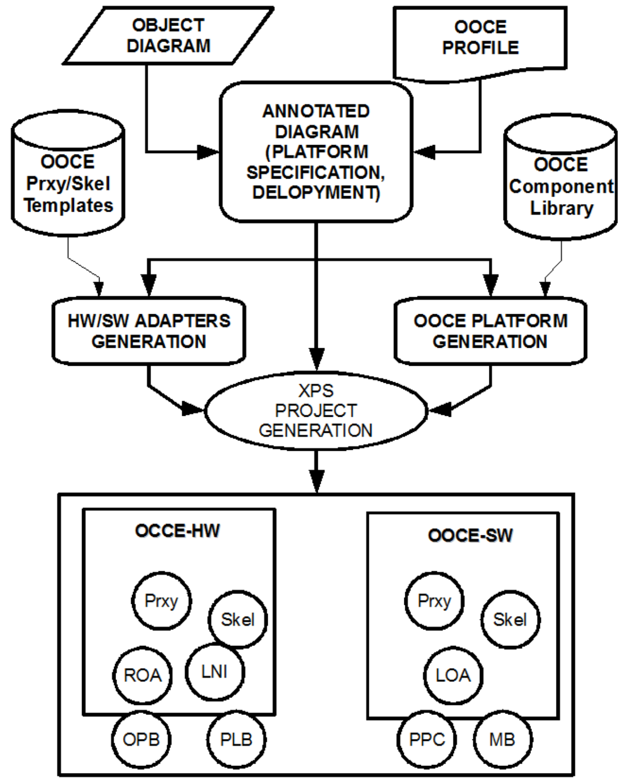

Figure 1. Interface synthesis and prototype platform generation flow in OOCE.

(*Collaboration Diagram*) between the objects that are part of the system. We have chosen UML as the modeling language since it is object-friendly and it also has demonstrated to be easy to use and integrate into a complete SoC design flow [18], which is one of the future challenges in OOCE.

Before the generation of RTL/C/C++ code for the hardware and software wrappers, the designer must annotate the UML entities with the stereotypes defined in an OOCE UML profile in order to specify (among many other aspects): (a) whether an object is going to be a SW or a HW object (partitioning); (b) the processor and model where the SW object is going to run; (c) a concrete scheme of communication (blocking or non-blocking); and (d) the bus infrastructure and protocols used to integrate the components. At this time, this step is performed manually by the designer through a GUI, but the aim is to derive such deployment and architectural information in an automatic way from a previous exploration of the design space. Figure 2 shows a simplified version of a minimal application where the main object (actually, the control logic implemented in software) needs of the cryptographic services of a hardware object that implements the DES algorithm.

A textual representation of the resulting UML diagram feeds: (1) a hardware interface compiler which generates the OOCE hardware adapters or *skeletons*; (2) a software interface compiler which generates the OOCE software drivers or *proxies*; (3) an OOCE platform generator which

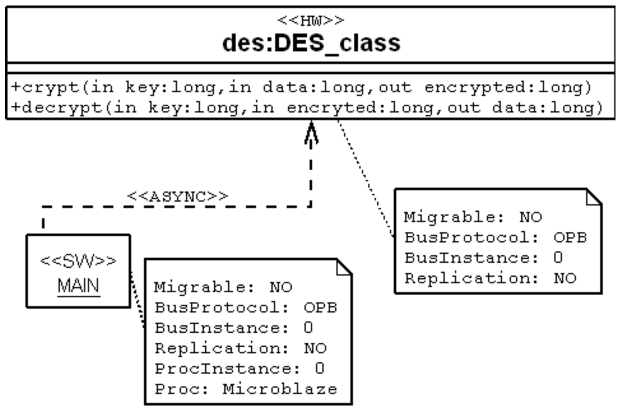

Figure 2. UML OOCE annotated diagram for a minimal cryptographic application.

selects, from a component template library, the communication engine components required by the application; and finally, (4) a script that combines all the above mentioned elements and generates a Xilinx XPS. Considering the platform generator, notice that some features of the component templates are tuned by this generator to be optimal (e.g. the size of the CAMs used to perform the translation of the bus addresses to software object's IDs, the size of the FIFOs that hold temporally protocol messages, etc.). To this end, the generator uses the information contained in the UML spec project. The two interface compilers and the platform generator work autonomously and they do not need of the designer's intervention.

At this point, the designer obtains a complete prototyping platform which is ready to be synthesized using Xilinx EDK standards tools. Figure 3 sketches the derived HW and SW infrastructure from the OOCE UML annotated diagram of Figure 2. The designer should only: (a) connect the DES core (dashed box), implemented as an OOCE hardware object, following a standard module interface and activation protocol [19]. This component could be retrieved from an existing OOCE compliant IP library, or a legacy one may be easily adapted; and (b) write the behavior of the client application, *MAIN* object (dashed ellipse), using the generated OOCE drivers. Examples of the use of the software DES API are also provided to the programmer.

Hereafter, we give a more detailed view of the OOCE elements concerning HW/SW communication.

4 Supporting hardware

4.1 IP skeletons

A HW skeleton (HWS) is the OOCE adapter in charge of providing connectivity to a hardware core that needs to be accessible from software. The HWS interprets address lines in order to detect if this specific HW core is the target of the communication. The HWS logic activates the hardware object interface signals to initiate the operation and decodes

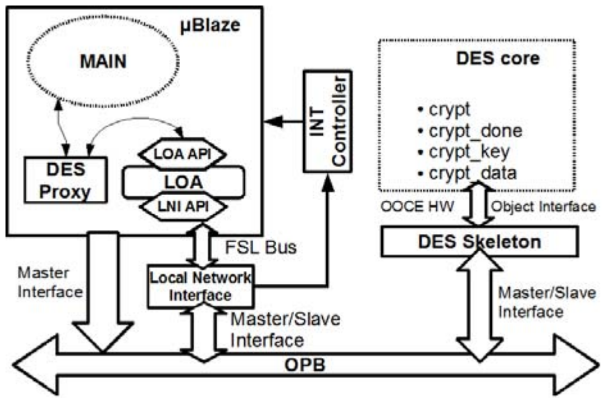

Figure 3. Derived HW and SW platform.

the stream of bytes to push the arguments to the *server*. Once the core notifies that the operation has been completed, the HWS builds a protocol response message with the output data as the body.

OOCE defines several skeleton templates that support blocking and non-blocking communication semantics. The specialization process is managed in an automatic way thanks to the interface compiler.

4.2 The Local Network Interface

The LNI is the bridge between the system microprocessor, where the application runs, and the hardware cores that implement the required functionality. The main goal of the LNI is to keep compatible the hardware interface and activation protocol defined in OOCE with the software invocation mechanisms. The LNI is conceived as a coprocessor, realizing the master interface of the on-chip bus in order to obtain the maximum performance (e.g. using bursts). The LNI also implements the slave bus interface, behaving as a generic skeleton, which is actually the only entry point to the processor from the HW objects (which saves interruption lines).

Outgoing communication (SW to HW) is managed almost without significant intervention of the LNI. The low level software routines, which abstract the link with the processor (see next section), put an already formatted request message into the LNI Tx FIFO. Once the presence of a new invocation is notified to the LNI and the access to the bus is granted, the body of the message is written word by word to the target address (which was first pushed into the Tx FIFO).

On the contrary, incoming communication (HW to SW) needs a little more effort and resources. To route the relevant bus traffic through the LNI to the processor, a *Translation Address Table* (TAT) is required. Each software object in the system has a unique *Object Identification Number* (OIN) and only the OINs of the software objects reachable from hardware are maintained in the TAT. The LNI uses the information hold in the TAT to detect incoming messages. The LNI activates an interrupt signal when a new message is available to the application. HW to SW communication is a scenario that is not usually considered by other approaches. Thus, in OOCE, HW cores may have an “active” role in the system, as they are able to invoke software methods as well as asynchronously respond to a request.

5 The intermediate software layer

The LNI represents a means to easily inject/retrieve OOCE protocol messages into/from the communication infrastructure. The protocol messages addressed to software objects are available to the applications through the LNI Rx FIFO. However, as said before, the LNI is conceived to be tightly attached to the processor and therefore the link between them is very dependent on the implementation.

To abstract the processor-LNI link, we have defined a layered software architecture that progressively offers services that help the programmers to use this communication infrastructure. Next, we review the main features of each layer and how they provide the object-oriented view of the system to the embedded software programmers.

5.1 The LNI layer

At the lowest level, the *LNI_link* interface offers a collection of services that hide the implementation details of the communication with the coprocessor. The primitives that support this interface are *send_msg*, *recv_msg*, *register* and *delete*. The last two functions add/remove the OINs of the software objects to/from the LNI TAT. The management of the TAT is performed by putting special operation codes on the coprocessor link that are conveniently interpreted by the LNI.

The implementation of the LNI layer has to be done manually for each type of connection between the processor and the coprocessor. In fact, the LNI layer is the only one that must be tailored for a new target platform. The rest of the software stack can be automatically obtained and it is platform independent, so it has only to be written once. The simplicity of the primitives to be coded makes this process very easy.

5.2 The LOA layer

The next layer in the OOCE software stack is the *Local Object Adapter*, a medium-level layer that uses the LNI low-level services. The LOA is platform independent so that it is written just once for all platforms.

The primitives defined in the LOA make it possible that all the objects running in the processor may share the LNI link. The LOA behaves as a multiplexer of the incoming messages as well. The functionality of the LOA comprises:

```
---- Generated OOCE stubs --------------------
void DES_crypt_handler(int data,
                       tOOCE_object *object) {
  tSkel *skel;
  skel = (tSkel *)object;
  /* resume object's thread */
  sem_post(skel->sem);
}

void DES_crypt(tPrxy *prxy,long key,long data,
               long *encrypted) {
  tOOCE_msg msg;
  tSkel skel;
  tLNI_msg lnimsg;

  msg.src = 0;  /* message header */
  msg.dst = prxy->objid; /* OIN */
  msg.rid = prxy->rid++; /*request ID */
  msg.op = DES_CRYPT_MID;
  msg.type = OOCE_MSG_REQUEST;
  msg.size = 4;
  *(long *)(msg.data+0) = key;  /*marshalling*/
  *(long *)(msg.data+8) = data;
  skel.objid = 0;
  skel.handler = DES_crypt_handler;
  sem_create(skel.sem,1);
  LOA_send((tLOA *)prxy->loa,&msg,&skel);
  sem_wait(skel.sem); /*block*/
  LNI_link_recv_msg(&lnimsg,2);
  /* unmarshalling */
  (*encrypted) = *(long *)(lnimsg.data+0);
  return;
}
```

Figure 4. Generated C code version of a DES crypto SW proxy for Microblaze.

- A constructor that initializes the internal state and structures of the LOA.
- A set of methods to manage the *Active Object Table* (AOT). The AOT maintains a set of pointers to the software objects that can be accessible from outside the processor.
- An *activate* method that links the software object to a concrete LOA instance. The LOA includes the software object reference into the AOT.
- A *send* primitive that builds the low-level protocol message from an *OOCE_msg* structure and sends it using the *LNI_link* functions. If a response is expected (two-way invocations), a reference to the function that will manage the response message is provided. This reference is internally annotated into a *list of pending requests*.
- A *process* routine, which is actually the interrupt service handler that manages the notifications of new messages coming from the LNI coprocessor. Depending on the header information of the low-level message, it selects the response handler from the list of pending requests or the AOT.

5.3 Software proxies

On top of the LOA layer, a collection of high-level software routines are generated from the UML interface object specification that represents the services offered by a hardware core. An interface compiler has been developed to generate the software version of the *proxies* and the *method response handlers*, which provide the programmers with the illusion that they are talking with software objects instead of with hardware devices (as it really happens).

```
#include <stdio.h>
#include "xparameters.h"
#include "mb_interface.h"
#include "ooce.h"
#include "des.h"

int main(){
  tPrxy prxyDES;
  tLOA loa;
  long key,data,res;

  /* LOA instance creation and initialization */
  LOA(&loa);
  /* Proxy to DES constructor */
  DES(&prxyDES,XPAR_DESASYNC_0_BASEADDR);
  /* synchronous invocation */
  DES_crypt(&prxyDES,key,data,&res);
  /* Use the result */
  return 0;
}
```

Figure 5. Example of use of the OOCE object-oriented generated API

The proxy fills the LOA structures that will be translated into low-level protocol messages. The necessary data come from the function arguments and the internal parameters hold within the own proxy structure. The OOCE development framework also defines synchronous and asynchronous invocation semantics for software methods, as for hardware ones. Figure 4 shows the generated C code version of a synchronous proxy that allows accessing the DES core implementing the cryptography operation. Due to efficiency criteria, in most of the projects, C code is preferred. Although C is not an object-oriented language, it can be used to implement an object-like programming interface using structures and pointers as shown in Figure 4.

5.4 Application programming

Finally, Figure 5 illustrates how to use the generated software infrastructure. The resulting code is clean and easy to understand, which promotes reutilization and maintainability. The application is more robust to unforeseen changes in the hardware platform and almost the entire software stack can be used 'as is' in future designs. Therefore, the writing of the embedded software can start as soon as desired, even in parallel with the design of the hardware design. No matter the physical interface of the hardware core, it can be modified or replaced by a different core from another manufacturer; the logical interface remains invariable.

6 Experimental results

Since communication efficiency is crucial in typical mixed hardware-software computation scenarios, we have performed several experiments in order to determine the overhead introduced by the proposed infrastructure and the

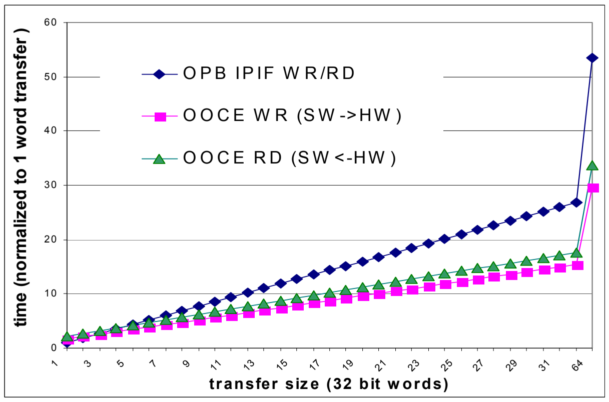

Figure 6. Performance of on-chip read/write transactions using Xilinx IPIF and OOCE.

automatically generated software stack. To this end, we have measured the time invested to complete read/write operations of various sizes using a generic OOCE-compliant hardware object. We have compared our results against the Xilinx IPIF-based implementation. The OPB IPIF architecture specification is a commercial solution which facilitates the connection of either Xilinx or the customer IP modules to the IBM On-Chip Peripheral Bus (OPB).

Figure 6 shows that the transmission time increases linearly with the size of the message transferred in all the cases, as it was expected. A *write* operation from software using the OOCE approach exhibits an important performance increase (up to 40%) when the size of the transfer is beyond the 3-word barrier. We have adapted the software proxies to use the classical I/O register interface when the number of writes/reads is below this limit, in order to be optimal. An OOCE *read* operation (incoming requests or reception responses) is less efficient than an OOCE *write* since the software routines start to take the words from the LNI link only when the entire message has been cached into the LNI Rx FIFO. On the contrary, a LNI bus *writing* pushes the body of the message at the same time the software proxy puts it into the LNI Tx FIFO. We are working on a parallel version of the reception process to improve the results obtained concerning the OOCE *read* operation The speed levels that can be achieved with OOCE can reach about 120 MB/s for both *read* and *write* operations (just up to 80 MB/s with OPB IPIF implementation).

In the described experiment, we have used a Microblaze soft processor and the LNI link has been implemented using two FSL (*Fast Serial Links*) interfaces. However, these data can be extrapolated when the Power PC version is taken into account. In this second scenario, the LNI link is implemented using the OCM (*On Chip Memory*) interface. In both cases, one of the reasons of the small overhead is the optimized version of the LNI and LOA software layers (only about 80 lines of C code).

Finally, Table 1 shows the synthesis results of the OOCE HW skeleton and the IPIF wrapper in order to evaluate the overhead in terms of hardware resources. Three legacy cores have been considered from OpenCores [20]. They have been modified to operate with blocks of data up to 1K words. Our skeleton architecture offers a significant reduction in the logic used by the core wrappers compared with the IPIF-based solution. It is worth mentioning that the synthesized LNI component represents less than 1% of the FPGA resources (in this case a XC2VP30 Xilinx-V2Pro), a small overhead that is shared by all the hardware objects in the system. This means that the percentages in the reduction of logic can be better when the design contains two or more hardware objects.

The adaptation work performed on the considered cores has consisted in the creation of the hardware wrappers needed to adapt the cores interface to the IPIF and the OOCE HW object interface. The complexity of such adapters and the time spent in the writing process is comparable (about 1 hour, senior engineer). However, the time spent in the writing process of a software application implementing a use case was double in the case of IPIF. In addition, in the case of OOCE, this application is fully reusable and portable to other platforms (i.e. the Power PC version).

Table 1. Synthesis results and comparison

Hardware overhead				
	FFs		LUTs	
	IPIF	HWS+LNI	IPIF	HWS+LNI
DES	1432	842 (-41%)	2312	1635 (-29%)
CORDI	404	229 (-43%)	630	370 (-41%)
MDCT	734	603 (-18%)	1093	1198 (+9 %)
LNI	123		376	

7 Conclusions and future work

From our point of view, the concept of object satisfies the need for a common view of the whole system shared by both, software and hardware engineers when facing the design task. This fact, along with the automatic generation of the HW/SW interfacing infrastructure, boost the productivity of the embedded software developers because (1) they do not have to wait for a physical platform prototype and (2) unnecessary iterations are avoided.

Current work is focused on the definition of a complete design methodology to exploit the potential of the distributed object approach stated in OOCE. This methodology will integrate the automatic HW/SW interface generation here proposed into an *Electronic-System Level* workflow. Our aim is to offer an incremental development cycle for SoC design making use of the concept of *location transparency* [17]. Rapid design space exploration, verification and synthesis from TLM models of the communication engine are also in the agenda.

8 Acknowloedges

This work has been funded by the Spanish Ministry of Science and Innovation under the project DAMA (TEC2008-06553/TEC), and by the Regional Government of Castilla-La Mancha under project RGRID (PAI08-0234-8083).

9 References

[1] Jerraya, A.A., "HW/SW Implementation from Abstract Architecture Models". *DATE*, 2007.

[2] Dondo, J. et al., "Dynamic reconfiguration management based on a distributed object model", *17th FPL International Conference*, 2007.

[3] Jerraya, A.A., Bouchhima, A. and Petrot, F., "Programming models and HW-SW interfaces abstraction for multi-processor SoC". *DAC, 2006.*

[4] Paulin, P.G., et al., "Distributed Object Models for Multi-Processor SoC's, with Application to Low-power Multimedia Wireless Systems". *DATE,* 2006.

[5] The Khronos Group Inc, "OpenMax Integration Layer API Specification", version 1.1.1, September. 2007.

[6] Rincón, F. et al., "Model reuse through hardware design paterns". *DATE*, 2005.

[7] Mignolet, J.-Y. et al., "Infrastructure for design and management of relocatable tasks in a heterogeneous reconfigurable system-on-chip". *DATE,* 2003.

[8] Van de Wolf, P., et al., "Design and Programming of Emebedded Multiprocessors: An Interface-centric approach". *CODES+ISS'04*, 2004.

[9] Edward A. Lee, "The Problem with Threads", *Computer*, vol. 39, no. 5, pp. 33-42, May 2006.

[10] Paulin, P.G. 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.

[11] So, H. K. and Brodersen, R., "A unified hardware/software runtime environment for FPGA-based reconfigurable computers using BORPH". *Trans. on Embedded Computing Sys.* 7, 2, 1-28, Feb 2008.

[12] Bouchhima, A., et al., "A unified HW/SW interface model to remove discontinuities between HW and SW design". *EMSOFT'05,* 2005.

[13] Brunel, J., et al., "COSY communication IP's". *DAC, 2007.*

[14] Wagner, F. R., Cesário, W., and Jerraya, A.A., "Hardware/software IP integration using the ROSES design environment". *Trans. on Embedded Computing Sys.* 6, 3, July 2007.

[15] Schirner, G., Gerstlauer, A., and Domer, R., "Automatic generation of hardware dependent software for MPSoCs from abstract system specifications". *ASPDAC,* 2008.

[16] Klingauf, W. et al.., "Embedded software development on top of transaction-level models". *CODES+ISSS '07*, 2007.

[17] Barba, J., et al., "OOCE: Object-Oriented Communication Engine for SoC Design". *10th Euromicro Conference on DSD*, 2007.

[18] Riccobene, E., et al. "A model-driven design environment for embedded systems". *DAC*, 2006.

[19] Barba, J. et al., "Lightweight Communication Infrastructure for IP integration". *IPSOC Conference*, 2006.

[20] OpenCores. http://www.opencores.org