Automatic deployment of component-based embedded systems from UML/MARTE models using MCAPI

Alejandro Nicolas, Hector Posadas, Pablo Peñil, Eugenio Villar
University of Cantabria
Santander, Spain
{nicolasa, posadash, pablop, villar}@teisa.unican.es

ABSTRACT

The increasing complexity in the development of embedded system is raising the need of system modularization, parallelization and component portability. High-level languages such as UML are clearly oriented to solve these needs, but implementation flows are usually highly dependent on platform details. Different platform-agnostic APIs such as MPI or MCAPI have appeared to increase the application independence from the executive HW. Nevertheless, the gap between the high-level models and the final system implementations is still too large. In this context, this paper presents a methodology for automating system deployment of component-based systems. The process starts from a high level description based on UML/MARTE, including complex channel semantics and provides automatic code generation for interconnection and deployment of system components based on MCAPI. This automatic process enables exploring different possibilities both in the component allocation and in the resulting concurrency, involving low designer effort.

1. INTRODUCTION

The evolution of fabrication technologies is enabling the development of increasingly powerful chips containing multiple processors. As a result, the development and deployment of concurrent applications, capable of taking advantage of the full computational power of these systems, are becoming critical issues. To solve that, development flows require modularizing the problem in order to reduce the effort required for modelling, implementing, testing and reusing previous works.

Component-based development (CBD) enables modular design flows, identifying and completely specifying the elements that compose the entire application [2]. It makes possible the substitution of a component for other with the same ports, improving product-line evolution. Further-more, specifying the application as a set of platform-independent components, designers ensure application reusability, enabling the use of different HW resources.

The paper has been funded by the European FP7-ICT247999-COMPLEX and the Spanish MCIN TEC2011-28666-C04-02 DREAMS

UML models are one of the most typical techniques to perform CBD [1]. However, the mapping of these models to specific platforms is not an easy task. In order to reduce the platform knowledge required by the designers and to improve reuse, different HW-independent APIs (Application Programming Interface) have been developed. Operating System APIs such as POSIX, provide certain generality, but are limited to the processors controlled by the same OS. Other middleware APIs, such as MPI [14] or CORBA [19], have focused on widely distributed systems or specific application domains. However, these message-passing interfaces, typically limits the semantics that can be associated to inter-component communications, limiting system concurrency.

To solve that, the Multicore Association has developed a more adaptable API, called the Multicore Communications API (MCAPI). MCAPI is scalable and can support virtually any number of cores, each with a different processing architecture and each running the same or a different operating system, and even supporting mappings to complex-to-use resources, such as DSPs or FPGAs. Additionally, the API includes features for concurrency and synchronization support.

In this context, this paper proposes a code generation methodology capable of automatically implementing component-based systems described in UML. The methodology also enables the user to select a wide variety of communication semantics, using the platform-provided APIs to implement them. The approach supports directly mapping to a Linux OS and to MCAPI, in order to provide a platform-independent approach.

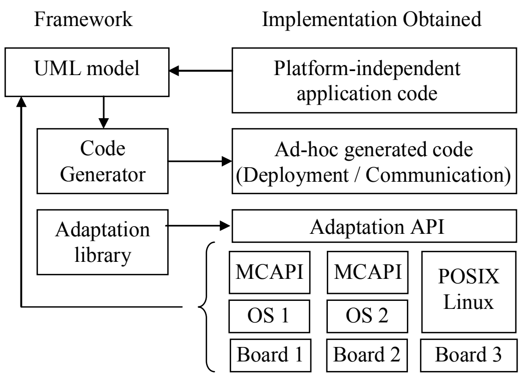

Figure 1: Support of multiple APIs

That way, the automatic synthesis processes can improve the evaluation and implementation steps, reducing designer effort time. The migration from one communication semantic to other is fast and easy, enabling a fast exploration process on the target platform where the application is allocated. Additionally, evaluation of different mapping alternatives is also possible, especially on heterogeneous or distributed systems. With simple changes in the model, different communication mechanisms are synthesised.

Finally, using synthesis tools also reduces the learning time of the engineer for using specific HW platforms.

2. STATE OF THE ART

The approach proposed applies code generation to maximize the benefits of combining of high-level modeling with different SW platform APIs.

On the one hand, model driven design methodologies are being commonly adopted to handle large functionalities. Latest design methodologies start from high-level UML models [11] combined with algorithmic codes (e.g. C, C++, etc.) of the different system components as a solution to reduce SoC design effort [7].

However, pure UML lacks of many of the concepts required to efficiently handle embedded designs. To solve this limitation, the OMG has proposed a standard called MARTE [12], which is oriented to enable designers to include all the information related to embedded, real-time system designs in the UML models [5].

However, most of times synthesizing the gap from the model to the implementation could be a problem [4]. In this context, code synthesis is currently a significant issue [5]. Focusing on homogeneous systems, [10] proposes an approach to control the data flow using different mechanism to handle the system threads between the different cores. [9] presents a framework for communications between heterogeneous devices, exploiting different, specific mechanisms for different devices and platforms. And [7] proposes the inclusion of multiple communication semantics to the communication channels. However, the use of specific SW platform mechanisms results in low portability results.

As a result, the interest of combining both portable APIs with high-level models is been increasing. In this context, [3] presents a framework for task communication through a method based on RMI protocol. [6] synthesize multitasking and the communication between software components of the system from a description in UML-ESL for virtual platform simulation. And [8] proposes a synthesis methodology for communication at transaction level synthesizing high level peer-to-peer communications.

However, all these code-generation approaches only focus on one API and one communication semantic. To overcome this limitation, this paper presents an alternative that enables describing in the UML model a wide set of communication semantics, enabling the selection of different APIs for their implementation. This dramatically increases the solution portability, providing more flexibility and support for different real platforms.

3. MULTICORE ASSOCIATION APIS

The Multicore Association is a grouping of industry companies [13] implementing multicore products that have defined a platform-agnostic API called MCA API. The API proposes a platform-independent interface, enabling abstraction from the target platform and ensuring portability.

The MCA API aims to solve the lacks of other APIs. Regarding to POSIX, MCA API spans beyond a single OS instantiation. Concerning OpenMP lacks, MCA API provides messaging applicable to both symmetric and asymmetric multi-processors. And respect to MPI, MCA API goes beyond, covering task creation and resource synchronization.

Entering in detail, the API is, in fact, divided in a set of three interfaces covering concurrency (MTAPI), synchronization (MRAPI) and communication (MCAPI)..

MCAPI provides a set of functions for communicating separate components running in the same or different platforms. MRAPI provides application-level primitives for synchronizing services and enables coordinating access to shared resources. In addition, concurrent applications need to generate task, in this way MTAPI provides a set of functions for developing task parallelism. Thus, MCAPI also relies on MRAPI and MTAPI to handle concurrency related to communications.

The APIs are scalable for any number of cores. For example, in [16] MCA API is deployed on the SCC, a 48 core concept vehicle for future many-core systems. Additionally, an extension of MCA API is shown in [21]. This extension enables inter-core communication.

Additionally, these APIs also cover heterogeneous multicore-architectures since some of the companies involved works developing codes for DSPs and Hardware accelerators. In [20], its application to map components to DSPs is presented. Additionally, in [15], its application to FPGAs is demonstrating, showing that its use aims 40% of performance improvement.

4. PROPOSED FLOW

This paper proposes an approach oriented to automatically generate the infrastructures required for deploying and communicating the application components in real target platforms.

The proposed approach starts from a platform-independent model of the system (Figure 2). In this model, the functionality is structured following a component-based methodology where each component provides and requires services. At the same time, the user must provide platform-independent application codes (in C/C++) with the functionality of all the application components.

Once the UML/MARTE model is complete, the information captured is automatically annotated in a set of XML files. These XML files are used as inputs by the code generator to create the code required to deploy and communicate the components functionality, which is provided by the user (Figure 2). The graphical tool used to create the UML/MARTE model is Papyrus [17]. A code generator has been developed as a set of generation templates written in the standard MTL language. The development has been done through Acceleo [18], a code generation framework fully integrated in Eclipse.

Then, the framework generates all the deployment code, including the communication infrastructure, the concurrency support, and the insertion of drivers, when needed. Additionally, makefiles are generated to enable automatic execution of the compilation processes.

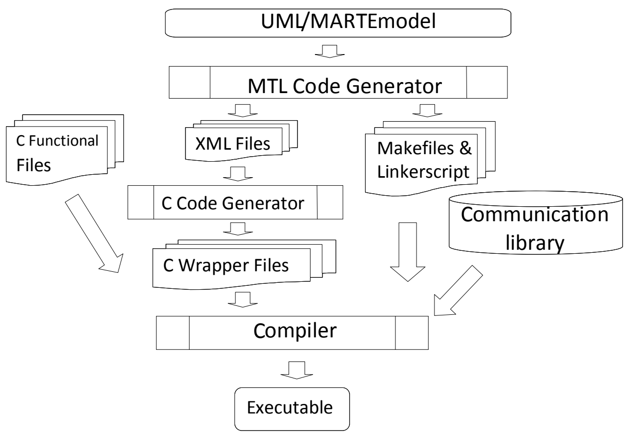

Figure 2: Proposed Synthesis Flow

Concurrency and communication issues must not be included in the user-provided code, being covered in the UML model, associated to channels and allocations. This maximizes component reusability.

Considering these channel semantics and resource allocations, the framework automatically generates the code required to deploy and interconnect the different components in the target platform (Figure 1).

Additionally, it is possible to associate a wide set of semantics to the channels used to interconnect the application components. That way, it is possible to specify and evaluate different implementation alternatives. For example, determining when it is better to use synchronous or asynchronous calls, integrating FIFOs or splitting data to support parallel operations can determine the success of a project. As a result, it is possible to increase the concurrency of the final system just modifying the UML model, in order to optimize the use of the target platform, as proposed in [7].

In addition to that, the methodology presents new specification capabilities that enable designers to select the SW platform API used for generating the required threads or for connecting the application components. The APIs considered are MCAPI and POSIX.

The user can combine these alternatives in the UML model depending on the platform availability, and then the infrastructure generates the final implementation considering the components mapping. Communication-related files are created providing the channel semantics resulting from the combination of parameters described in the UML/MARTE model. Thus, for each communication channel, depending on the allocation of client and server (memory spaces allocations and HW/SW platform allocation), and the semantics of the associated channel, quite different implementations are obtained.

Additionally, the generated code integrates concurrency support. It includes creating and synchronizing the threads required to provide the concurrency captured in the model, as described in next sections.

4.1. UML/MARTE MODELING

The ULM/MARTE methodology followed in this work proposes the specification of the system as a set of application components interconnected by channels and communicated by using service calls encapsulated in interfaces. Then, the application components are grouped in memory spaces for their later HW/SW platform allocation.

Thus, application components are modeled first, using the MARTE stereotype <<RtUnit>>. These components include their communication ports and the "static" threads, which are launched once, at the beginning of the execution. These threads are modelled by the MARTE stereotype <<SwSchedulableResource>>.

In addition to that, application components can be characterized by a pool of threads that enable the dispatching of multiple concurrent calls to the services provided. The size of this pool defines how many incoming petitions can be handled in parallel.

By instantiating the application components, the entire application structure is composed (Figure 3). The component interconnections are performed through service calls which are encapsulated in interfaces (modeled by the MARTE stereotype <<ClientServerSpecification>>). Here, the services are characterized by their name and parameters.

The application components are connected by using the new stereotype <<Channel>> (Figure 3) that capture specific semantics in order to define the resulting communication and concurrency behaviour of the application.

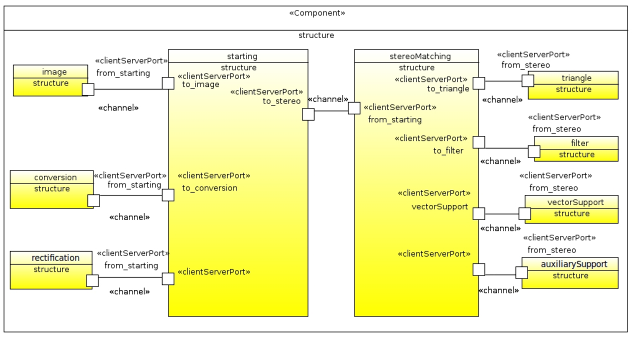
«Component»
structure
image structure
conversion structure
rectification structure
«clientServerPort» from_starting
«channel»
starting structure
«clientServerPort» to_image
«clientServerPort» to_stereo
«clientServerPort» to_conversion
«clientServerPort»
stereoMatching structure
«clientServerPort» from_starting
«clientServerPort» to_triangle
«clientServerPort» to_filter
«clientServerPort» vectorSupport
«clientServerPort» from_stereo
triangle structure
filter structure
vectorSupport structure
auxiliarySupport structure

Figure 3: Application structure

The channels used to connect the application components include well-defined communication semantics, which involves different concurrent effects. A *Channel* element enables capturing well-defined communication semantics. *Channel* semantics are defined by the type of service calls (blocking/non-blocking), the buffer capacity for storing service calls and the timeout and priority associated to a service call, following the proposal in [7].

According to the previous channels characteristics and the capability of the application components for dispatching multiple, concurrent service calls, different concurrent implementations are obtained.

Furthermore, the Channel has implementation-specific information. A channel specification can denote which communication libraries can be used for the channel synthesis. Current support covers MCAPI and POSIX.

Then, application components are mapped to memory spaces (modeled by the MARTE stereotype <<MemoryPartition>>). These mappings are captured as UML abstractions specified by the MARTE stereotype <<Allocate>>, as can be seen in Figure 4.

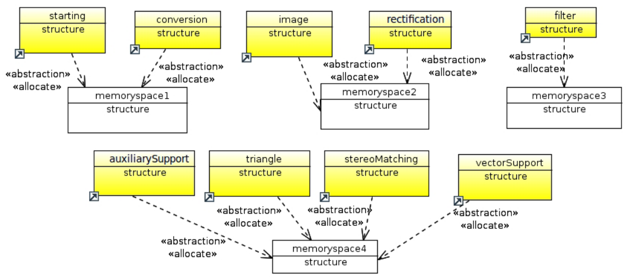

Figure 4: Memory spaces allocation

The HW/SW platform is finally modelled including OSs and the processing resources. The OSs are modeled by the new stereotype <<OS>> which includes information about the solutions available for channel implementation. This enables checking that the options selected for implementing each channel are really supported by the SW infrastructure where they are mapped. Additionally, networks and the network interfaces can be included in the model, with the <<HwMedia>> and <<HwEndPoint>> MARTE stereotypes. The latter elements include the IP address required for making the connections.

Finally, in order to support the automatic makefile generation, the model requires information about the compilers and compilation flags of each specific HW processor (modeled by the MARTE stereotype <<HwProcessor>>). This information is annotated in a UML constraint linked to the *HwProcessor* component, containing the variables: $cc_compiler, $cxx_compiler, $cflags and $lflags. That way, it is possible to set flags for using specific libraries, such as MCAPI library versions.

5. CODE GENERATION

The code generation process receives the information in the XML files resulting from the UML/MARTE model, and creates the code with the entire infrastructure to be compiled together with the user-provided, functional code. As a result, the final implementation binaries are obtained.

This generated infrastructure performs the component instantiations, the channel initializations, the concurrency management and communications. The infrastructure is generated to run the platform-independent code on the specified platforms.

To do so, the generator combines function interface information with channel semantics, component allocation and provided SW platform services. In order to simplify the process, the infrastructure is divided in two parts (Figure 1). First, the framework includes a library with a set of functions that provides the communication and required services in a platform-independent way. The library implements the services depending on the API provided by the actual platform, but SW platform these details are hidden for the code generator.

Then, the code generator can create the deployment codes considering interface information, channel semantics, and component allocation to memory spaces.

The generation of the code depending on allocations, channel semantics and selected APIs is a complex task. To solve it, the proposed approach generates code based on a three-layer solution (Figure 5).

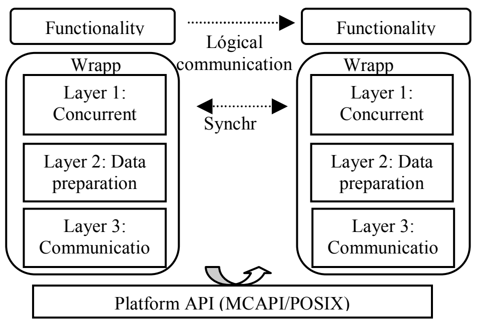

Figure 5 Three layer solution

Then, depending on the information specified in the UML/MARTE model, the synthesis tool creates the codes required for deployment and communication, based on these layers. This flow provides versatility on the application possibilities. In this way, a wide exploration of communication possibilities can be performed in the application with minimum re-engineering effort and time, since it only requires working with the UML model.

To do so, first layer is in charge of describing the concurrency generation and synchronization. This layer injects the channel semantics and the other properties that can produce the concurrent behaviour of the application. In addition to that, the layer uses mechanisms of synchronization to keep the application coherence.

The second layer focuses on data transfers and describes the stack management, that is, the packaging of all the information required to be send to the component connected.

Finally, the third layer is in charge of communication transference mechanisms, sending and receiving packages in an ordered way. To do so, the packaging performed in layer 2 is critical. Since it is independent from the channel semantics and the communication mechanism, it enables managing the generation of both parts in an orthogonal way, supporting a full set of combinations.

5.2 PLATFORM API

The generation of these codes, highly depends on the services provided by the SW platform where components are going to be mapped. The POSIX and MCAPI APIs are used to provide communication (layer 1) and concurrency support (layer 3), depending on the capabilities of the real platform.

5.2.1 POSIX OS

The first alternative covered is to use the services provided by the underlying OS to implement the infrastructure. In that context, POSIX API has been selected, since it is quite common in multiple systems, such as those including Linux OSs.

Additionally, it enables implementing the concurrent elements of the infrastructure. The generation of concurrency is enabled using POSIX thread creation. as required by channel, Synchronization is performed using POSIX mutexes. This synchronization is required to perform the blocking defined by channels semantics, the blocking derived from interface access or the resources availability.

Additionally, using this API, all the functions related to communication can be implemented in several ways. There is a *sending request* function for FIFO implementation, socket implementation, shared memory, etc.

5.2.2 Multicore Association APIs

The use of MCA API for SW synthesis also affects the first and third layer. Task creation is performed using MTAPI [13]. When a channel needs to create a new task for attending a service the code generates a task using MTAPI interface. A task first is created with the function to run (*mtapi_register_impl*) and then the task starts (*mtapi_create_task*). The finalization of the task is performed with *mtapi_task_wait* if it is need.

Then tasks have to be synchronized ensure correct resource access and operation order, so MRAPI plays an important role providing mutex primitives to limit the access of a channel resource when for example a task needs to send a request. Using *mrapi_mutex_lock* these accesses can be protected. Also these mutexes can be used to perform the blocks derived from channel semantics and interface protection. Additionally, all these MCA API resources are initialized and destroyed with the proper functions (i.e.:*mrapi_mutex_init*, *mrapi_mutex_destroy*).

The comm unications included in third layer has been implemented using MCAPI. Their services provide the client-server communication mechanisms and encapsulate data exchanging using its message passing functions.

MCAPI provides three kinds of communication modes: messages, packets, and scalars. But the code generator only uses message passing, since are more flexible and allows the information exchange between nodes. Following MCAPI terminology, each component is treated as a node and each channel end as an endpoint. Thus, channels are created instantiating their endpoints through *mcapi_endpoint_create and mcapi_endpoint_get*.

The C code generator, which is independent of the API selected, also requires the same services than in the POSIX implementations, which are *sending request, receive_request, sending response* and *receive response*. However, in that case, these functions are implemented with the MCAPI message passing functions *mcapi_msg_send* and *mcapi_msg_recv*.

Finally, the generator takes advantage of the API primitives and data types for particular error management and debugging.

6. USE CASES AND RESULTS

The methodology presented was applied to a Stereovision use case. This use case starts with two images initially rectified. Then, both images are compared and the position of the different objects of the image with respect of the observer is extracted. Then, a classical two-frame stereo matching algorithm starts, computing a dense disparity map from the pair of images.

The images can be processed sequentially or in parallel, depending on the application model and the channel semantics. Even though, this work focuses on communication, and not in the benefit of defining different concurrency structures, which follows the work in [7].

Following the methodology proposed, the application model is composed by several components as shown in Figure 3. Therefore, different communication configurations can be applied between the components to communicate them according to the memory spaces structure and the application component mapping on the memory spaces.

In addition, different memory space structures were applied on different target platforms, to demonstrate the difference of applying each implementation on different target platforms. Figure 6 represents the allocation of memory spaces on the Beagle board.

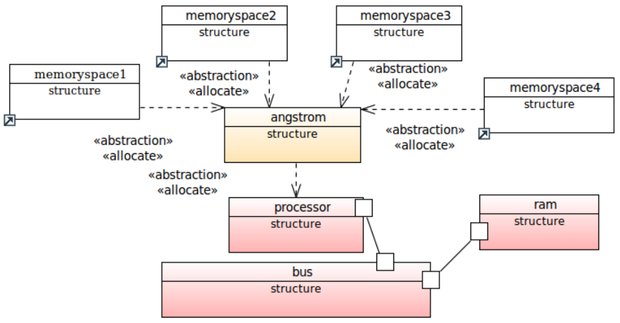

Figure 6: Memory space allocation on Beagleboard

The platforms used to test the results were a dual core, Intel-based platform, a Beagleboard, a Pandaboard and a SabreLite board. Beagleboard is a platform based on TI OMAP-3 chip with one ARM cortex-A8 core at 1Ghz. Pandaboard is a platform based on TI OMAP-4 chip with a SMP of two ARM cortex-A9 cores at 1 GHz. The last platform is based on IMX6 chip with a SMP quad core ARM cortex-A9 at 1 GHz.

In order to demonstrate the generation tool, different experiments were applied considering this use case. For such purpose, the size of the images processed is 1024x768 in all the examples to not alter the results and the functional code is the same, the only change is in the communication mechanism synthesized.

In a first set of experiments, the system has been grouped in a single memory space. Then, Table 2 shows the results using direct service calls (case A), argument passing through Linux FIFOs (case B) and MCA API support (case C).

Case	**Intel**	**Beagle**	**Panda**	**IMX6**
A. Direct call	4.129	25.970	23.204	22.370
B. Linux FIFO	4.140	26.290	23.162	22.470
C. MCA API	19.123	234.60	94.768	49.70

Table 1: One memory space measures in seconds

From these experiments, we can conclude that, while POSIX services imply few overhead in the channel implementation, the MCA API provided obtain worse results. Thus, MCA API is only recommended for those resources mapped to different chips or heterogeneous resources, while the OS API should be used within components in the same OS.

To explore that, experiments mapping components to two memory spaces, and limiting the use of MCA API inter process communications have been performed. Additionally, different channel semantics have been applied on both APIs on the IMX6 board. Channels configuration enabled a pipeline (cases O, P, and Q) from the sequential configuration (case N). The difference among pipeline configurations is the number of concurrency resources on each component, case O has one resource, case P has two and case Q has four. Table 5 shows the results obtained.

CASE	POSIX	MCAPI
N. Sequential	141.66	147.56
O. Pipe_1	131.25	135.49
P. Pipe_2	70.74	76.30
Q. Pipe_4	54.83	95.70

Table 2: Different channel semantics on IMX6 (sec)

As can be seen, this application of MCA API is more adequate. Additionally, the increase in the number of piped images always improves system performance in POSIX, while, in the MCA API case, too much stages result in performance reduction, due to data management overheads.

7. CONCLUSIONS

The paper presents a toolkit for SW code synthesis. The process starts capturing in UML/MARTE models the application structure, communication mechanism, the memory spaces structure and application allocation and HW/SW platform specification and memory spaces allocation. Additionally, the model contains information about the API to be used for the channel implementations: MCAPI or the POSIX API for Linux-like systems.

The toolkit extracts automatically the information required for communications implementation from the UML/MARTE model. A synthesis tool generates the entire needed infrastructure to implement the communications. Automating this process the effort time of engineering can be reduced. Thus, the toolkit enables an easy, fast migration from one communication implementation to other, enabling the exploration for the best system implementation.

The entire toolkit has been applied to a stereovision use case on different target platforms. The obtained results show how the toolkit enables to capture and synthesize different system configurations with different SW platforms.

Results show that the open-source version of MCAPI is not optimal to be used in all component communications. However, the use of optimized versions or its application for heterogeneous or distributed mappings is really interesting.

8. REFERENCES

[1] D. C. Schmidt, “Model-driven Engineering” IEEE Computer, 2006.

[2] C. Szyperski, Component Software: Beyond Object-Oriented Programming. 2nd ed. Addison-Wesley Professional, 2002.

[3] P.A. Hartmann, K. Gruttner, P.Ittershagen, A. Rettberg “A framework for generic HW/SW communication using remote method invocation”. ESLSyn, 2011.

[4] F. Mischkalla, D. He, W. Mueller: “Closing the gap between UML-based modeling, simulation and synthesis of combined HW/SW systems”. DATE, 2010.

[5] É. Piel, R. Atitallah, P. Marquet, S. Meftali, S. Niar, A. Etien, J.-L. Dekeyser, P. Boulet: "Gaspard2: from MARTE to SystemC Simulation", proc. of the DATE'08 workshop, 2008.

[6] T.Cardoso, E. Barros, B.Prado, A. Aziz: “Communication software synthesis from UML-ESL models”. SBCCI, Brasilia 2012.

[7] P. Peñil, H. Posadas, A. Nicolás, E. Villar, "Automatic synthesis from UML/MARTE models using channel semantics" ACES-MB 2012.

[8] N. Hatami, M. Indaco, P. Prinetto, G.Tiotto “Communication interface synthesis from TLM 2.0 to RTL”. Proceedings of IEEE East-West Design and Test Symposium, EWDTS’10.

[9] V. Aggarwal, G. Stitt, A. George, C. Yoon “SCF: A framework for task-level coordination in reconfigurable, heterogeneous systems”. ACM Transactions on Reconfigurable Technology and Systems. Volume 5, Issue 2, June 2012, Article number 7.

[10] C.-S. Peng, L.-C. Chang, C.-H. Kuo, B.-D. Liu “Dual-core virtual platform with QEMU and SystemC”, ISNE 2010.

[11] OMG: “UML profile for system on chip (SoC) specification”, 2006.

[12] "UML Profile for MARTE", www.omgmarte.org, 2009.

[13] Multicore Communications API (MCAPI). Specification V2.015. http://www.multicore-association.org/workgroup/mcapi.php

[14] MPI website http://www.open-mpi.org/

[15] L. Matilainen, E. Salminen, T.D. Hämäläinen “MCAPI abstraction on FPGA based SoC designen”. 9th FPGAworld Conference, 2012.

[16] C. Clauss, S. Pickartz, S. Lankes, T.Bemmerl “Towards a Multicore Communications API implmentation (MCAPI) for the Intel Single-Chip Cloud Computer”. ISPDC, 2012.

[17] Papyrus website: http://www.papyrus.org

[18] Acceleo website: http://www.acceleo.org November 2010.

[19] http://www.corba.org/

[20] T. Gribb, “Simplifying Multicore Migration”, http://eecatalog.com/ dsp/2011/10/18/simplifying-multicore-migration/

[21] S. Miura, T. Hanawa, T. Boku, M. Sato “XMCAPI: Inter-core communication iinterface on multi-chip embedded systems”. Int. Conference on Embedded and Ubiquitous Computing, 2011.