Unified Component Integration Flow for Multi-Processor SoC Design and Validation

Mohamed-Anouar Dziri*, W. Cesário*, Flávio R. Wagner**, and A.A. Jerraya*
* TIMA Laboratory – 46, av. Félix Viallet - 38031 Grenoble – France
** UFRGS – Instituto de Informática – Porto Alegre – Brazil

Abstract

Most system-on-Chip (SoC) design methodologies promote the reuse of pre-designed (hardware, software, and functional) components. However, as these components are heterogeneous, their integration requires complex interface sub-systems. These sub-systems can also be constructed by assembling pre-designed basic interface components. Hence, SoC design and validation involves component composition techniques to create hardware, software, and functional interface sub-systems by assembling basic interface components. We propose a unified methodology for automatic component integration that allows designers to reuse pre-designed components effectively. We also present ROSES, a design flow that uses this methodology to generate hardware, software, and functional interface sub-systems automatically starting from a system-level architectural model.

1. Introduction

SoC design must rely on pre-designed or third party components (e.g., processors, big macro-cells, embedded RTOS, etc.) due to always increasing time-to-market pressures. Components obtained from different providers, and even those designed by different teams of the same company, may be heterogeneous on several aspects: design domains, interfaces, abstraction levels, granularities, etc. Thus, building global synthesis and simulation models is becoming more and more difficult as it requires interfacing complex heterogeneous sub-systems. Consequently, many design bottlenecks are created as the different design teams--for software, hardware, and test--are not able to work concurrently.

Component integration techniques for different design domains have been proposed. For instance, the VxWorks [1] embedded systems' RTOS from Wind River Systems can be adapted to different applications using tedious manual configuration. IBM's Coral framework [2] abstracts the interface of hardware blocks and automates interface generation for its CoreConnect protocol. Sonics [3] provides wrappers to adapt the bus-independent OCP component interface protocol to its μNetwork bus.

Most existing component integration approaches are limited to a standardized component or bus interface; some of them propose automation tools that rely on these standards. We propose a new unified methodology for automatic component integration that allows designers to reuse effectively pre-designed hardware, software, and functional components and is independent of any given standard. Component integration is required:

1. *At the system-level*: to compose heterogeneous (hardware, software, functional) components onto a single global model of the SoC. This requires the generation of complex interface sub-systems. Figure 1 shows the target multi-processor SoC (MPSoC) architectural model. It uses software components (tasks), hardware components (processors and IP cores), interface sub-system components (hardware and software wrappers), and one global on-chip communication network component.
2. *At the interface sub-system level*: to compose basic and homogeneous interface components (e.g. hardware blocks, or software functions) into an interface sub-system component (wrapper).

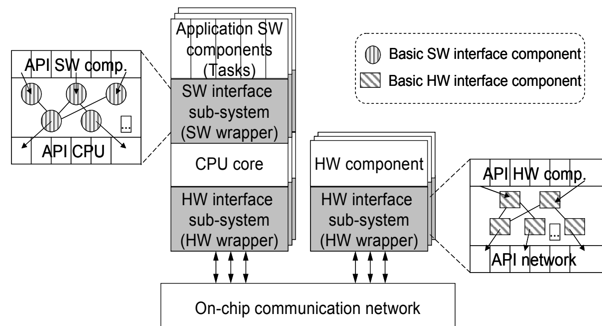

Fig. 1. MPSoC target architectural model

This paper is organized as follows: Section 2 discusses related work. Section 3 introduces different SoC components and models. Section 4 presents the unified component integration methodology. Section 5 introduces ROSES, a design flow that applies the proposed methodology through a case study. Finally, in Section 6, we evaluate the methodology and present our conclusions.

2. Related work

Many standards have been proposed to ease component integration in the software, hardware and simulation

1530-1591/04 $20.00 (c) 2004 IEEE

domains. In the software domain, UML 2.0 [4] is a new standard proposed for modeling software components of reactive systems. Code generation and formal analysis techniques can be used for software (homogeneous) component integration. Although real-time extensions have been incorporated in version 2.0, heterogeneous (hardware and software) component integration methodologies still depend mostly on non-standard extensions of UML.

In the hardware domain, VSIA [5] and OCP-IP [6] propose standards (VCI and OCP, respectively) to ease hardware components (called IP, for “intellectual property”) integration. Compliant components must use a bus-independent standardized interface. Other organizations propose standard interconnects (e.g., bus or on-chip networks) and adapters. In this case, compliant components must be designed according to the interconnect protocol. Sonics [3] uses both approaches, providing wrappers to adapt the bus-independent OCP socket to its μNetwork bus. In some cases, compliance to standards represents a significant overhead since component’s interfaces must support non-relevant functionality only to guarantee their compatibility.

In the simulation domain, SystemC [7] is a C++ library that provides hardware modeling concepts (e.g. time, concurrency, events, bit types, etc.) for simulation of hardware/software systems at different levels of abstraction. SystemC components must use interfaces to call methods implemented in communication channels. Different channel implementations can be used to adapt heterogeneous components. Ptolemy-II [8] provides a heterogeneous system simulation environment that uses formal model-of-computation analysis to govern component interaction. In both cases, for precise modeling of hardware/software interactions (e.g., as provided by executing the software using an instruction-set simulator for the target processor), an ad-hoc setup of a co-simulation environment is required.

In the above paragraphs, only a small fraction of the languages and standards that have been proposed in the software, hardware and simulation domains were discussed. Clearly, the problem is that many de-facto standards exist, coming from different companies or organizations, thus preventing a real interchange of components developed for different sub-standards. This leads to inefficient ad-hoc practices for heterogeneous component integration.

The main contribution of this paper is a unified methodology for automatic integration of heterogeneous components. It allows designers from different design teams to effectively reuse pre-designed components and rapidly build global models for synthesis or simulation by using a unified component integration flow.

3. Component models

3.1. Virtual component model

Abstraction is an essential feature in any reuse technique [9]. Figure 2 shows a virtual component model (VC model) that represents an abstract architecture of the system. This model is composed of several virtual components and an execution environment. A virtual component has an abstract interface that includes information such as provided/required services, control/synchronization, parameters, etc. The execution environment represents an abstract interconnect (e.g. NoC, co-simulation backplane, etc).

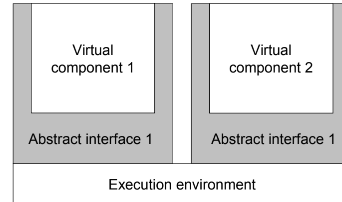

Fig. 2. Virtual component model

We classify components for MPSoC design into three groups (as illustrated in Figure 3):

1. *Software components*: application tasks using high-level, portable code, and software wrappers that adapt them to the hardware part of the system. Software wrappers provide high-level services (e.g. scheduling, I/O, and interrupt handling) and low-level services (e.g. context switching, boot, drivers, etc.).

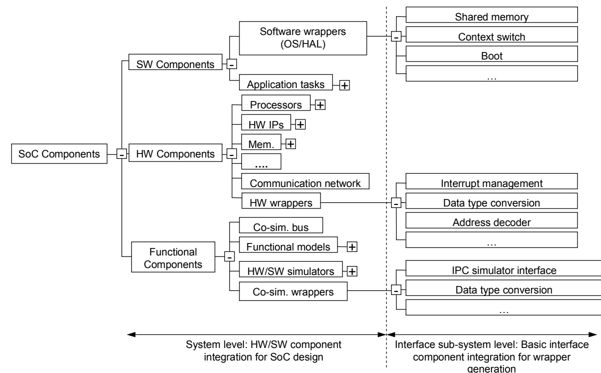

Fig. 3. Component hierarchy for MPSoC design

2. *Hardware components*: processors, hardware IP cores, memories, hardware wrappers, and the on-chip communication network. Hardware wrappers provide services such as interrupt management, data type conversion, etc.
3. *Functional components*: simulation models, functional models, co-simulation wrappers, and a co-simulation bus. The co-simulation bus enables the interaction between different hardware, software, and functional components. Co-simulation wrappers are used to adapt different simulators and different hardware and software component interfaces to the co-simulation bus.

3.2. Interface sub-system models (wrappers)

As illustrated in Figure 4(a), software components (tasks) use an application programming interface (API) to access an abstract communication network. Figure 4(b) shows the structure of the software interface sub-system (SW wrapper) as a stack of layers. The API layer isolates software components from the hardware platform and the other software layers. The next layer contains high-level OS services, such as task scheduling, high-level communication primitives, and interrupt handling. Finally, the hardware abstraction layer (HAL) contains all software that is directly dependent on the hardware platform [10]. For instance, boot code, context-switching code, code for configuration and access to the hardware resources (MMU, bus bridge, timer, etc.), and drivers.

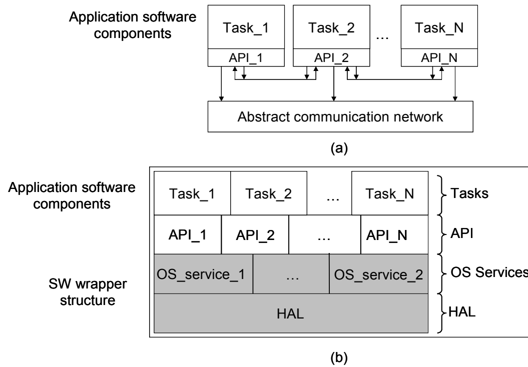

Fig. 4. SW adaptation and SW wrapper structure

Figure 5 shows the structure of the hardware interface sub-system (HW wrapper) used to adapt each processor/IP core to the on-chip communication network. It is composed of reusable basic hardware interface components: processor/IP adapters and communication adapters (CA).

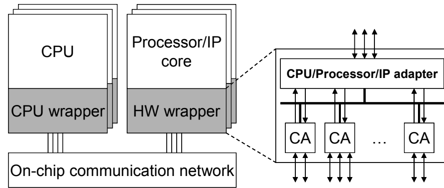

Fig. 5. HW adaptation and HW wrapper structure

Component integration at the system-level may be used to generate two kinds of models: a RTL model for synthesis and a co-simulation model for validation. For the synthesis model, it uses the HW and SW wrappers described above and processor-specific local architecture components. For the co-simulation model, specific co-simulation interface sub-systems (co-simulation wrapper) are used to adapt different simulators and functional components to the co-simulation bus.

Figure 6 shows the architectural model used for co-simulation. The co-simulation wrapper structure is the same as that of the HW wrappers (see Figure 5). Functional components simulate some behavior independently of their implementation in hardware and/or software.

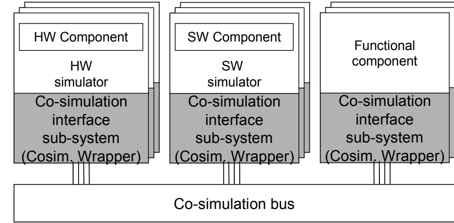

Fig. 6. Generic MPSoC co-simulation model

4. Unified flow for component integration

4.1. Generic component integration flow

Figure 7 shows the overall template for the unified flow for component integration.

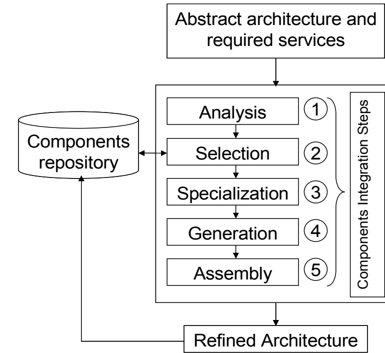

Fig. 7. Unified flow for component integration

Inputs and outputs for the flow are as follows:

- *Input abstract architecture*: it represents the system in terms of interconnected components that provide/require services. It can also be annotated with parameters that guide the integration process.
- *Output architecture*: it is a refined architecture that represents the system after the integration process, may be a global architecture (for prototyping or co-simulation) or an interface sub-system architecture.
- *Components repository*: includes customizable components used for composition. Each component holds a list of provided/required services.

The *component integration process* consists of five elementary steps:

1. *Analysis step*: extracts design information from the abstract architecture to guide the integration process. For instance, information about component interfaces such as required services, protocols, data types and sizes, etc.
2. *Selection step*: locates, compares, and selects components providing required services from a library, according to the information extracted in the previous step. This step must be executed recursively until we identify a suitable (and optimal) set of components providing all required services.

3. *Specialization step*: customizes components selected in the previous step to match service requirements (e.g., data types, bus sizes) and verifies compatibility. Customized values for all parameters are output to configuration files.
4. *Generation step*: takes customizable component code and configuration files and generates specialized source code files that correspond to each component behavior.
5. *Assembly step*: assembles customized components produced in the previous step into a complete refined architecture.

4.2. System and interface integration

Figure 8 shows a component-based MPSoC design flow where each wrapper generator uses an instance of the unified flow presented before (dark shadowed boxes).

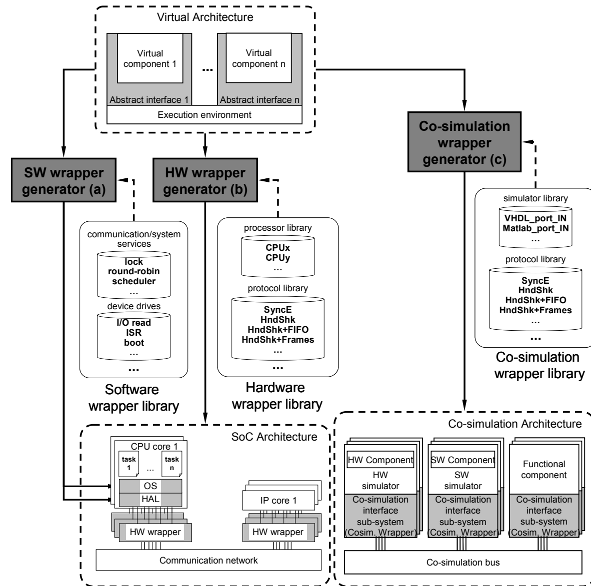

Fig. 8. Component-based MPSoC design flow

For the software part, the flow integrates application software components running on one or several target processors (Figure 8a). This requires building SW wrappers for each processor by composing basic software interface components available in a library. The generated SW wrapper – a specialized RTOS – will adapt the application software components to the hardware part. Binding the application software components to the generated RTOS will produce an executable SW component specific to a selected processor. Processors and IPs must be adapted to the on-chip communication-network, which is achieved through HW wrapper generation (Figure 8b) using a library of basic hardware interface components.

Finally, the unified flow is used by the co-simulation wrapper generator in order to create co-simulation models for system validation (Figure 8c).

5. ROSES: MPSoC design flow

ROSES [11] is a set of tools aimed at the integration of heterogeneous HW/SW IP components for MPSoC design. It generates hardware, software, and co-simulation wrappers to produce refined MPSoC models for synthesis or simulation at the system level. Table 1 summarizes all the models manipulated. The next subsections illustrate these concepts through a subset of a high-bandwidth modem application [12]. Figure 9 shows the VC model for the subset used; it corresponds to a deframing/framing unit (DFU). The DFU has two processors (VM1 and VM2) and the TX_Framer (VM3), which is a dedicated hardware IP component described at the RT-level.

In the rest of the paper, we will deal only with the second processor (VM2) and the IP component (VM3). VM2 includes six sub-modules corresponding to software tasks (T4-T9); they require multipoint communication channels and other sophisticated OS services.

5.1. System-level integration

ROSES starts by capturing design parameters annotated in the DFU virtual architecture (see Figure 9). Two hardware IP cores have been selected: an ARM7 processor and the TX_Framer. The application software has been built by reusing several available SW IP components for implementing tasks T4 to T9.

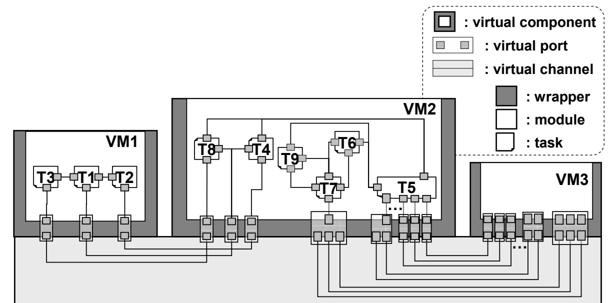

Fig. 9. DFU virtual architecture specification

Figure 10 illustrates the RTL-architecture produced by

		Abstract architecture	Components	Refined architecture
Interface sub-system component Integration	SW wrapper generator	Abstract SW interface	Pipe, Interrupt, Block, Unblock, Boot,…etc.	Software wrapper (OS/HAL)
	HW wrapper generator	Abstract HW interface	Module adapter, Channel adapter (Fifo,…), etc.	Hardware wrapper
	Co-simulation wrapper generator	Abstract functional interface	Simulator adapters (ISS, …), protocols adapters,…etc.	Co-simulation wrapper

Table 1. ROSES component integration models

ROSES. It includes the "concrete" ARM7 local architecture containing additional IP components (local memory, local bus, and address decoder). The HW wrapper implements point-to-point communication between the processor and the IP component. The SW wrapper is composed of a dedicated OS that includes only the functionality required by tasks T4-T9 and a HAL.

For system-level integration, wrappers are treated as components of a library. However, if they do not exist, ROSES can generate them automatically by assembling basic interface components available in a user-extensible library. The next section presents component integration dedicated for wrapper generation.

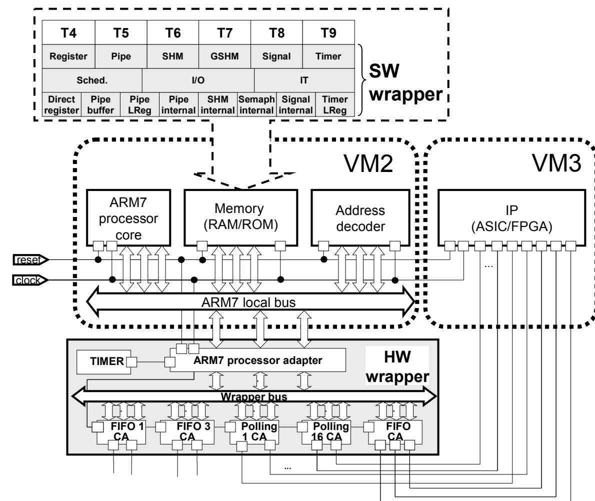

Fig. 10. The generated RTL architecture

5.2. Interface sub-system level integration

Hardware interface sub-systems generator

This generator [13] produces the HW wrapper, which contains a processor adapter that bridges the ARM7 local bus to channel adapters (CA). A CA is generated for each communication protocol used in the virtual architecture.

• *Component library*. The generator uses an extensible library, containing customizable basic hardware interface components, which is organized as follows:

- *A set of "Processor Centric Architectures"* (PCAs) annotated with design parameters such as internal bus type and CPU type. Each PCA has four types of elements: processor cores, local buses, local IP components (such as memory, address decoders, and coprocessors), and processor adapters.
- *A set of "Channel Adapters"* (CAs) corresponding to a list of communication protocols, a list of compatible software communication drivers, and a list of compatible internal buses.

• *Component integration steps*. The HW wrapper generator follows the unified flow illustrated in Figure 7. It analyzes the DFU virtual architecture in order to extract design information such as data type, data size, CA's internal buffer size, and channel's protocol type. Then, the generator performs a search in its library to select an appropriate ARM7 adapter and a CA for each communication protocol. Components are specialized by configuration files that are generated according to the design parameters extracted from the virtual architecture. Each component in the library has its behavior described in customizable macro-code. The generation step configures different macro definitions in order to generate the final source code that describes the hardware component. Finally, the different CAs and the processor adapter are connected to an internal wrapper bus.

Software interface sub-systems generator

This generator [14] produces the SW wrapper, which is a customized multi-task operating system providing several high-level (e.g., task scheduling, interrupt management) and low-level services (e.g., context switching, boot code, drivers).

• *Component library*. The generator uses an extensible library, containing customizable basic software interface components, which is organized into three layers: APIs, communication/system services, and HAL. The library is structured as a dependency graph where nodes correspond to elements and services, and arcs connect each element to its provided and required services. The generated OS avoids including elements that provide unnecessary services. Configurable files (macro files) are also stored in order to give a desired implementation of an element within a given architecture.

Services	Basic interface components
API	Pipe, Memory, Over, Semaphore, Signal
kernel	Cxt, Task, Boot, Schedule
Interrupt	Call, LowInterrupt, HardInterrupt, SoftInterrupt
Synchronization	Block, UnBlock
HAL	Fifo, Shm, Register, LockedRegister, Access

Table 2: Basic SW interface components

• *Component integration steps*. The SW wrapper generator analyzes the DFU virtual architecture to extract design parameters related to the tasks T4-T9. Basic software interface components are then selected from its library according to a dependency relationship (starting from the services required by the tasks). Specialization is performed by using a macro definitions' file for each selected wrapper component. Finally, all software wrapper components are compiled and linked together to produce an application-specific OS using a HAL. Table 2 summarizes the basic software interface components used to generate the OS/HAL for the ARM7 processor.

Simulation interface sub-systems generator

This generator [10] starts by capturing design parameters from the DFU virtual architecture and produces an executable model containing a SystemC simulator that acts as a master for other simulators. Different simulators

(e.g., ISS, SystemC, and VHDL) can be involved in the co-simulation session. The simulation sub-system interface generator automatically generates co-simulation wrappers to adapt the required simulators.

• *Component library*. The co-simulation library includes three different kinds of components:

- C*o-simulation buses:* based on standard signals provided by the SystemC library to model the communication at the RT-level, and on dynamic events to model the communication at the transaction-level.

- *Customizable basic communication interface components*: port adapters, channel adapters, and data type converters. These components are SystemC template object models that can be configured using design information extracted from the system description such as transferred-data type, data size, and characteristics of communication protocol (e.g. FIFO size).

- *Unix inter-process communication*: components for generation of simulator interface sub-systems, supporting various simulators. They are grouped into a UNIX library that supports inter-process communication by using shared memory and semaphores.

• *Component integration steps*. The co-simulation wrapper generator analyzes the virtual architecture and extracts design information such as abstraction level, port directions, communication protocols, and type of data. Then, basic components are selected from the libraries according to this information. Components can be specialized by configuring their template parameters according to design parameter's values. Finally, a global top-level co-simulation netlist for SystemC is generated which assemble together all co-simulation wrappers.

6. Conclusion

This paper has presented a unified methodology that enables heterogeneous components integration for MPSoC design and validation using basic interface components integration for generation of different wrappers. ROSES uses this methodology for the generation of interface sub-systems. Starting from an abstract architecture model, hardware, software and co-simulation wrappers can be automatically generated by assembling basic interface components. Global MPSoC models can then be obtained by using these wrappers to integrate hardware, software, and functional components into a single model. Designers do not need to generate manually any interfacing code. This efficient solution to the component integration problem presents a unique combination of features:

• It implements a unified approach for the component integration that eases the integration of heterogeneous components for MPSoC design and validation at the system level by automating the integration of basic interface components for the generation of different interface sub-systems;

• It provides an environment that is able to accommodate components having different interfaces and granularities and to handle complex interface sub-systems by composing user-extensible open-library components; and

• It provides an automatic process for component reuse and integration, so that designers are free from the tedious and error-prone work of manual wrapper coding for each new architectural solution. Thus, they can concentrate on more critical design problems where their expertise is essential.

Acknowledgments

This research has been sponsored in part by the ToolIP-A511, SpeAC-A508 projects of the Europe's MEDEA+ (Microelectronics for European Applications) program and the French project MERCED/ITEA.

References

[1] VxWorks, http://www.windriver.com
[2] R.A.Bergamaschi, et al., "Automating the Design of SOCs Using Cores", IEEE Design & Test of Computers, Vol. 18, Nr. 5, 2001.
[3] Sonics, Inc. "Sonics µNetworks, Technical Overview", June 2000, available at http://www.sonicsinc.com/
[4] T.Ziadi, B.Traverson, J.M.Jézéquel, "From a UML Platform Independent Component Model to Platform Specific Component Models", Workshop in Software Model Engineering, Germany, 2002.
[5] Virtual Socket Interface Alliance, http://www.vsi.org
[6] Open Core Protocol, http://www.ocpip.org
[7] SystemC, http://www.systemc.org
[8] "The Ptolemy 2 Project, UC Berkeley". Available at: http://ptolemy.eecs.berkeley.edu/
[9] C.W.Krueger, "Software Reuse", ACM Computing Surveys, Vol. 24, Nr. 2, June 1992.
[10] A.Bouchhima, S.Yoo, A.A.Jerraya, "Fast and Accurate Timed Execution of High Level Embedded Software using HW/SW Interface Simulation Model", ASP-DAC, to appear 2004.
[11] W.O.Cesario, et al., "Multiprocessor SoC Platforms: A Component-Based Design Approach", IEEE Design & Test of Computers, Vol. 19, Nr. 6, Nov-Dec 2002.
[12] M.Diaz-Nava, G.S.Okvist, "The Zipper Prototype: A Complete and Flexible VDSL Multi-carrier Solution", ST Journal, September 2000.
[13] D.Lyonnard, S.Yoo, A.Baghdadi, A.A.Jerraya, "Automatic Generation of Application-Specific Architectures for Heterogeneous Multiprocessor System-on-Chip", DAC, Las Vegas, USA, June 2001.
[14] L.Gauthier, S.Yoo, A.A.Jerraya, "Automatic Generation and Targeting of Application Specific Operating Systems and Embedded Systems Software", IEEE Transactions on Computer-Aided Design, Vol. 20, Nr. 11, November 2001.