Models for Co-Design of Heterogeneous Dynamically Reconfigurable SoCs

Jean-Luc Dekeyser, Abdoulaye Gamatié, Samy Meftali, Imran Rafiq Quadri

Chapter 1

MODELS FOR CO-DESIGN OF HETEROGENEOUS DYNAMICALLY RECONFIGURABLE SOCS

Jean-Luc Dekeyser,[1] Abdoulaye Gamatié,[1] Samy Meftali,[1] and Imran Rafiq Quadri[1]

[1]*INRIA Lille Nord Europe - LIFL - USTL - CNRS*
Parc Scientifique de la Haute Borne, Park Plaza - Batiment A
40 avenue Halley,
59650 Villeneuve d'Ascq, FRANCE
{Firstname.LastName}@lifl.fr

Abstract The design of Systems-on-Chip is becoming an increasing difficult challenge due to the continuous exponential evolution of the targeted complex architectures and applications. Thus, seamless methodologies and tools are required to resolve the SoC design issues. This chapter presents a high level component based approach for expressing system reconfigurability in SoC co-design. A generic model of reactive control is presented for Gaspard2, a SoC co-design framework. Control integration in different levels of the framework is explored along with a comparison of their advantages and disadvantages. Afterwards, control integration at another high abstraction level is investigated which proves to be more beneficial then the other alternatives. This integration allows to integrate reconfigurability features in modern SoCs. Finally a case study is presented for validation purposes. The presented works are based on Model-Driven Engineering (MDE) and UML MARTE profile for modeling and analysis of real-time embedded systems.

Keywords: **SoC co-design, Component based approach, System adaptivity, Reactive control, Model-Driven Engineering, MARTE, UML, FPGAs, Partial dynamic reconfiguration**

1. Introduction

Since the early 2000s, Systems-on-chip (or SoCs) have emerged as a new paradigm for embedded systems design. In a SoC, the computing units: programmable processors; memories, I/O devices, etc., are all integrated into a single chip. Moreover, multiple processors can be integrated into a SoC (Multiprocessor System-on-Chip, MPSoC) in which the communication can be achieved through Networks-on-Chips (NoCs). Some examples of domains where SoCs are used are: multimedia, automotive, defense and medical applications.

SoC complexity and need of reconfiguration

As the computational power increases for SoCs, more functionalities are expected to be integrated in these systems. As a result, more complex software applications and hardware architectures are integrated, leading to a *system complexity* issue which is one of the main hurdles faced by designers. The fallout of this complexity is that the system design, particularly software design, does not evolve at the same pace as that of hardware. This has become a critical issue and has finally led to the *productivity gap*.

Reconfigurability is also a critical issue for SoCs which must be able to cope with end user environment and requirements. For instance, mode-based control plays an important role in multimedia embedded systems by allowing to describe Quality-of-Service (QoS) choices: 1) changes in executing functionalities, e.g., color or black and white picture modes for modern digital cameras; 2) changes due to resource constraints of targeted platforms, for instance switching from a high memory consumption mode to a smaller one; or 3) changes due to other environmental and platform criteria such as communication quality and energy consumption. A suitable control model must be generic enough to be applied to both software and hardware design aspects.

The reduction in complexity of SoCs, while integrating mechanisms of system reconfiguration in order to benefit from QoS criteria, offers an interesting challenge. Several solutions are presented below.

Component based design

An effective solution to SoC co-design problem consists in raising the design abstraction levels. This solution can be seen through a *top-down* approach. The important requirement is to find efficient design methodologies that raise the design abstraction levels to reduce overall SoC

complexity. They should also be able to express the control to integrate reconfigurability features in modern embedded systems.

Component based design is also a promising alternative. This approach increases productivity of software developers by reducing the amount of efforts needed to develop and maintain complex systems [E. 03]. It offers two main benefits. First, it offers an incremental or *bottom-up* system design approach permitting to create complex systems, while making system verification and maintenance more tractable. Secondly, this approach allows reuse of development efforts as component can be re-utilized across different software products.

Controlling system reconfiguration in SoCs can be expressed via different component models. Automata based control is seen as promising as it incorporates aspects of modularity that is present in component based approaches. Once a suitable control model is chosen, implementation of these reconfigurable SoC systems can be carried out via Field Programmable Gate Arrays (or FPGAs). FPGAs are inherently reconfigurable in nature. State of the art FPGAs can change their functionality at *runtime*, known as Partial Dynamic Reconfiguration (PDR) [P. 06]. These FPGAs also support internal self dynamic reconfiguration, in which an internal controller (a *hardcore/softcore* embedded processor) manages the reconfiguration aspects.

Finally the usage of *high level component based design approach* in development of real-time embedded systems is also increasing to address the compatibility issues related to SoC co-design. High abstraction level SoC co-modeling design approaches have been developed in this context, such as Model-Driven Engineering (MDE) [OMG07] that specify the system using the UML graphical language. MDE enables high level system modeling (of both software and hardware), with the possibility of integrating heterogeneous components into the system. *Model transformations* [T. 06] can be carried out to generate executable models from high level models. MDE is supported by several standards and tools.

Our contribution relates to the proposal of a high level component based SoC co-design framework which has been integrated with suitable control models for expressing reconfigurability. The control models are first explored at different system design levels along with a brief comparison. Afterwards, the control model is explored at another design abstraction level that permits to link the system components with respective implementations. This control model proves more beneficial as it allows to exploit reconfigurability features in SoC by means of partial dynamic reconfiguration in FPGAs. Finally a case study is illustrated which validates our design methodology.

The plan of the chapter is as follows. Section 2 gives an overview of some related works while section 3 defines the notions associated with component based approaches. Section 4 introduces our SoC co-design framework while section 5 illustrates a reactive control model. Section 6 compares control models at different levels in our framework. Section 7 provides a more beneficial control model in our framework, illustrated with a case study. Finally section 8 gives the conclusion.

2. Related works

There are several works that use component based high abstraction level methodologies for defining embedded systems. MoPCoM [A. 08c] is a project that targets modeling and code generation of embedded systems using the block diagrams present in SysML which can be viewed as components. In [F. 08], a SynDEx based design flow is presented to manage SoC reconfigurability via implementation in FPGAs, with the application and architecture parts modeled as components. Similarly in [Gra08], a component based UML profile is described along with a tool set for modeling, verification and simulation of real-time embedded systems. Reconfiguration in SoC can be related to available system resources such as available memory, computation capacity and power consumption. An example of a component based approach with adaptation mechanisms is provided in [SA00]; e.g. for switching between resources [BAP05].

In [Lat99],[SKM01], the authors concentrate on verification of real-time embedded systems in which the control is specified at a high abstraction level via UML state machines and collaborations; by using model checking. However, control methodologies vary in nature as they can be expressed via different forms such as Petri Nets [B. 04], or other formalisms such as mode automata [MR98].

Mode automata extend synchronous dataflow languages with an imperative style, but without many modifications of language style and structure [MR98]. They are mainly composed of *modes* and *transitions*. In an automaton, each mode has the same interface. Equations are specified in modes. Transitions are associated with conditions, which serve to act as triggers. Mode automata can be composed together in either in parallel or hierarchical manner. They enable formal validation by using the synchronous technology. Among existing UML based approaches allowing for design verification are the Omega project [Gra08] and Diplodocus [AMAB$^+$06]. These approaches essentially utilize model checking and theorem proving.

In the domain of dynamically reconfigurable FPGA based SoCs, Xilinx initially proposed two design flows, which were not very effective leading to new alternatives. An effective modular approach for 2-D shaped reconfigurable modules was presented in [P. 05]. [J. 03] implemented modular reconfiguration using a horizontal slice based bus macro in order to connect the static and partial regions. They then placed arbitrary 2-dimensional rectangular shaped modules using routing primitives [M. 06]. This approach has been further refined in [C. 08]. In 2006, Xilinx introduced the *Early Access Partial Reconfiguration Design Flow* [Xil06] that integrated concepts of [P. 05] and [J. 03]. Works such as [Bay08],[K. 07] focus on implementing softcore internal configuration ports on Xilinx FPGAs such as Spartan-3, that do not have the hardware Internal Configuration Access Port (ICAP) reconfigurable core, for implementing PDR. Contributions such as introduced in [C. 07] and [A. 08a], illustrate usage of customized ICAPs. Finally in [R. 06], the ICAP reconfigurable core is connected with Networks-on-chip (NoC) implemented on dynamically reconfigurable FPGAs.

In comparison to the above related works, our proposition takes into account the following domains: SoC co-design, control/data flow, MDE, UML MARTE profile, SoC reconfigurability and PDR for FPGAs; which is the novelty of our design framework.

3. Components

Components are widely used in the domain of component based software development or component based software engineering. The key concept is to visualize the system as a collection of *components* [E. 03]. A widely accepted definition of components in software domain is given by Szyperski in [Szy98]:

A component is a unit of composition with contractually specified interfaces and fully explicit context dependencies that can be deployed independently, and is subject to third-party composition.

In the software engineering discipline, a component is viewed as a representation of a self-contained part or subsystem; and serves as a building block for designing a complex global system. A component can provide or require *services* to its environment via well-specified *interfaces* [E. 03]. These interfaces can be related to *ports* of the component. Development of these components must be separated from the development of the system containing these modules. Thus components can be used in different contexts, facilitating their reuse.

The definition given by Szyperski permits to separate the component *behavior* and the component *interface*. Component *behavior* defines the functionality or the executable realization of a component. This can be viewed as associating the component with an *implementation* such as compilable code, binary form, etc.; depending upon the component model. This notion enables to link the component to user defined or third party implementations or *intellectual properties* (IPs). A component *interface* represents the properties of the component that are externally visible to other parts of the system.

Two basic prerequisites permit integration and execution of components. A *component model* defines the semantics that components must follow for their proper evolution [E. 03]. A *component infrastructure* is the design-time and run-time infrastructure that allows interaction between components and manages their assembly and resources. Obviously, there is a correspondence between a component model and the supporting mechanisms and services of a component framework.

Typically, in languages such as *Architecture Definition Languages* (ADLs), description of system architectures is carried out via compositions of hardware and software modules. These components follow a component model; and the interaction between components is managed by a component infrastructure [L. 98].

For describing hardware components in embedded systems, several critical properties, such as timing, performance and energy consumption, depend on characteristics of the underlying hardware platform. These extra functional properties such as performance cannot be specified for a *software* component but are critical for defining a hardware platform.

Component models

A component model determines the behavior of components within a component framework. It states what it means for a component to implement a given interface, it also imposes constraints such as defining communication protocols between components etc. [E. 03]. We have already briefly described the use of components in software engineering. There exit many component models such as COM (Component Object Model), CORBA, EJB and .NET. Each of these models have distinct semantics which may render them incompatible with other component models. As these models prove more and more useful for the design, development and verification of complex software systems, more and more research is being carried out by hardware designers in order to utilize the existing concepts present in software engineering for facilitating the development of complex hardware platforms.

Already hardware and system description languages such as VHDL and SystemC which support incremental modular structural concepts can be used to model embedded systems and SoCs in a modular way.

Component Infrastructure

A component infrastructure provides a wide variety of services to enforce and support component models. Using an simple analogy, components are to infrastructures what processes are to an operating system. A component infrastructure manages the resources shared by the different components [E. 03]. It also provides the underlying mechanisms that allow component interactions and final assembly. Components can be either *homogeneous*: having the same functionality model but not the same behavior; or *heterogeneous*. Examples of homogeneous components can be found in systems such as grids and cubes of computing units. In systems such at TILE64 [ea08], homogeneous instances of processing units are connected together by communication media. These types of systems are partially homogeneous concerning the computation units but heterogeneous in terms of their interconnections. Nowadays, modern embedded systems are mainly composed of heterogeneous components. Correct assembly of these components must be ensured to obtain the desired interactions. A lot of research has been carried out to ensure the correctness of interface composition in heterogeneous component models. Enriching the interface properties of a same component enables in addressing different aspects, such as timing and power consumption [DHJP08]. The semantics related to component assembly can be selected by designers according to their system requirements. The assembly can be either static or dynamic in nature.

Towards SoC co-design

It is obvious that in the context of embedded systems, information related to hardware platforms must be added to component infrastructures. Properties such as timing constraints and resource utilization are some of the integral aspects. However, as different design platforms use different component models for describing their customized components, there is a lack of consensus on the development of components for real-time embedded systems. Similarly interaction and interfacing of the components is another key concept.

Dynamic reconfiguration. Dynamic reconfiguration of component structure depends on the context required by designer and can be determined by different Quality-of-Service (QoS) criteria. The dynamic

aspects may require the integration of a *controller* component for managing the overall reconfiguration. The semantics related to component infrastructure must take into consideration several key issues: instantiation and termination of these components, deletion in case of user requirement etc. Similarly communication mechanisms such as message passing, operation calls can be chosen for inter and intra communication (in case of composition hierarchy) of components.

In case of embedded systems, a suitable example can be of FPGAs. These reconfigurable architectures are mainly composed of heterogeneous components, such as processors, memories, peripherals, I/O devices, clocks and communication media such as buses and Network-on-Chips. For carrying out internal dynamic reconfiguration, a controller component: in the form of a hard/soft core processor, can be integrated into the system for managing the overall reconfiguration process.

4. Gaspard2: a SoC Co-Design Framework

Gaspard2 [DaR09],[A. 08b] is a SoC co-design framework dedicated to parallel hardware and software and is based on the classical Y-chart [D.D83]. One of the most important features of Gaspard2 is its ability for system co-modeling at a high abstraction level. Gaspard2 uses the Model-Driven Engineering methodology to model real-time embedded systems using the UML MARTE profile [OMG]; and UML graphical tools and technologies such as Papyrus and Eclipse Modeling Framework.

Figure 1.1 shows a global view of the Gaspard2 framework. Gaspard2 enables to model *software applications*, *hardware architectures* and their *allocations* in a concurrent manner. Once models of software applications and hardware architectures are defined, the functional parts (such as application tasks and data) can be mapped onto hardware resources (such as processors and memories) via *allocation(s)*. Gaspard2 also introduces a *deployment* level that allows to link hardware and software components with intellectual properties (IPs). This level is elaborated later in section 7.

For the purpose of automatic code generation from high level models, Gaspard2 adopts MDE model transformations (*model to model* and *model to text* transformations) towards different execution platforms, such as targeted towards synchronous domain for validation and analysis purposes [GRY+08b]; or FPGA synthesis related to partial dynamic reconfiguration [QMMD09], as shown in Figure 1.1. Model transformation chains allow moving from high abstraction levels to low enriched levels. Usually, the initial high level models contain only domain-specific

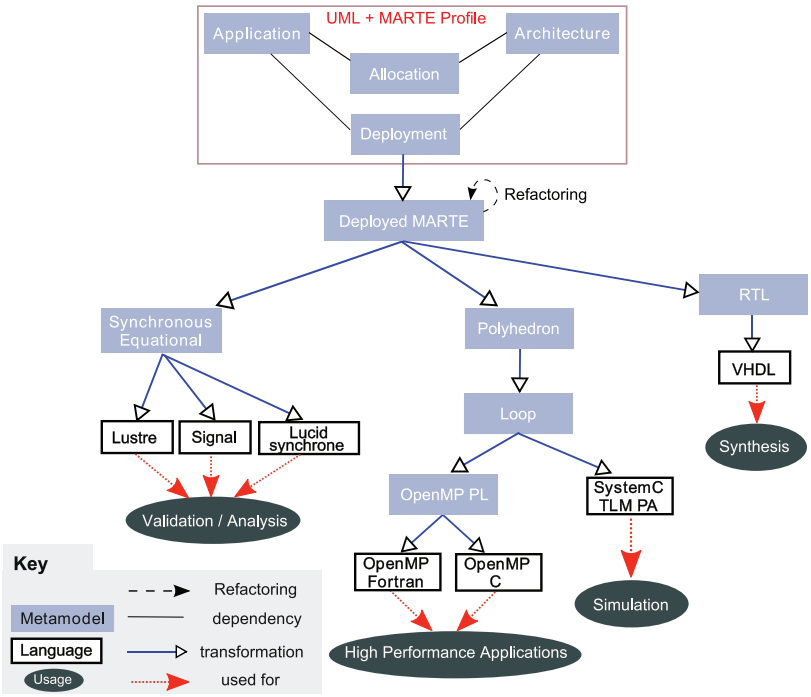

Figure 1.1. A global view of the Gaspard2 framework

concepts, while technological concepts are introduced seamlessly in the intermediate levels.

5. A reactive control model

We first describe the generic control semantics which can be integrated into the different levels (application, architecture and allocation) in SoC co-design. Several basic control concepts, such as `Mode Switch Component` and `State Graphs` are presented first. Then a basic composition of these concepts, which builds the mode automata, is discussed. This modeling derives from mode concepts in mode automata. The notion of exclusion among modes helps to separate different computations. As a result, programs are well structured and fault risk is reduced. We then use the Gaspard2 SoC co-design framework for utilization of these concepts.

Modes

A mode is a distinct method of operation that produces different results depending upon the user inputs. A mode switch component in Gaspard2 contains at least more than one mode; and offers a switch functionality that chooses execution of one mode, among several alternative present modes [O. 05]. The mode switch component in Figure 1.2 illustrates such a component having a *window* with multiple tabs and interfaces. For instance, it has an m (mode value input) port as well as several data input and output ports, i.e., i_d and o_d respectively. The switch between the different modes is carried out according to the mode value received through m.

The modes, M_1, ..., M_n, in the mode switch component are identified by the mode values: m_1, ..., m_n. Each mode can be hierarchical, repetitive or elementary in nature; and transforms the input data i_d into the output data o_d. All modes have the same interface (i.e. i_d and o_d ports). The activation of a mode relies on the reception of mode value m_k by the mode switch component through m. For any received mode value m_k, the mode runs exclusively. It should be noted that only mode value ports, i.e., m; are compulsory for creation of a mode switch component, as shown in Figure 1.2. Thus other type of ports are represented with dashed lines.

State graphs

A state graph in Gaspard2 is similar to state charts [Har87], which are used to model the system behavior using a state-based approach. It can be expressed as a graphical representation of transition functions as discussed in [GRY08a]. A state graph is composed of a set of vertices, which are called *states*. A state connects with other states through directed edges. These edges are called *transitions*. Transitions can be conditioned by some events or Boolean expressions. A special label *all*, on a transition outgoing from state s, indicates any other events that do not satisfy the conditions on other outgoing transitions from s. Each state is associated with some mode value specifications that provide mode values for the state. A state graph in Gaspard2 is associated with a `Gaspard State Graph` as shown in Figure 1.2.

Combining modes and state graphs

Once mode switch components and state graphs are introduced, a `MACRO` component can be used to compose them together. The MACRO in Figure 1.2 illustrates one possible composition. In this component,

the Gaspard state graph produces a mode value (or a set of mode values) and sends it (them) to the mode switch component. The latter switches the modes accordingly. Some data dependencies (or connections) between these components are not always necessary, for example, the data dependency between I_d and i_d. They are drawn with dashed lines in Figure 1.2. The illustrated figure is used as a basic composition, however, other compositions are also possible, for instance, one Gaspard state graph can control several mode switch components [QMD09].

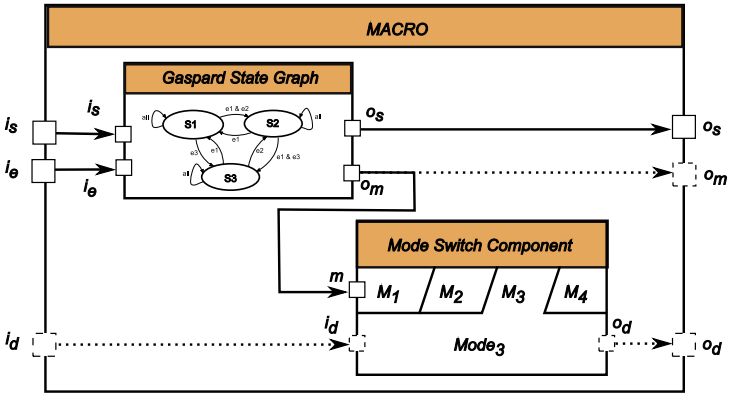

Figure 1.2. An example of a macro structure

6. Control at different system design levels

The previously mentioned control mechanisms can be integrated in different levels in a SoC co-design environment. We first analyze the control integration at the application, architecture and allocation levels in the particular case of the Gaspard2 framework, followed by a comparison of the three approaches.

Generic modeling concepts

We first present some concepts which are used in the modeling of mode automata. Gaspard2 uses the Repetitive Structure Modeling (RSM) package in the MARTE UML profile to model intensive data-parallel processing applications. RSM is based on Array-OL [Bou07] that describes the *potential parallelism* in a system; and is dedicated to data intensive multidimensional signal processing. In Gaspard2, data are manipulated in the form of multidimensional arrays. For an application functionality, both data parallelism and task parallelism can be expressed easily via RSM. A `repetitive component` expresses the data parallelism in an

application: in the form of sets of input and output `patterns` consumed and produced by the repetitions of the interior `part`. It represents a regular scalable component infrastructure. A `hierarchical` component contains several `parts`. It allows to define complex functionalities in a modular way and provides a structural aspect of the application. Specifically, task parallelism can be described using a hierarchical component in our framework.

The basic concepts of Gaspard2 control have been presented in Section V, but its complete semantics have not been provided. Hence, we propose to integrate mode automata semantics in the control. This choice is made to remove design ambiguity, enable desired properties and to enhance correctness and verifiability in the design. In addition to previously mentioned control concepts, three additional constructs as present in the RSM package in MARTE, namely the `Interrepetition` dependency (IRD), the `tiler` connector and `defaultLink` are used to build mode automata.

A `tiler` connector describes the tiling of produced and consumed arrays and thus defines the shape of a data pattern. The `Interrepetition` dependency is used to specify an acyclic dependency among the repetitions of the same component, compared to a tiler, which describes the dependency between the repeated component and its owner component. The interrepetition dependency specification leads to the sequential execution of repetitions. A `defaultLink` provides a default value for repetitions linked with an interrepetition dependency, with the condition that the source of dependency is absent.

The introduction of an interrepetition dependency serializes the repetitions and data can be conveyed between these repetitions. Hence, it is possible to establish mode automata from Gaspard2 control model, which requires two subsequent steps. First, the internal structure of `Gaspard Mode Automata` is presented by the `MACRO` component illustrated in Figure 1.2. The Gaspard state graph in the macro acts as a state-based controller and the mode switch component achieves the mode switch function. Secondly, interrepetition dependency specifications should be specified for the macro when it is placed in a repetitive context. The reasons are as follows. The macro structure represents only a single transition between states. In order to execute continuous transitions as present in automata, the macro should be repeated to have multiple transitions. An interrepetition dependency forces the continuous sequential execution. This allows the construction of mode automata which can be then executed.

Application level

With previous presented constructs, the modeling of Gaspard mode automata, which can be eventually translated into synchronous mode automata [MR98], is illustrated with an example in Figure 1.3, where the assembly of these constructs is presented. An interrepetition dependency connects the repetitions of `MACRO` and conveys the current state. It thus sends the target state of one repetition as the source state for the next repetition of the macro component as indicated by the value of -1. The states and transitions of the automata are encapsulated in the Gaspard state graph. The data computations inside a mode are set in the mode switch component. The detailed formal semantics related to Gaspard mode automata can be found in [GRY08a]. It should be noted that parallel and hierarchical mode automata can also be constructed using the control semantics.

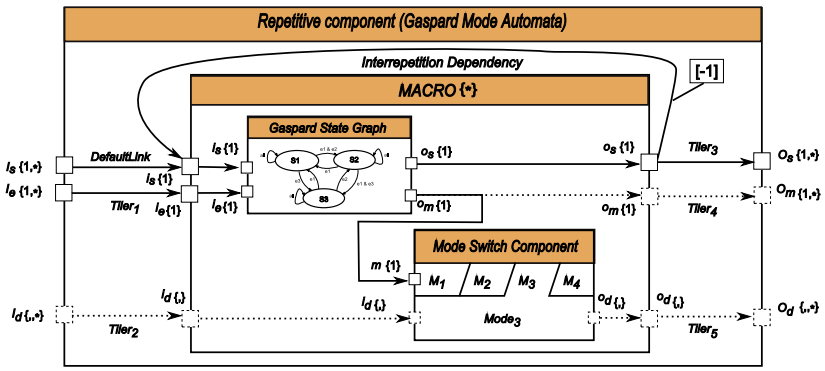

Figure 1.3. The macro structure in a repetitive component

The proposed control model enables the specification of system reconfigurability at the application level [H. 08]. Each mode in the switch can have different effects with regards to environmental or platform requirements. A mode represents a distinct algorithm to implement the same functionality as others. Each mode can have a different demand of memory, CPU load, etc. Environmental changes/platform requirements are captured as events; and taken as inputs of the control.

Architecture level

Gaspard2 uses the *Hardware Resource Modeling* (or HRM) package in the MARTE profile in combination with the RSM package to model large regular hardware architectures (such as multiprocessor architectures) in

a compact manner. Complex interconnection topologies can also be modeled via Gaspard2 [I.-08].

Control semantics can also be applied on to the architectural level in Gaspard2. As compared to the integration of control in other modeling levels (such as application and allocation), the control in architecture is more flexible and can be implemented in several forms. A controller can modify the structure of the architecture in question such as modifying the communication interconnections. The structure can be either modified globally or partially. In case of a global modification, the reconfiguration is viewed as static and the controller is present exterior to the targeted architecture. If the controller is present inside the architecture, then the reconfiguration is partial and could result in partial dynamic reconfiguration. However, the controller can be related to both the structural and behavioral aspects of the architecture. An example can be of a controller unit present inside a processing unit in the architecture for managing *Dynamic frequency scaling* [YHBM02] or *Dynamic voltage scaling* [IKH01]. These techniques allow power conservation by reducing the frequency or the voltage of an executing processor.

Allocation level

Gaspard2 uses the Allocation Modeling package (Alloc) to allocate SoC applications on to the targeted hardware architectures. Allocation in MARTE can be either *spatial* or *temporal* in nature [OMG].

Control at the allocation level can be used to decrease the number of active executing computing units to reduce the overall power consumption levels. Tasks of an application that are executing parallely on processing units may produce the desired computation at an optimal processing speed, but might consume more power, depending upon the inter-communication between the system. Modification of the allocation of the application on to the architecture can produce different combinations and different end results. A task may be switched to another processing unit that consumes less power, similarly, all tasks can be associated on to a single processing unit resulting in a temporal allocation as compared to a spatial one. This strategy may reduce the power consumption levels along with decrease in the processing speed. Thus allocation level allows to incorporate *Design Space Exploration* (DSE) aspects which in turn can be manipulated by the designers depending upon their chosen QoS criteria.

Comparison of control at the three levels

Integrating control at different aspects of system (application, architecture and allocation) has its advantages and disadvantages as briefly shown in the Figure 1.4. With respect to control integration, we are mainly concerned with several aspects such as the range of impact on other modeling levels. We define the impact range as either *local* or *global*, with the former only affecting the concerned modeling level while the later having consequences on other modeling levels. These consequences may vary and cause changes in either *functional* or *non-functional aspects* of the system. The modification in application may arise due to QoS criteria such as switching from a high resolution mode to a lower one in a video processing functionality. However, the control model may have consequences, as change in an application functionality or its structure may not have the intended end results.

Control integration in an architecture can have several possibilities. The control can be mainly concerned with modification of the hardware parameters such as voltage and frequency for manipulating power consumption levels. This type of control is local and mainly used for QoS, while the second type of control can be used to modify the system structure either globally or partially. This in turn can influence other modeling levels such as the allocation. Thus allocation needs to be modified every single time when there is a modification in the structure of the execution platform.

	Impact on other models	Conditions	Consequences
Application	Local	—	Change in application functionality/structure occurs, the functionality can be related to QoS criteria as well
Architecture	Local	QoS variation only	Variability in observed performance
	Global	Modification in structure	Allocation model needs to be modified as well
Allocation	Local	Other models are fixed	Variability in observed performance
	Global	Other models modifiable	Change in functionality, variation in QoS possible

Figure 1.4. Overview of control on the first three levels of a SoC framework

Control at the allocation is local only when both the application and architecture models have been pre-defined to be static in nature which is rarely the actual scenario. If either the application or the architecture is changed, the allocation must be adapted accordingly.

It is also possible to form a merged control by combining the control models at different aspects of the system to form a mixed-level control approach. However, detailed analysis is needed to ensure that any combination of control levels does not cause any unwanted consequences. This is also a tedious task. During analysis, several aspects have to be

monitored, such as ensuring that no conflicts arise due to a merged approach. Similarly, redundancy should be avoided: if an application control and architecture control produce the same result separately; then suppression of control from one of these levels is warranted. However, this may also lead to an instability in the system. It may be also possible to create a global controller that is responsible for synchronizing various local control mechanisms. However, clear semantics must be defined for the composition of the global controller which could lead to an overall increase in design complexity.

The global impact of any control model is undesirable as the modeling approach becomes more complex and several high abstraction levels need to be managed. A local approach is more desirable as it does not affect any other modeling level. However, in each of the above mentioned control models, strict conditions must be fulfilled for their construction. These conditions may not be met depending upon the designer environment. Thus an ideal control model is one that has only a local impact range and does not have any strict construction conditions.

7. Control at Deployment level

In this section we explain control integration at another abstraction level in SoC co-design. This level deals with linking the modeled application and architecture components to their respective IPs. We explain the component model of this *deployment* level in the particular case of the Gaspard2 framework within the context of dynamic reconfiguration.

For dynamic reconfiguration in modern SoCs, an embedded controller is essential for managing a dynamically reconfigurable region. This component is usually associated with some control semantics such as state machines, Petri nets etc. The controller normally has two functionalities: one responsible for communicating with the FPGA *Internal Configuration Access Port* hardware reconfigurable core or ICAP [B. 03] that handles the actual FPGA switching; and a state machine part for switching between the available configurations. The first functionality is written manually due to some low level technological details which cannot be expressed via a high level modeling approach.

The control at the deployment level is utilized to generate the second functionality automatically via model transformations. Finally the two parts can be used to implement partial dynamic reconfiguration in an FPGA that can be divided into several static/reconfigurable regions. A reconfigurable region can have several implementations, with each having the same interface, and can be viewed as a mode switch component with different modes. In our design flow, this dynamic region

is generated from the high abstraction levels, i.e., a complex Gaspard2 application specified using the MARTE profile. Using the control aspects in the subsequently explained Gaspard2 deployment level, it is possible to create different configurations of the modeled application. Afterwards, using model transformations, the application can be transformed into a hardware functionality, i.e., a dynamically reconfigurable hardware accelerator, with the modeled application configurations serving as different implementations related to the hardware accelerator.

We now present integration of the control model at the deployment level. We first explain the deployment level in Gaspard and our extensions followed by the control model.

Deployment in Gaspard2

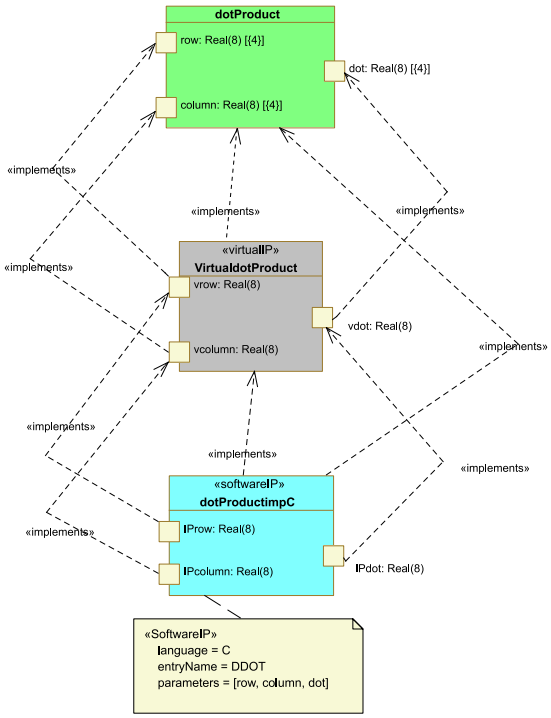

Figure 1.5. Deployment of an elementary `dotProduct` component in Gaspard2

The Gaspard2 deployment level enables one to precise a specific IP for each elementary component of application or architecture, among several possibilities [APN+07]. The reason is that in SoC design, a functionality can be implemented in different ways. For example, an application functionality can either be optimized for a processor, thus

written in C/C++, or implemented as a hardware accelerator using *Hardware Description Languages* (HDLs). Hence the deployment level differentiates between the hardware and software functionalities; and allows moving from platform-independent high level models to platform-dependent models for eventual implementation. We now present a brief overview of the deployment concepts.

A `VirtualIP` expresses the functionality of an elementary component, independently from the compilation target. For an elementary component K, it associates K with all its possible IPs. The desired IP(s) is (are) then selected by the SoC designer by linking it (them) to K via an `implements` dependency. Finally, the `CodeFile` (not illustrated in the chapter) determines the physical path related to the source/binary code of an IP, along with required compilation options.

Multi-Configuration approach

Currently in deployment level, an elementary component can be associated with only one IP among the different available choices (if any). Thus the result of the application/architecture (or the mapping of the two forming the overall system) is a static one. This collective composition is termed as a *Configuration*.

Integrating control in deployment allows to create several configurations related to the modeled application for the final realization in an FPGA. Each configuration is viewed as a collection of different IPs, with each IP associated with its respective elementary component. The end result being that one application model is transformed by means of model transformations and intermediate metamodels into a dynamically reconfigurable hardware accelerator, having different implementations equivalent to the modeled application configurations.

A `Configuration` has the following attributes. The name attribute helps to clarify the configuration name given by a SoC designer. The ConfigurationID attribute permits to assign unique values to each of the modeled `Configuration`, which in turn are used by the control aspects presented earlier. Theses values are used by a Gaspard state graph to produce the mode values associated with its corresponding Gaspard state graph component. These mode values are then sent to a mode switch component which matches the values with the names of its related collaborations as explained in [QMD09]. If there is a match, the mode switch component switches to the required configuration. The InitialConfiguration attribute sets a Boolean value to a configuration to indicate if it is the initial configuration to be loaded onto the target

FPGA. This attribute also helps to determine the initial state of the Gaspard state graph.

An elementary component can also be associated with the same IP in different configurations. This point is very relevant to the semantics of partial bitstreams, e.g., FPGA configuration files for partial dynamic reconfiguration, supporting *glitchless dynamic reconfiguration*: if a configuration bit holds the same value before and after reconfiguration, the resource controlled by that bit does not experience any discontinuity in operation. If the same IP for an elementary component is present in several configurations, that IP is not changed during reconfiguration. It is thus possible to link several IPs with a corresponding elementary component; and each link relates to a unique configuration. We apply a condition that for any n number of configurations with each having m elementary components, each elementary component of a configuration must have *at least* one IP. This allows successful creation of a complete configuration for eventual final FPGA synthesis.

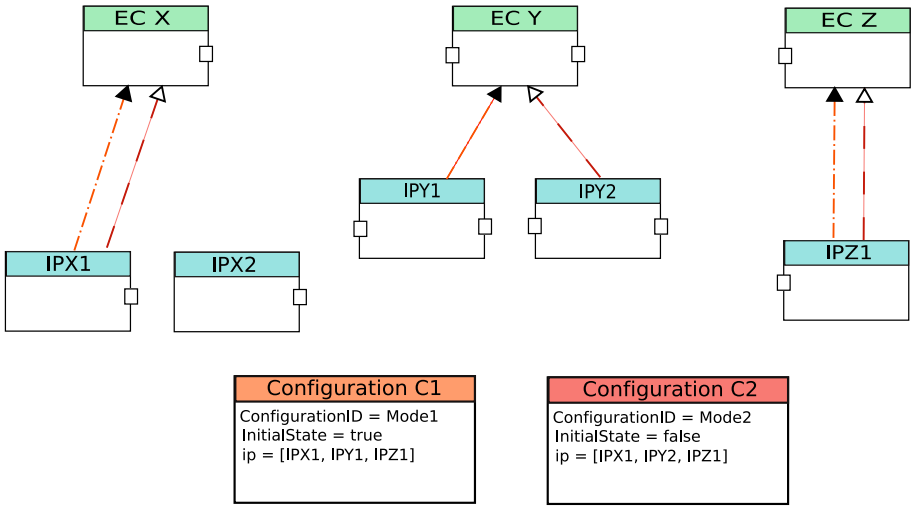

Figure 1.6. Abstract overview of configurations in deployment

Figure 1.6 represents an abstract overview of the configuration mechanism introduced at the deployment level. We consider a hypothetical Gaspard2 application having three elementary components *EC X*, *EC Y* and *EC Z*, having available implementations *IPX1*, *IPX2*, *IPY1*, *IPY2* and *IPZ1* respectively. For the sake of clarity, this abstract representation omits several modeling concepts such as `VirtualIP` and `Implements`. However, this representation is very close to UML model-

ing as presented earlier in the chapter. A change in associated implementation of any of these elementary components may produces a different end result related to the overall functionality, and different QoS criteria such as effectively consumed FPGA resources.

Here two configurations *Configuration C1* and *Configuration C2* are illustrated in the figure. *Configuration C1* is selected as the initial configuration and has associated IPs: *IPX1*, *IPY1* and *IPZ1*. Similarly *Configuration C2* also has its associated IPs. This figure illustrates all the possibilities: an IP can be globally or partially shared between different configurations: such as *IPX1*; or may not be included at all in a configuration, e.g., case of *IPX2*.

Once the different implementations are created by means of model transformations, each implementation is treated as a source for a partial bitstream. A bitstream contains packets of FPGA configuration control information as well as the configuration data. Each partial bitstream signifies a unique implementation, related to the reconfigurable hardware accelerator which is connected to an embedded controller. While this extension allows to create different configurations, the state machine part of the controller is created manually. For automatic generation of this functionality, the deployment extensions are not sufficient. We then make use of the earlier control semantics at the deployment level.

Implementation

Once control has been integrated at deployment level, it helps to switch between the different modeled configurations [QMMD09]. The configurations relate to a Gaspard2 application modeled at the high abstraction levels. This application is transformed into a hardware functionality, i.e., a hardware accelerator, by means of the model transformations, as stated earlier.

The application targeted for the validation of our methodology is a delay estimation correlation module integrated in an anti-collision radar detection system. Our radar uses a PRBS (Pseudorandom binary sequence) of length of 127 chips. In order to produce a computation result, the algorithm requires 127 multiplications between the 127 elements of the reference code that is generated via MATLAB and the last 127 received samples. The result of this multiplication produces 64 data elements. The sum of these 64 data elements produces the final result. This result can be sent as input to other parts of our radar detection system [QEMD09] in order to detect the nearest object. The different configurations related to our application change the IPs related to the elementary components, which in turn allow us to manipulate

different QoS criteria such as consumed FPGA resources and overall energy consumption levels. The partially reconfigurable system has been implemented on a Xilinx XC2VP30 Virtex-II Pro FPGA with a hardcore PowerPC 405 processor as a reconfiguration controller with a frequency of 100 MHz. We implemented two configurations on the targeted architecture, two with different IPs related to an multiplication elementary component in the application and a blank configuration. The results are shown in Figure 1.7.

	FPGA resources			Dynamic Power (mW)	Mean Reconfiguration Time (secs)
	Slices	Slice Flip Flops	LUTs		
Configuration 1 : Multiplication IP written with DSP functionality	1272/13696 (9.287%)	2084/27392 (7.608%)	1584/27392 (5.782%)	221.42	1.45
Configuration 2 : Multiplication IP written with if-else functionality	1186/13696 (8.659%)	1944/27392 (7.096%)	1836/27392 (6.702%)	214.56	1.41

Figure 1.7. An overview of the obtained results

Advantages of control deployment level

The advantage of using control at deployment is that the impact level remains local and there is no influence on other modeling levels. Another advantage is that the application, architecture and allocation models can be reused again and only the necessary IPs are modified. As we validate our methodology by implementing partial dynamic reconfigurable FPGAs, we need to clarify about the option of choosing mode-automata.

Many different approaches exist for expressing control semantics, mode automata were selected as they clearly separate control/data flow. They also adapt a state based approach facilitating seamless integration in our framework; and can be expressed at the MARTE specification levels. The same control semantics are then used throughout our framework to provide a single homogeneous approach. With regards to partial dynamic reconfiguration, different implementations of a reconfigurable region must have the same external interface for integration with the static region at run-time. Mode automata control semantics can express the different implementations collectively via the concept of a mode switch, which can be expressed graphically at high abstraction levels using the concept of a mode switch component. Similarly a state graph component expresses the controller responsible for the context switch between the different configurations.

8. Conclusion

This chapter presents a high abstraction level component based approach integrated in Gaspard2, a SoC co-design framework compliant with the MARTE standard. The control model is based on mode automata, and takes task and data parallelism into account. The control semantics can be integrated into various levels in Gaspard2. We compare the different approaches with respect to different criteria such as impact on other modeling levels. Control integration in application level allows dynamic context switching. In addition, safety of the control can be checked by tools associated with synchronous languages when the high-level model is transformed into synchronous code. Control at the architectural level can be concerned with QoS criteria as well as structural aspects. Similarly, control at the allocation level offers advantages of *Design Space Exploration*. Finally we present control semantics in the deployment level which offer reuse of application, architecture and allocation models. This control model makes it possible to support partial dynamic reconfiguration in reconfigurable FPGAs. A case study has also been briefly presented to validate our design methodology. Currently we have only focused on isolating controls at different levels in Gaspard2. An ideal perspective could be a combination of the different control models to form a merged approach.

References

[A. 08a] A. Cuoccio and P. R. Grassi and V. Rana and M. D. Santambrogio and D. Sciuto. A Generation Flow for Self-Reconfiguration Controllers Customization. *Forth IEEE International Symposium on Electronic Design, Test and Applications, DELTA 2008*, pages 279–284, 2008.

[A. 08b] A. Gamatié and S. Le Beux and E. Piel and A. Etien and R. B. Atitallah and P. Marquet and J.-L Dekeyser. A model driven design framework for high performance embedded systems. Research Report RR-6614, INRIA, 2008. `http://hal.inria.fr/inria-00311115/en`.

[A. 08c] A. Koudri et al. Using MARTE in the MOPCOM SoC/SoPC Co-Methodology. In *MARTE Workshop at DATE'08*, 2008.

[AMAB+06] L. Apvrille, W. Muhammad, R. Ameur-Boulifa, S. Coudert, and R. Pacalet. A UML-based environment for system design space exploration. *Electronics, Circuits and Systems, 2006. ICECS '06. 13th IEEE International Conference on*, pages 1272–1275, Dec. 2006.

[APN+07] R. B. Atitallah, E. Piel, S. Niar, P. Marquet, and J.-L Dekeyser. Multilevel MPSoC simulation using an MDE approach. In *SoCC 2007*, 2007.

[B. 03] B. Blodget and S. McMillan and P. Lysaght. A lightweight approach for embedded reconfiguration of FPGAs. In *Design, Automation & Test in Europe, DATE'03*, 2003.

[B. 04] B. Nascimento et al. A partial reconfigurable architecture for controllers based on Petri nets. In *SBCCI '04: Proceedings of the 17th symposium on Integrated circuits and system design*, pages 16–21. ACM, 2004.

[BAP05] Jérémy Buisson, Franoise André, and Jean-Louis Pazat. A Framework for Dynamic Adaptation of Parallel Components. In *ParCo 2005*, 2005.

[Bay08] Bayar, S., and Yurdakul, A. Dynamic Partial Self-Reconfiguration on Spartan-III FPGAs via a Parallel Configuration Access Port (PCAP). In *2nd HiPEAC workshop on Reconfigurable Computing, HiPEAC 08*, 2008.

[Bou07] P. Boulet. Array-OL revisited, multidimensional intensive signal processing specification. Research Report RR-6113, INRIA, `http://hal.inria.fr/inria-00128840/en/`, February 2007.

[C. 07] C. Claus and F.H. Muller and J. Zeppenfeld and W. Stechele. A new framework to accelerate Virtex-II Pro dynamic partial self-reconfiguration. *IPDPS 2007*, pages 1–7, 2007.

[C. 08] C. Schuck and M. Kuhnle and M. Hubner and J. Becker. A framework for dynamic 2D placement on FPGAs. In *IPDPS 2008*, 2008.

[DaR09] DaRT team. GASPARD SoC Framework, 2009. `http://www.gaspard2.org/`.

[D.D83] D.D. Gajski and R. Khun. New VLSI Tools. *IEEE Computer*, 16:11–14, 1983.

[DHJP08] L. Doyen, T. Henzinger, B. Jobstmann, and T. Petrov. Interface theories with component reuse. In *EMSOFT'08: Proceedings of the 8th ACM international conference on Embedded software*, pages 79–88, New York, NY, USA, 2008. ACM.

[E. 03] E. Brinksma and G. Coulson and I. Crnkovic and A. Evans and S. Grard and S. Graf and H. Hermanns and J. Jzquel and B. Jonsson and A. Ravn and P. Schnoebelen and F. Terrier and A. Votintseva. Component-based Design and Integration Platforms: a Roadmap. *The Artist consortium*, 2003.

[ea08] S. Bell et al. TILE64 - Processor: A 64-Core SoC with Mesh Interconnect. In *IEEE International Digest of Technical Pa-*

pers on Solid-State Circuits Conference (ISSCC 2008), pages 88–598, 2008.

[F. 08] F. Berthelot and F. Nouvel and D. Houzet. A Flexible system level design methodology targeting run-time reconfigurable FPGAs. *EURASIP Journal of Embedded Systems*, 8(3):1–18, 2008.

[Gra08] S. Graf. Omega – Correct Development of Real Time Embedded Systems. *SoSyM, int. Journal on Software & Systems Modelling*, 7(2):127–130, 2008.

[GRY08a] A. Gamatié, É. Rutten, and H. Yu. A Model for the Mixed-Design of Data-Intensive and Control-Oriented Embedded Systems. Research Report RR-6589, INRIA, `http://hal.inria.fr/inria-00293909/fr`, July 2008.

[GRY+08b] A. Gamatié, É. Rutten, H. Yu, P. Boulet, and J.-L. Dekeyser. Synchronous modeling and analysis of data intensive applications. *EURASIP Journal on Embedded Systems*, 2008. To appear. Also available as INRIA Research Report: `http://hal.inria.fr/inria-00001216/en/`.

[H. 08] H. Yu. *A MARTE based reactive model for data-parallel intensive processing: Transformation toward the synchronous model*. PhD thesis, USTL, 2008.

[Har87] David Harel. Statecharts: A Visual Formalism for Complex Systems. *Science of Computer Programming*, 8(3):231–274, June 1987. article note found.

[I.-08] I.-R. Quadri and P. Boulet and S. Meftali and J.-L. Dekeyser. Using An MDE Approach for Modeling of Interconnection networks. In *The International Symposium on Parallel Architectures, Algorithms and Networks Conference (ISPAN 08)*, 2008.

[IKH01] C. Im, H. Kim, and S. Ha. Dynamic voltage scheduling technique for low-power multimedia applications using buffers, 2001.

[J. 03] J. Becker and M. Huebner and M. Ullmann. Real-Time Dynamically Run-Time Reconfigurations for Power/Cost-optimized Virtex FPGA Realizations. In *VLSI'03*, 2003.

[K. 07] K. Paulsson and M. Hubner and G. Auer and M. Dreschmann and L. Chen and J. Becker. Implementation of a Virtual Internal Configuration Access Port (JCAP) for Enabling Partial Self-Reconfiguration on Xilinx Spartan III FPGA. *International Conference on Field Programmable Logic and Applications, FPL 2007*, pages 351–356, 2007.

[L. 98] L. Bass and P. Clements and R. Kazman. Software Architecture In Practice. *Addison Wesley*, 1998.

[Lat99] Latella, D. and Majzik, I. and Massink, M. Automatic Verification of a Behavioral Subset of UML Statechart Diagrams Using the SPIN Model-Checker. In *Formal Aspects Computing*, volume 11, pages 637–664, 199.

[M. 06] M. Huebner and C. Schuck and M. Kiihnle and J. Becker. New 2-Dimensional Partial Dynamic Reconfiguration Techniques for Real-Time Adaptive Microelectronic Circuits. In *ISVLSI'06*, 2006.

[MR98] F. Maraninchi and Y. Rémond. Mode-automata: About modes and states for reactive systems. In *European Symposium On Programming*, Lisbon (Portugal), March 1998. Springer verlag.

[O. 05] O. Labbani and J.-L Dekeyser and Pierre Boulet and É. Rutten. Introducing control in the gaspard2 data-parallel metamodel: Synchronous approach. In *Proceedings of the International Workshop MARTES: Modeling and Analysis of Real-Time and Embedded Systems*, 2005.

[OMG] OMG. Modeling and analysis of real-time and embedded systems (MARTE). http://www.omgmarte.org/.

[OMG07] OMG. Portal of the Model Driven Engineering Community, 2007. `http://www.planetmde.org`.

[P. 05] P. Sedcole and B. Blodget and J. Anderson and P. Lysaght and T. Becker. Modular Partial Reconfiguration in Virtex FPGAs. In *International Conference on Field Programmable Logic and Applications, FPL'05*, pages 211–216, 2005.

[P. 06] P. Lysaght and B. Blodget and J. Mason. Invited Paper: Enhanced Architectures, Design Methodologies and CAD Tools for Dynamic Reconfiguration of Xilinx FPGAs. In *FPL'06*, 2006.

[QEMD09] I. R. Quadri, Yassin Elhillali, S. Meftali, and J.-L. Dekeyser. Model based design flow for implementing an Anti-Collision Radar system. In *9th International IEEE Conference on ITS Telecommunications (ITS-T 2009)*, 2009.

[QMD09] I. R. Quadri, S. Meftali, and J.-L. Dekeyser. Integrating Mode Automata Control Models in SoC Co-Design for Dynamically Reconfigurable FPGAs. In *International Conference on Design and Architectures for Signal and Image Processing (DASIP 09)*, 2009.

[QMMD09] I. R. Quadri, A. Muller, S. Meftali, and J.-L. Dekeyser. MARTE based design flow for Partially Reconfigurable

Systems-on-Chips. In *17th IFIP/IEEE International Conference on Very Large Scale Integration (VLSI-SoC 09)*, 2009.

[R. 06] R. Koch and T. Pionteck and C. Albrecht and E. Maehle. An adaptive system-on-chip for network applications. In *IPDPS 2006*, 2006.

[SA00] M.T. Segarra and F. André. A framework for dynamic adaptation in wireless environments. In *Proceedings of 33rd International Conference on Technology of Object-Oriented Languages (TOOLS 33)*, pages 336–347, 2000.

[SKM01] T. Schäfer, A. Knapp, and S. Merz. Model checking UML state machines and collaborations. In *CAV Workshop on Software Model Checking*, ENTCS 55(3), 2001.

[Szy98] C. Szyperski. ACM Press and Addison-Wesley, New York, N.Y, 1998.

[T. 06] T. Mens and P. Van Gorp. A taxonomy of model transformation. In *Proceedings of the International Workshop on Graph and Model Transformation, GraMoT 2005*, pages 125–142, 2006.

[Xil06] Xilinx. Early Access Partial Reconfigurable Flow. 2006. `http://www.xilinx.com/support/prealounge/protected/index.htm`.

[YHBM02] L. Yung-Hsiang, L. Benini, and G. De Micheli. Dynamic frequency scaling with buffer insertion for mixed workloads. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 21(11):1284–1305, 2002.