MARTE based model driven design methodology for targeting dynamically reconfigurable FPGA based SoCs

Imran Rafiq Quadri

MARTE based model driven design methodology for targeting dynamically reconfigurable FPGA based SoCs

By

IMRAN RAFIQ QUADRI

A dissertation submitted in partial fulfillment of the requirements for the degree of

DOCTOR OF PHILOSOPHY

in

Computer Science

in the

École Doctorale Sciences pour l'Ingénieur

UNIVERSITÉ DES SCIENCES ET TECHNOLOGIES DE LILLE - FRANCE

Committee in charge:

Gilles Grimaud	Professor USTL	**President**
Bertrand Granado	Professor ETIS, ENSEA	**Reviewer**
Guy Gogniat	Professor Lab-STICC, Université de Bretagne Sud	**Reviewer**
Hans Vandierendonck	Post-doctoral Research Fellow FWO, Ghent University	**Examiner**
Jean-Luc Dekeyser	Professor USTL	**Director**
Samy Meftali	Assistant Professor USTL	**Co-director**

Université des Sciences et Technologies de Lille
LIFL - UMR 8022 - Cité Scientifique, Bât. M3 - 59655 Villeneuve d'Ascq Cedex

Contents

List of Figures

List of Tables

Introduction

Context and problematic

Systems-on-Chips (SoCs) are gradually becoming an essential aspect of our professional and personal lives. From avionics, transport, defense, medical and telecommunication systems to general commercial appliances such as smart phones, high definition TVs, gaming consoles; SoCs are now omnipresent, and it is difficult to find a domain where these miniaturized systems have not made their mark. In accordance with Moore's law that will probably hold true for the next several years, the continuous hardware evolution has permitted to double the number of integrated transistors in a single chip. Already, the 45nm technology mark has been achieved, with future evolution aiming to move into the field of quantum or nanoelectronics, in order to achieve the elusive single digit nm technology node. These future SoCs will be heavily integrated in human lives, such as their implantation in human body to regulate body functions and to monitor behavior; helping in executing tasks such as cooking, shopping and even driving advanced autopilot cars, akin to flying modern aircrafts with fly-by-wire and remote-control driven systems.

Modern SoCs are also considered as an integral solution for designing embedded systems. According to a modest estimate, until now, during the writing of this thesis; the global embedded systems market has a current value of 88.144 million U.S dollars with an average annual growth rate of 14%, due to the sale of over 10 billion embedded processors in 2008. Similarly, the embedded software market has a current value of 3.488 million U.S dollars with an average growth of 16%. SoCs offer advantages in the embedded domain such as reduction of physical surface area, consumed energy and overall fabrication costs. These SoC based embedded systems generally target *data intensive processing* (DIP) applications where large amount of data are processed in a regular manner by means of repetitive computations.

In addition to requiring more computing power, these applications are often subjected to timing constraints that must be respected. Additionally, in order to keep up to pace with the rapid hardware evolution, SoC software developers need to increase the computational capacity of the targeted applications, for handling large numbers of incoming signals/data, on which computations are to be applied rapidly. Optimization of these functionalities often results in parallelization of applications and resources that make up the embedded system. The parallelism increases the number of computations executed at a time while limiting the energy consumption levels. In SoCs, it is also possible to execute these applications as hardware functionalities: i.e., *hardware accelerators*, in order to perform a parallel execution in comparison to a sequential one. A hardware accelerator is an electronic circuit specifically designed to handle systematic signal processing, and permits maximum parallelization of the computations necessary for the execution of an application.

Reconfiguration can be seen as an integral feature of modern SoCs based embedded systems, in an increasingly evolving market space. A reconfigurable SoC offers increased functional extensibility in return for lower performance. These systems can be reconfigured an arbitrary number of times and offer designers the means to add new functionalities and make system modifications after the fabrication of a SoC. *Dynamic reconfiguration*, which is a special type of reconfiguration, enables system modification at run-time, introducing the concept of *virtual hardware*. Thus designers can change the executing applications on these systems, depending upon Quality-of-Service (QoS) criteria related to the environment or the platform: such as used surface area, energy consumption levels, etc. Currently, *Field Programmable Gate Array* (FPGA)

based SoCs offer an ideal solution for implementing dynamic reconfiguration. Moreover, SoC application functionalities can be easily implemented as hardware designs on these reconfigurable SoCs. As compared to traditional SoCs, these dynamically reconfigurable SoCs offer advantages such as low energy consumption, increased flexibility; with the compromise of additional costs per unit. Normally, these systems also integrate some sort of a reconfiguration controller that manages the reconfiguration process between the static and dynamically reconfigurable regions of the system. This module is one of the key integral concepts in the system, and is usually associated with some control semantics such as petri nets, state machines or automata. While much work has been realized related to run-time reconfigurable systems, many of the research efforts have been primarily motivated by proposals of new reconfigurable *architectures*; and optimizations of the related low level technical details, that require careful manipulations. Consequently, not enough attention has been given to the application-driven research. This is one of the possible primary reasons why partial reconfiguration has not taken off in the SoC industry.

In the wake of the continuous hardware/software evolution related to SoCs and the addition of features such as dynamic reconfiguration, the complexity of design and development of SoC has escalated to new heights in an exponential manner. If more hardware components are integrated, or an application is deemed to provide more features, development costs and time to market shoot up proportionally. Without the usage of effective design tools and methodologies, large complex SoCs are becoming increasingly difficult to manage, resulting in a productivity gap. The design space, representing all technical decisions that need to be elaborated by the SoC design team is therefore, becoming difficult to explore. Similarly, manipulation of these systems at low implementation levels such as Register Transfer Level (RTL) can be hindered by human interventions and the subsequent errors.

It is therefore essential to offset the gradual building complexity of SoC by intensifying the productivity of SoC designers. Indeed, it is one of the primary objectives in order to avoid the rising development costs. Moreover, even if the cost was not a crucial constraint, the size of a design team cannot be increased endlessly. There comes a time when the division of labor no longer helps in reducing design time. This design time is directly linked to the time-to-market, in order for a SoC vendor to be the first to produce a specific product in the electronics industry. This in turn, insures a successful profit and helps in attaining a dominant foothold in the industry.

Currently, we are therefore faced with a need to design more effective SoCs. Various methodologies and propositions have been proposed for this purpose. A *Platform* or *component based approach* is widely accepted in the SoC industry, permitting system conception and eventual design in a compositional manner. The hierarchy related to the SoC is visible quite clearly, and designers are capable to re-use components that have been either developed internally or by third parties. Other methodologies make use of high abstraction levels, in order to elevate the low level technical details. The management of parallelism and repetitiveness in the system is a key point that ought to be treated specifically, and must be conserved irrespective of the design methodology. In addition, these systems should also be eventually developed, and efforts must be made to maximize debugging and testing for minimizing the manufacturing costs, power consumption levels and system size.

Contributions

It is in the context of improving the primary productivity of parallel embedded reconfigurable SoCs, that this dissertation finds its proper place. One of the primary guidelines followed during this work is the utilization of *Model-Driven Engineering* (MDE) for SoC Co-Design specification and development. MDE is able to benefit from a component based model driven approach, allowing to abstract and simplify the system specifications, while integrating a compilation chain to transform the high level models into suitable code for SoC creation. The SoC Co-Design specifications have been provided by the Object Management Group (OMG); in the form of the *Modeling and Analysis of Real-Time and Embedded Systems* (MARTE) Unified Modeling Language (UML) profile that is gradually becoming the de-facto industry standard, enabling increased synergy between SoC vendors, designers and eventual users.

One of the objectives of this dissertation was to evolve the MDE compliant Gaspard2 SoC Co-Design framework that has been developed by the DaRT project-team at INRIA Lille-Nord Europe. Gaspard2 uses the MARTE profile for unified high level specifications of SoC applications and architectures; along with their allocations and the eventual deployment to *Intellectual Properties* (IPs). This representation is carried out in a graphical manner, and enables to target different execution platforms for automatic generation of the respective code.

However, while Gaspard2 and in turn MARTE, permits abstract modeling of SoCs, in the particular case of dynamically reconfigurable SoCs, the specifications lack suitable control semantics for expressing reconfigurability at the high abstraction levels. Similarly, it is not evident to determine which parts of a reconfigurable system can be specified via the MARTE specifications. In turn, the modeled SoCs are static in nature, and minute changes require modification of the design models. Our dissertation addresses these limitations and introduces several contributions which help to integrate dynamic reconfiguration in Gaspard2. The main contributions of this dissertation are defined below:

- ***Selecting parts of dynamically reconfigurable SoCs for high level MARTE modeling*:** One of the initial challenges faced by a SoC designer is selecting parts of the reconfigurable system for eventual specification at the high abstraction levels. As described in the previous section, with respect to partial dynamic reconfiguration, research is more oriented towards architectural aspects as compared to being application-driven, resulting in gradual decrease in the evolution of this particular domain. Additionally, it is currently not a simple task to bridge the gap between high level MARTE models and the SoC vendor proprietary tools that help in the construction of these systems. This is one of the reasons that we have selected to base this dissertation on an applicative oriented approach for the conception of reconfigurable SoCs. We intend to provide high level models of two key aspects related to a reconfigurable SoC, namely the control semantics related to the reconfiguration controller; and the dynamically reconfigurable region which is created from the MARTE compliant application model.

- ***Introduction of control semantics*:** Related to the first contribution, we initially provide generic control semantics in the Gaspard2 framework. The introduced semantics permit to integrate reconfigurability features at high modeling levels in Gaspard2, and respectively in MARTE. Mode-automata control semantics were chosen due to their compositional nature, facilitating integration in a component oriented MDE methodology. While the control semantics can be added to any modeling level in a SoC framework, with respect to run-time reconfiguration, the deployment level has been targeted for the successful integration of the control semantics. These control semantics are related to producing part of the code of a reconfigurable controller, for managing the dynamically reconfigurable region.

- ***Integrating configurations in deployment modeling level*:** Currently, one of the main features of Gaspard2 is the ability to link the elementary components of the modeled application/architecture to the available user defined or third party intellectual properties. Along with the integration of control semantics, the current deployment level has also been extended to integrate the notion of *configurations*, which are unique global implementations of a high level modeled application functionality, with each configuration comprised of different combinations of IPs related to the elementary components. Using a combination of the deployment level and the introduced control semantics, it is possible for a designer to change the global configuration related to an application, permitting different results. Ultimately, the MARTE *metamodel* and *profile*, respecting the MDE principles, have been extended to support the control and deployment semantics; enabling expression of the introduced semantics at the UML modeling level.

- ***Hardware execution model for reconfigurable MARTE applications*:** Another significant contribution related to this dissertation is the proposal of a specific hardware execution model. The model describes the behavior of a high level modeled MARTE application in the context of the Gaspard2 framework, when it is translated into a hardware functionality

for execution as a hardware accelerator. This hardware accelerator is treated as a dynamically reconfigurable region in a targeted reconfigurable SoC, with the associated global configurations serving as its different implementations. Among several available execution models, we have selected an execution model that corresponds to the requirements of dynamic reconfiguration.

- ***Concepts for transforming high level models into Register Transfer Level equivalents*:** This contribution introduces the Register Transfer Level (RTL) *metamodel* which comprises of two different aspects. Firstly, it incorporates the concepts for translating the application model into a hardware functionality using the execution model described previously. Secondly, the metamodel enriches the control semantics with details related to RTL. The two aspects although share certain common *metaclasses*, are quite different in nature. The RTL *model*, which is an instance of the RTL metamodel, mainly corresponds to the description of the dynamically reconfigurable hardware accelerator and is the end model from which eventual code can be generated.

- ***A complete transformation chain*:** Model transformations in our design flow enable creation of a complete model transformation chain for automatic code generation of high level MARTE compliant UML models. The model-to-model transformations in our flow permit to move from high level UML models to the RTL model, all the while enriching the intermediate models. Thereafter, a model-to-text transformation generates HDL code equivalent to the different implementations of the hardware accelerator. At the same time, C/C++ language code is generated for the switch mechanism related to the reconfiguration controller. Henceforth by using commercial synthesis tools, it is possible to create a dynamically reconfigurable SoC.

Finally, the various contributions were implemented in the existing Gaspard2 framework. Our design flow is built on the MDE principles which are used throughout the life cycle of our methodology. We also limit the targeted application domain and the control semantics due to certain limitations present in MARTE and in the Gaspard2 framework.

Plan

Our research tries to respond to some of the critical questions: how to model complex applications using the MARTE profile ? how to provide a control semantics at high modeling levels ? how to integrate these models into a SoC Co-Design framework ? how to use these models for implementation in a dynamically reconfigurable SoC ? how to generate code from these high level models ?. We offer some answers regarding these questions in this document, with the subsequent plan as follows:

This dissertation is structured into three main portions. The first part concerns the basic concepts related to SoC, dynamic reconfiguration, component based design, MDE and the Gaspard2 framework.

- ***Chapter 1 : Systems-on-Chip***: The first chapter gives a brief overview of concepts related to SoCs such as the SoC Co-Design approach; and afterwards sheds lights on reconfigurable computing. We focus specially on dynamic reconfiguration in SoCs and provide some in-depth details related to its functioning.

- ***Chapter 2 : Component Based Design***: This chapter highlights a popular design methodology offering possible partial solutions to reduce the problems plaguing the SoC Co-Design. Concepts related to component models and frameworks are described while maintaining the link with dynamic reconfiguration.

- ***Chapter 3 : Model-Driven Engineering and MARTE***: An overview related to the basic terminologies related to MDE concepts is provided, such as *models*, *metamodels* and *model transformations*. Finally the MARTE profile has been summarized, along with its limitations regarding reconfigurability features.

- ***Chapter 4 : Gaspard2: An MDE-based framework for SoC Co-Design***: The Gaspard2 SoC Co-Design Framework is introduced in this chapter. Information about different available modeling levels is given with special emphasis on the application and deployment levels that are featured in this thesis.

The second and third portions of this dissertation are related to the personal contributions. The second portion highlights the concepts that have been integrated in our design flow at the high abstraction levels, for the eventual implementation and code generation phases, which are detailed in the third portion.

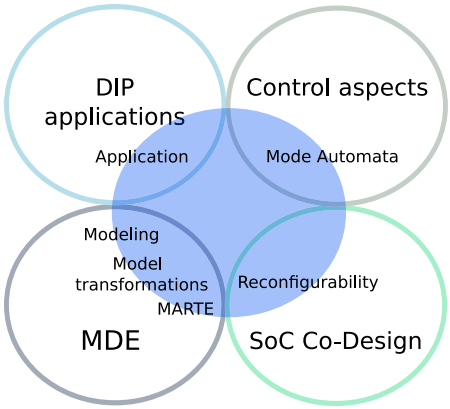


Figure 1: This thesis: a combination of several domains

- ***Chapter 5 : Methodology for global contribution***: This chapter summarizes the global contributions of this dissertation. We first present the motivations behind our proposed approach followed by the presentation of our design flow.
- ***Chapter 6 : Expressing adaptivity for SoC with MDE***: This chapter first provides the basic conditions and abstract concepts for the generic control models at different SoC Co-Design levels. Afterwards, the different control models are compared, resulting in integration of control semantics at the novel deployment level. Thereafter, concrete high level concepts are provided to integrate the control and configuration aspects in the MARTE metamodel for the eventual model transformations.
- ***Chapter 7 : A metamodel for targeting Register Transfer Level***: This chapter provides the details related to the RTL metamodel and the underlying concepts. We first describe the choice of selecting a hardware execution model and the associated conditions, that allow to describe the behavior of the hardware accelerator for executing the high level modeled application. Equally, the RTL metamodel contains concepts related to the control semantics expressed at the modeling level. These semantics are converted into the code for the reconfiguration controller by means of the model transformations present in our design flow.
- ***Chapter 8 : Model transformations and code generation***: The chapter specifies the different types of model transformations present in our design flow. The two main model-to-model transformations permit to transform the high level models into an RTL model; from which eventual code is generated for the hardware accelerator and the controller by means of a model-to-text transformation.
- ***Chapter 9 : Case study***: The validation of the transformation chain has been carried out by means of a case study. An application illustrating a delay estimation correlation module in an anti-collision radar detection system has been modeled via the MARTE profile, and is subsequently deployed along with integration of the control semantics. The generated code is then taken as input by commercial synthesis tools for final implementation on a target FPGA based reconfigurable SoC.

Finally, Figure 1 illustrates the various domains and methodologies contributing to this dissertation. The MDE framework provides the solutions of modeling and model transformations that bridge the gap between high-level specifications and low-level execution models. Control semantics, specially *mode automata* semantics offer modular-base control aspects. Reconfigurability, and especially dynamic reconfigurability, one of the emerging trends in SoC Co-design is also explored. Finally we also focus on parallel data intensive processing applications which are mainly used in these SoCs, and are specially targeted in Gaspard2.

Part I

Problematic

Chapter 1

Systems-on-Chip

1.1 Introduction

Since the early 2000s, Systems-on-Chip (or SoCs) have emerged as a new paradigm and one of the principle solutions for embedded systems design. Thanks to the rapid evolution in semicondutor technology, in a SoC, all the necessary components can be integrated in a single chip. Most Systems-on-Chip (SoC) designs are based on a *platform-based* solution, where standard *components* like microprocessors make up significant portion of the SoC. The components to be integrated can be any of, but not restricted to, the following components listed below:

- Processors (Hard/Soft core microprocessors, Digital Signal Processors (DSPs), etc.);
- Memory blocks (RAM, ROM, Flash, etc.);
- Inter-component communication connections (bus, crossbar, Network-on-Chip, etc.);
- External interfaces (USB, FireWire, Ethernet, PCI, etc.);
- Analog/digital converters;
- Sensors;
- Timing sources (clocks, oscillators, phase-locked loops, etc.);
- Reconfigurable elements, such as *Field Programmable Gate Array* (FPGAs).

As the whole system is integrated into a single chip, the system size is restricted by the chip size. It has been estimated that the number of integrated transistors on chip increases by 50 % per year. Moreover, as the integration level escalates in accordance with Moore's law [163], the chip size is getting smaller and smaller. For example, TSMC started to produce 0.183 mm^2 32 nanometer SRAM cells from 2005. Similarly in July 2009, Synopsys along with ARM, IBM, Samsung Electronics and Chartered Semiconductor Manufacturing announced an agreement to develop low power 22 nanometer node technologies which should be available by the end of 2011. In August of 2009, research using DNA origami on silicon showed promising results for reaching the six nanometer mark which is eight times better[1] than the current industry process. According to a safe estimate, this approach can be viable in the next 10 years. Similarly, in September 2009, Intel unveiled the *"Sodaville"* 45nm Atom CE4100 SoC[2] to bring multimedia applications and specially Internet content to digital electronic devices, with a clock speed of 12 GHz. The reduction in size of a SoC is highly beneficial to the development of mobile electronic devices, such as smart phones and tablets which themselves, in turn, are becoming more miniaturized. Low energy consumption is also one of the most significant features of SoCs: as these SoCs are usually placed in other systems, or implemented as mobile systems; which do not always provide large amounts of energy. In comparison with computing power growth, research into improving energy technology has demonstrated a tardy progress. In consequence, the low-energy consumption will be critical for future mobile devices, as the on-die signal delivery in SoCs helps to consume less energy. Figure 1.1 illustrates an example of a SoC with its interconnected components. This chapter details the different aspects of SoC, such as SoC design methodologies, the targeted application domains and reconfigurable computing which is a critical feature of modern SoCs. Afterwards we take a look at reconfigurable SoCs and different types of reconfiguration currently existing in reality. Finally, issues related to SoC Co-Design are mentioned at the end of the chapter.

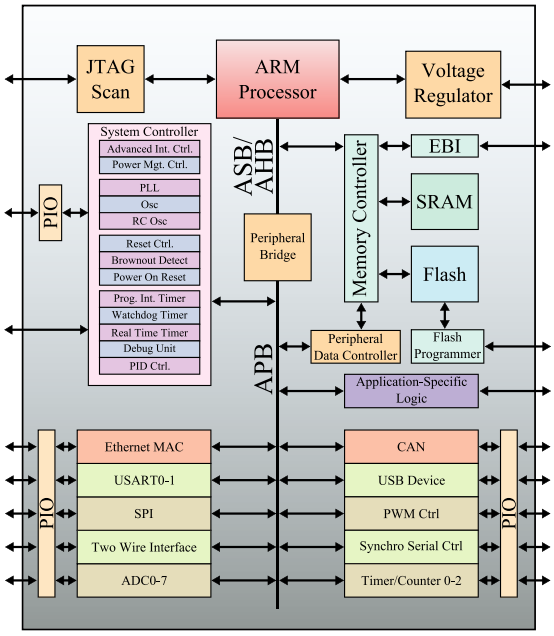


Figure 1.1: An example of SoC from Wikipedia: consisting of an ARM based processor along with communication interfaces

General computer systems are conceived as self-contained systems, which are capable of performing general purpose computing tasks. Whereas SoCs can integrate dedicated hardware, such as DSPs and hardware accelerators, to parallely execute specific applications or computation tasks (e.g., signal/image/video processing, Fast Fourier Transforms or FFTs[3], Discrete

[1] http://news.bbc.co.uk/2/hi/technology/8204906.stm
[2] http://news.cnet.com/8301-13924_3-10361101-64.html
[3] http://en.wikipedia.org/wiki/FFT

Cosine Transform or DCTs[4], encryption, etc.); which can take too long for serial execution in an embedded microprocessor. The software part of a SoC is written in accordance with the characteristics of the hardware components (such as execution speed, memory availability etc). Generally, there are few software layers and the OS (Operating System) is minimal or non-existential.

Nonetheless, the design and manufacturing of SoCs is a costly and time consuming process. For example, for fabrication on silicon, SoCs require creation of *masks*, which can cost about a couple of million of Euros. Moreover, the process of *photolithography* requires mandatory ultra-clean workspaces. Hence, manufacturing of SoCs is always involved in mass production as compared to costly prototypes; in order to reduce the overall costs including non-recurring engineering (NRE).

Towards Multiprocessor SoCs. Additionally, due to the strong requirements of the embedded systems community; and considering constraints such as time-to-market, fabrication costs etc., SoCs are expected to be specialized to respond to the previously mentioned needs, as well as to the needs for flexibility, high-performance etc. Also due to physical constraints related to frequency and voltage, it is not possible to just simply increase the size of the processor. It is therefore necessary to put *several* processors in a SoC.

Multiprocessor SoCs (MPSoCs) [128] are thus emerging, where multiple homogeneous or heterogeneous processing elements are integrated on the chip such as in Tile64 [25]; together with on-chip interconnections like Network-on-Chip (NoC) [28], hierarchical memory, I/O components, etc. MPSoCs are expected to satisfy the high-performance and low-energy consumption requirements demanded by targeted SoC applications. The performance of these architectures corresponds no longer to the speed of program instructions (as in the case of normal monoprocessors), but it is the absolute sum of the speed of the instructions of the different cores and the available processors.

1.2 SoC design

The SoC design develops into a system design, as the chip itself is a complete system. SoC system level design takes advantage of existing technologies to address SoC complexity issue [51], such as system level architecture and architectural verification, hardware/software Co-Design, and high level modeling.

1.2.1 SoC Co-Design

In [94], Gajski and Kuhn presented the *Y chart* for representing the different stages in SoC conception. A SoC design can be considered from three basic viewpoints, irrespective of its complexity. Each of the viewpoints represents a different aspect of the system. The Y chart has three axes to represent behavioral (functional) specification, structure (electronic description) and geometrical layout (physical arrangement) respectively. The Y chart also depicts different levels of abstraction in the design. Several levels are illustrated through circles from outside to the center in Figure 1.2. These levels denote different granularities and precisions in the design. SoC development tools help to move from high levels to more detailed low levels and aid in moving between the axes, with the final goal to have a precise physical design layout.

Inspired by the Y chart, the Y schema is generally adapted to represent the SoC Co-Design approaches. Its three axes represent functional behavior, hardware architecture and final implementation in specific technologies (e.g., circuit, programming languages, etc.). The central point of these three axes denotes the allocation of the application resources (data and instructions) onto the hardware resources (such as processors and memories). In parallel, elementary concepts in software and hardware can be deployed with user defined or third party *Intellectual Properties* (IPs) implemented by some specific technologies.

As seen in Figure 1.3, the architectural (hardware) and behavioral (software) parts of a complete system can be developed in parallel by different teams in order to benefit from their respective experiences in different domains. Moreover, concurrent and parallel development of

[4] http://en.wikipedia.org/wiki/Discrete_cosine_transform

hardware and software by different teams helps to shorten the overall design time. For instance, the software teams need not wait for the final configuration/netlist[5] of architecture conceived by hardware teams; to start software development. The application behavior is then mapped onto the hardware architecture, on which different analyses (such as simulation) can be carried. Analysis results can be used for the modification of the original design at the system modeling stage; and at the mapping stage for different purposes resulting in a *Design Space Exploration* (DSE) strategy. If the mapping result is approved by the analysis, it can be utilized for implementation purposes.

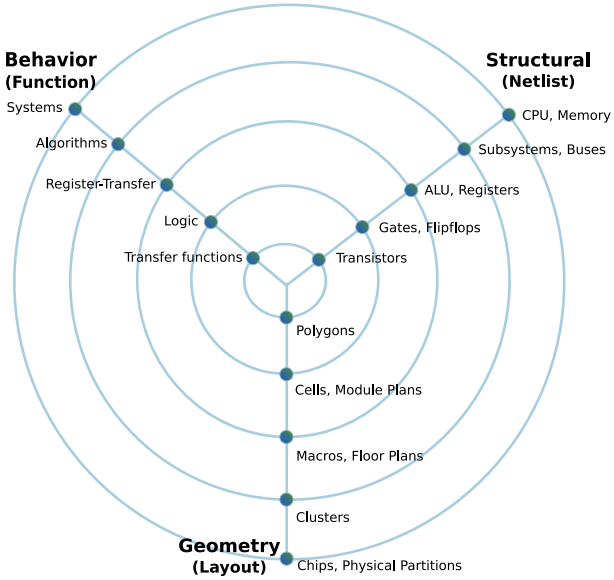


Figure 1.2: Y chart for the system-level design

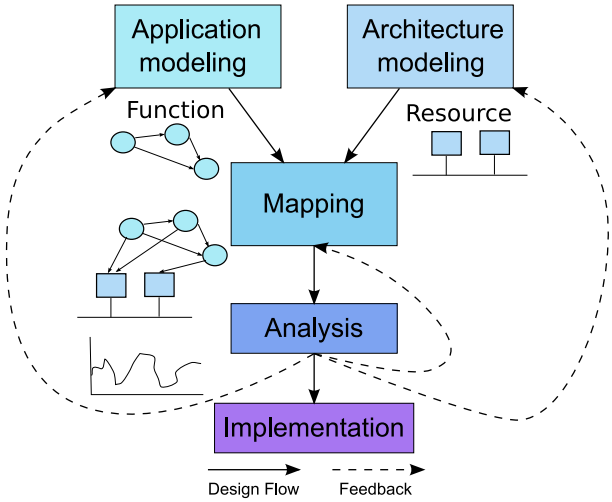


Figure 1.3: SoC Co-Design flow [51]

In the hardware design, IP building blocks have been widely used in the SoC design for re-usability, cost effectiveness and time-to-market reasons. SoC hardware IPs can involve processors, memories, I/O, etc. SoC software can also have IPs, such as some elementary functions in embedded multimedia processing, which include FFTs, filters, codecs, etc. IPs help SoC designers to avoid re-inventing the wheel for existing designs and help to reduce development

[5] In electronic design, a netlist describes the connectivity of the design

time. IP technology has turned out to be one of the most encouraging aspects in the SoC approach, facilitating rapid design and development.

1.2.2 SoC application domains

Due to their numerous advantages, such as low energy consumption, powerful computing capacity and small size, SoC based embedded systems are omnipresent nowadays. For example, as compared to 260 million processors that were sold in 2004, 14 billion embedded processors (such as microprocessors, DSPs, micro-controllers, etc.) have been sold in 2008. The previously mentioned characteristics make SoCs well adapted for diverse application domains such as defense industry, satellite based systems, telecommunications, aeronautics, automobile, transport, domestic appliances, medical equipments, mobile electronic products, etc. The targeted embedded applications cover both critical as well as non-critical systems. Current SoC based embedded systems can be used in more complex systems, such as electronic commerce, video processing etc; as well as commercial electronic products for the general public (smart phones, PDAs, set-top boxes, gaming consoles etc).

High-Performance Computing (HPC). HPC applications indicate a significant application domain of SoCs, such as embedded multimedia devices, radar/sonar signal processing devices, defense based missile trajectory/tracking systems etc., which have become quickly widespread over recent years. The applications on these devices are always involved in signal (data)-intensive parallel computing, which are generally regular for reasons of high performance. Moreover, some specific building blocks on SoCs are dedicated for handling large amounts of data parallel processing with performance. These blocks can include DSPs, hardware accelerators, GPUs (Graphical Processing Units) etc.

1.3 Reconfigurable computing

Reconfigurable computing is also an emerging paradigm for present and future computing requirements of embedded applications, in terms of flexibility and performance. Conventional execution of complex algorithms employs two methods. The first is using traditional Von Neumann computing by programming microprocessors, micro-controllers with sequential based data processing. The second method is the usage of application specific processors or integrated circuits such as ASICs (Application Specific Integrated Circuits) with real data parallel processing. The first solution offers increase flexibility with degraded performance while in the second solution, the system is designed for customized applications with tight performance constraints (such as latency, throughput, power consumption, area etc.); making future modifications non-cost effective and improbable. The gap between these two approaches can be bridged by using *Reconfigurable Computing Systems* (RCS) that were first introduced in the 1960s[6]. The general definition of RCS can be stated as:

Computing via a post-fabrication and spatially as well as temporally programmed connection of processing elements [34]

In a reconfigurable computing system, both the hardware and software parts can be reconfigured depending upon the designer requirements. In [109], different terminologies such as *Configware*, *Morphware* and *Flowware* have been used in connection with the development of reconfigurable systems.

These reconfigurable systems normally consist of a matrix of large reconfigurable fabric (i.e. computational units), along with a dense reconfigurable routing network superimposed on the fabric. This fabric permits creation of custom functional units. These systems embed *fine* or *coarse-grain* elements (logical functions, operators, memory blocks, etc.) which are organized into clusters. A system might also incorporate some controlling processors which are coupled with, or embedded in the fabric. The processor(s) can execute non-critical code sequentially,

[6] http://en.wikipedia.org/wiki/Reconfigurable_computing

while key kernels of application that are time critical, exhibit high degree of parallelism and could be efficiently mapped to hardware; can be *'executed'* by processing units that have been allocated to the reconfigurable fabric, such as in the case of an FPGA. These mapped tasks/functions can take advantage of the parallelism achievable in a hardware implementation (such as hardware accelerators). Thus the interfaces between the processor(s) and the fabric, as well as the interfaces between the fabric and the available memory are of utmost significance. Finally, a system designer can develop the optimal combination of functional and storage units in the reconfigurable fabric, providing an architecture that supports the application.

1.3.1 Classification of Reconfigurable Computing Systems

A detailed overview of reconfigurable computing systems has been presented in [55] and [249]. The main characteristics that nearly all these systems have in common are normally spatial computation, distributed control, distributed resources and presence of a configurable datapath. A large number of reconfigurable systems have been conceived by researchers as well as the industry. These systems can be classified on the basis of several parameters [55]:

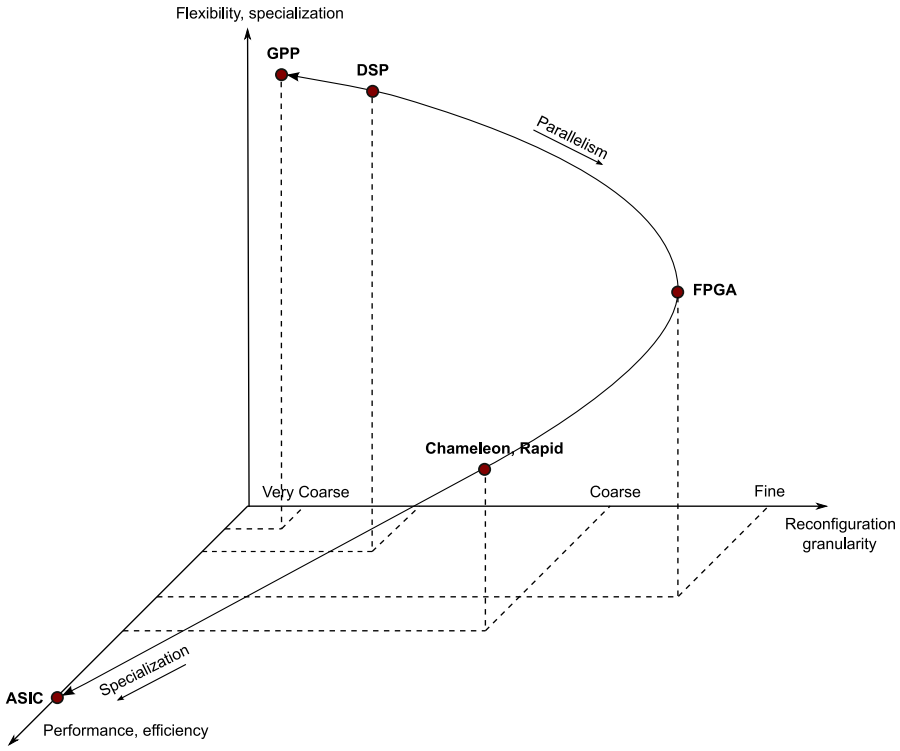


Figure 1.4: Architecture flexibility with regards to granularity and performance

- ***Granularity:*** The granularity of a RCS is defined as the size of its smallest functional unit. These systems can be either fine-grain (e.g: Xilinx Virtex Series FPGAs[7]) for control and bit oriented operations for a wide range of applications; or coarse-grain in nature. Coarse-grain architectures such as Chameleon [220] and Rapid [79] use word-width data paths and utilize functional levels of reconfiguration allowing to reach greater computational density and power efficiency, as compared to fine grain architectures. Also, the amount of needed configuration data is limited, but these architectures lose some of their utilization and performance when smaller computations are required that cannot be executed by their high granularity levels. They also target only precise domain specific applications that are known in advance enabling the logic, memory and routing resources to be customized. Architectures that fall in neither of these two categories are classified as medium-

[7] http://www.xilinx.com/products/v6s6.htm

grain (e.g. Garp [110], CHESS [153]). Figure 1.4 represents the architecture flexibility with regards to the performance and granularity of the above mentioned architectures.

- ***Degree of coupling:*** The degree of coupling can be of different types as illustrated in Figure 1.5 and determines the type of data transfers, latency, power consumption etc. A reconfigurable fabric such as an FPGA can be standalone in nature, relying on an I/O interface to communicate with a processor; that provides the control functions along with other additional tasks. This type of coupling is the loosest form of coupling. A reconfigurable fabric can be attached as an additional processor, or as a co-processor for forming a tight coupling. In a tighter form of coupling, a reconfigurable fabric behaves as a functional unit in the processor's data path. Finally the last form of coupling is different from the others in which a reconfigurable fabric has embedded processors as shown in Figure 1.6: e.g. an FPGA with embedded microprocessor(s).
- ***Types of interconnect networks:*** There can be a fixed network for communication between host and reconfigurable fabric, as well as a reconfigurable network for communication between configurable logical and functional blocks attached with the host. A popular style of interconnections can be found in current FPGAs which present an island style layout: configurable blocks are arranged in arrays with horizontal and vertical routing. Inadequate routing layout has drawbacks of poor flexibility and decreased performance. In contrast, a layout with too many interconnects requires more transistors, more silicon area, longer wires and hence more power consumption.
- ***Types of reconfiguration:*** Reconfiguration can be either *static* (passive) or dynamic (active) in nature. Both types of reconfigurations can either be full or partial depending upon the nature of the reconfigurable architecture and design specifications. These types of reconfigurations are discussed later on in section 1.3.3.1.

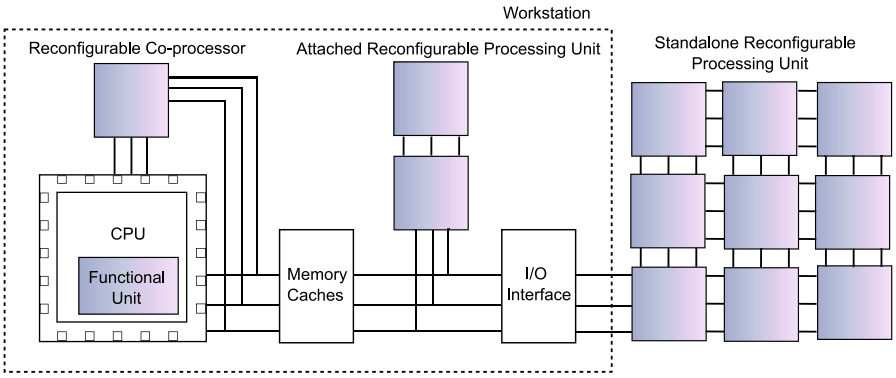


Figure 1.5: Different degrees of coupling [55]

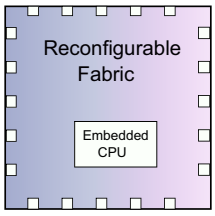


Figure 1.6: Tight on-chip coupling: Processor(s) embedded in a reconfigurable fabric

1.3.2 Reconfigurable Systems-on-Chips

The main difference between a classical SoC architecture and a reconfigurable SoC is the presence of reconfigurable areas: i.e., if FPGAs or *Complex programmable logic devices* (CPLDs) are

part of the Systems-on-Chip device. Others may be design specific only: these types of SoCs are thus implemented as ASICs. Classical ASIC based SoCs are normally designed to execute only one application with very tight performance constraints (latency, area, power consumption, throughput). Whereas reconfigurable SoCs are designed to execute different applications relying on same hardware capabilities. They thus introduce the notion of *virtual hardware*. Similarly in terms of fabrication, ASIC based solutions produce a extremely costly SoC with a long time-to-market requiring intervention by different teams and multiple designers, resulting in introduction of errors in the design cycle. The alternative solution is the utilization of FPGAs for construction of the reconfigurable SoCs.

A Reconfigurable SoC (RSoC) offers the same type of custom IP support except that the IP is implemented using the reconfigurable fabric. The software must set up the hardware before it can be used. Co-Design of these reconfigurable SoCs permit application partitioning as well as performance estimation techniques for evaluation of hardware/software implementations. Afterwards, integrated tools enable co-validation, co-simulation and design testing/de-bugging. DSE exploration of RSoCs effectively permits to find the optimal architecture for an application or applications family.

Figure 1.7 shows an example of a coarse grain reconfigurable SoC used in the European *MORPHEUS* (Multi-purpOse dynamically Reconfigurable Platform for Intensive HEterogeneoUS processing) project. This 100 mm^2 90 nanometer RSoC is composed of 97 million transistors along with an ARM9 microprocessor and three reconfigurable architectures (DREAM PiCoCA, Abound Logic eFPGA, Pact XPP matrix), memories, buses, NoC etc. The first prototype of MORPHEUS chip has been produced by STMicroelectronics earlier in 2009[8]. The complexity of such a large complex SoC architecture necessitates the use of an effective SoC design methodology.

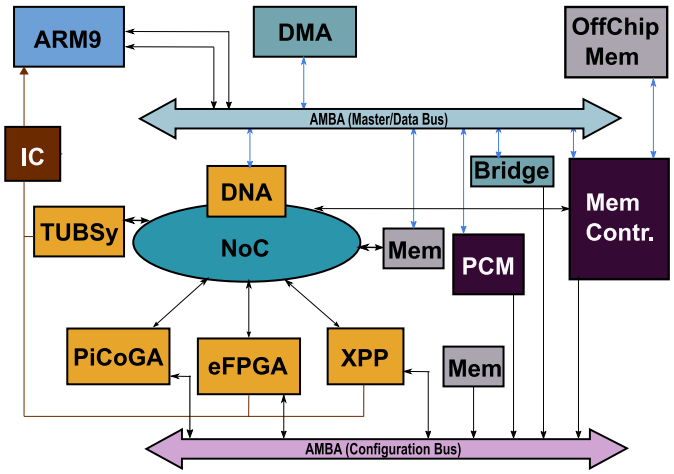


Figure 1.7: Morpheus: a dynamically reconfigurable SoC

1.3.2.1 Advantages and disadvantages

Reconfigurable FPGA based SoC designs have two main features that distinguish them from traditional SoC designs. The first is that the hardware functionality can be switched by modifying the corresponding executing configuration. So, a SoC can contain a digital-to-analog converter for one application, reconfigured for a analog-to-digital converter for another application; or even a completely different peripheral such as a network device. The second advantage is an offshoot of the first. Some elements of the reconfiguration can be performed at run-time once the initial configuration has been loaded allowing to handle issues related to fault tolerance and system performance. Also, reconfiguration can be carried out an arbitrary number of times after loading of the initial configuration.

[8] http://www.eetimes.com/showArticle.jhtml?articleID=21710013

FPGA based reconfigurable SoCs have minimal upfront costs as compared to custom designs implemented on ASICs. Design costs are reduced because changes can be immediately made to the chip during development phases before final fabrication. They are thus a popular choice for prototyping as designers can initially implement, and afterwards, reconfigure a complete SoC for the required customized solution. Thus these prototypes offer a path for final customized ASIC or SoC implementation. Similarly, chip simulation becomes less tiresome as the real hardware is available immediately. Other advantages include task swapping depending upon application needs, overcoming hardware limitations and Quality-of-Service (QoS) requirements fulfillment (power consumption, performance, execution time etc.). In [234], researchers found that moving critical software loops to reconfigurable hardware resulted in average energy savings of 35% to 70% along with a speedup of 3 to 7 times. Similarly, FPGA based RSoCs have been utilized in the Mars Spirit rover[9] to adapt to environments that require utmost durability.

The main downside of using a standard reconfigurable SoC is the cost compared to a custom SoC. The trade-off is related to the number of chips that will be shipped and any advantage for getting the product to market sooner. While custom designs normally have large up-front development costs, they offer low individual chip costs. Reconfigurable SoCs, on the other hand, have a comparatively small up-front cost but usually they are more expensive per single unit or chip.

1.3.3 Field Programmable Gate Arrays (FPGAs)

SRAM based Field Programmable Gate Arrays (FPGAs) are considered as an ideal solution for SoC implementation due to their reconfigurable nature: they can be reconfigured an unlimited number of times. These FPGAs usually consist of two layers. The first layer contains the reconfigurable logical blocks, i.e., *Configurable Logical Blocks* (CLBs) or *Logical Elements* (LEs) depending upon the FPGA vendor terminology. These logical blocks are present along with a hierarchy of reconfigurable interconnects allowing inter-communication between the blocks. This layer also incorporates different types of heterogeneous components such as RAM blocks, DSPs, multipliers, processors etc. All blocks of the same type (except the I/O blocks) are aligned into columns as shown in Figure 1.8. The second layer consists of the FPGA configuration memory layer. The configuration memory of FPGA contains the application specific data. Writing into a configuration memory is accomplished via configuration files known as *bitstreams* (that contain packets of configuration control information as well as the configuration data). Figure 1.9 shows the configuration memory layer for Xilinx Virtex-II/Pro series FPGAs. This layer matches the first layer and is also organized into columns. Each column whose width depends on the covered block-columns of the first layer is further composed into sub-columns called *frames*. For Virtex-II/Pro series FPGAs, a frame is the smallest unit of reconfiguration information which can be written on to the FPGA, while the more recent FPGAs such as Virtex-IV have smaller units of granularity [189]. Each frame contains fractions of configuration information required to configure the associated logical blocks assigned to a column.

In terms of reconfigurable architectures, FPGAs are given preference over CPLDs due to the presence of higher level embedded functions such as adders, multipliers and embedded memories. They are also more flexible due to the dominance of configurable interconnects, but with a cost of increased design complexity. FPGAs find their use in any area or domain where massive parallelism is a requirement. High performance applications also exploit FPGAs as key computational kernels for executing operations, such as FFTs and convolution.

Traditionally, a system design consisting of the hardware portion is specified using *Hardware Description Languages* such as VHDL or via a schematic design. The software portion related to hardware drivers and microprocessor code is written in assembly language or C/C++. The hardware/software aspects are usually developed using an Electronic Design Automation (EDA) tool such as Xilinx's Embedded Development ToolKit (EDK)[10], or Altera's SoPC Builder[11]. Afterwards, a technology mapped netlist is created followed by the process of *place-*

[9] http://www.xilinx.com/prs_rls/design_win/0412_marsrover.htm
[10] http://www.xilinx.com/ise/embedded/edk_pstudio.htm
[11] http://www.altera.com/products/software/quartus-ii/subscription-edition/design-entry-synthesis/qts-des-ent-syn.html#sopc

and-route (PAR) (it includes *floorplanning* which decomposes an FPGA into zones where hardware modules and other elements are to be placed). PAR tries to place and finally route the netlist onto a target FPGA. This process is strongly dependent on an FPGA vendor's proprietary place-and-route software. At each stage of the design process, simulation can be carried out to remove design faults and errors. Similarly, timing analysis and other verifications can also be accomplished. Afterwards, a bitstream is generated containing both the hardware and software portions, and can be loaded onto an FPGA via a serial-bit interface like *Joint Test Action Group* (JTAG) or a parallel byte interface such as SelectMAP [259]. An FPGA effectively incorporates DSE strategies based on its reconfigurable nature [35].

There exists a large number of research works for determining the effective placement of bitstreams onto an FPGA. We do not detail these works, which take into account different criteria, such as configurable resources, routing area, nature of integrated components (homogeneous/heterogeneous), abstraction levels, etc. A brief summary about these works has been presented in [143].

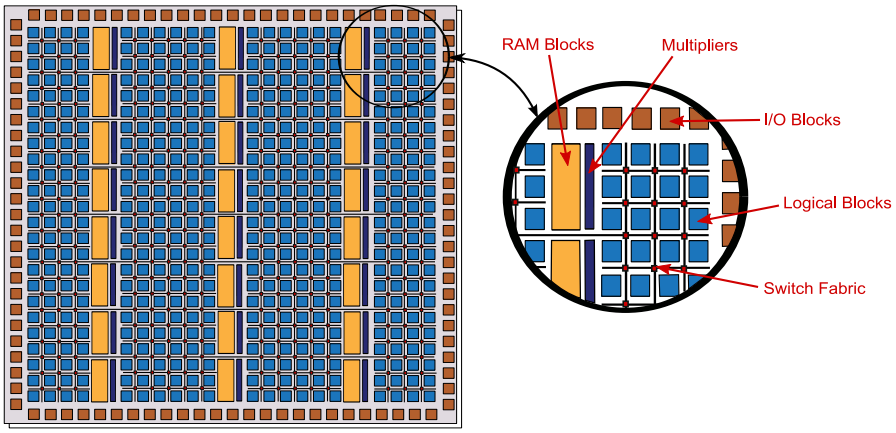


Figure 1.8: Configurable hardware layer in FPGAs

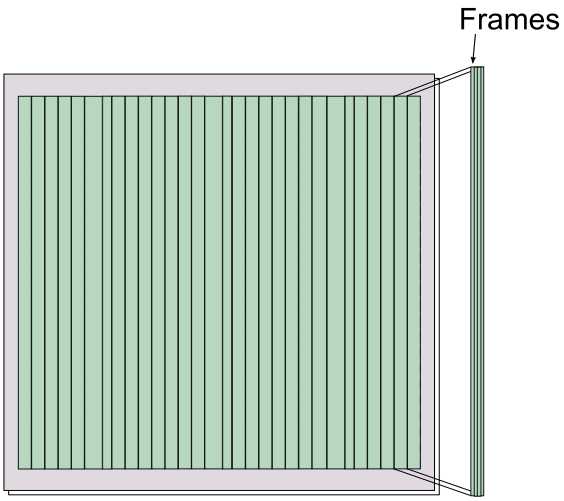


Figure 1.9: SRAM-Configuration memory layer for Xilinx Virtex-II/Pro series FPGAs

1.3.3.1 Types of reconfigurations

As research related to reconfigurable FPGAs is quite wide, a three-axis classification scheme is used to classify a reconfigurable approach by the community: mainly *where*, *when* and *how* the reconfiguration takes place. We briefly describe each point:

- ***Where***: Reconfiguration can either be *exo-reconfigurable* (external) or *endo-reconfigurable* (internal) in nature. In exo-reconfiguration, reconfiguration is initiated and controlled by an external source. An example is an FPGA co-processor on a PCI bus. Where as in an endo-reconfiguration, the FPGA itself loads the bitstream and reconfigures itself. In this case, usually an embedded controller such as a hard/soft processor manages the reconfiguration. For internal reconfiguration, special modules are needed inside the FPGA, which are detailed later on in the chapter.
- ***When***: The reconfiguration can be either *static or dynamic*. Static configuration requires the FPGA to be inactive, while dynamic reconfiguration is carried out on the fly when the FPGA is active and running. Dynamic reconfiguration can be directed by the application and offers flexibility advantages for systems where static reconfiguration is not possible, such as satellites.
- ***How***: The reconfiguration can be either *full* or *partial*. Full reconfiguration completely reconfigures the whole area of an FPGA, while partial reconfiguration concerns only a region or regions of FPGA while the remaining portions continue their normal execution. In a full reconfiguration, a full device bitstream is transmitted over a communication channel even in the case of minute changes, resulting in needless high data transfers. This is detrimental in bandwidth limited applications such as satellite payloads.

Figure 1.10 shows an overview of system reconfiguration. It should be noted that external reconfiguration introduces additional latency, which can prohibit or complicate dynamic reconfiguration. In contrast, internal reconfiguration is more suitable for rapidly evolving systems such as SoCs. *Internal Dynamic Partial Reconfiguration* or *Partial Dynamic Reconfiguration* (PDR) has shown tremendous advantages over other forms of reconfiguration. PDR or *self reconfiguration* [17] as it is sometimes called, has been explored only recently, and is the reconfiguration type addressed in this dissertation.

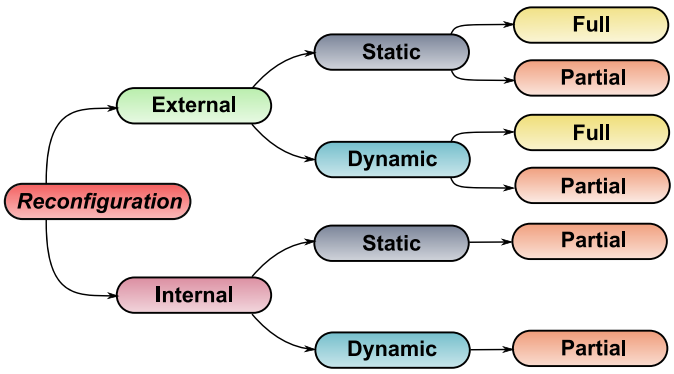


Figure 1.10: An overview of different types of reconfiguration

1.3.4 Partial Dynamic Reconfiguration (PDR)

PDR enables modification of specific regions of an FPGA on the fly, with the advantage of time-sharing the available hardware resources for executing multiple (mutually exclusive) tasks. Some regions can be reconfigured dynamically, while others regions remain operational and functioning. PDR enables context swapping depending upon application needs, hardware limitations and Quality-of-Service (QoS) requirements, such as power consumption [122, 192, 193], performance, execution time; etc. Partial reconfiguration shows tremendous potential for wide variety of applications across different industries. The aerospace, telecommunications and defense industries have taken a particular shine to this feature. PDR has also influenced the Software Defined Radio (SDR) domain [124]. It allows to share multiple applications on the same FPGA with benefits such as reduced energy consumption, fault tolerance: such as in case of single event upsets (SEUs); and reduced overall costs.

To the best of our knowledge, until the writing of this thesis in 2009, PDR is only fully supported by FPGAs fabricated by Xilinx. Xilinx provides a comprehensive tool support for PDR, as compared to other vendors that have chosen not to incorporate this feature due to reliability and economical issues. Xilinx FPGAs also support internal self dynamic reconfiguration, in which an internal controller (a hardcore/softcore embedded processor) manages the reconfiguration process [17].

Xilinx initially proposed two methodologies (difference based and module based) [257, 258] followed by the *Early Access Partial Reconfiguration* (EAPR) flow [260] in 2006. We provide some brief details related to these methodologies:

1.3.4.1 Difference based Partial Reconfiguration

This type of partial reconfiguration is carried out by making small modifications to a system design. Afterwards, a bitstream is generated based on only the *difference* between the *before* and *after* designs. The advantage of this approach is that configuration switch of a module between the different designs is very rapid, as bitstream differences can be extremely minute in comparison to a bitstream for configuring the whole FPGA. However the drawback of this approach is that a designer has to edit the design with low level editing tools such as *FPGA editor*, by fine tuning components such as Lookup tables (LUTs), RAMs, FlipFlops. This requires an in-depth knowledge of the underlying FPGA architecture. Similarly this solution is not suitable for complex designs requiring large reconfigurable areas; and signal integrity cannot be ensured on reconfigurable modules boundaries.

1.3.4.2 Module based Partial Reconfiguration

This method is based on the Xilinx modular design flow [257, 258]. This feature allows to split up the system design into multiple modules, permitting different designers to work independently on different modules. These modules can either be static or dynamically reconfigurable in nature. Afterwards, the modules can be merged together to form the complete FPGA design. This methodology gains in time savings and enables modification of a module independently from others in case of faults or user requirements. An initial full configuration bitstream is required for initial bootup along with partial bitstreams for each partial reconfigurable region. For swapping a reconfigurable module with another, their external interfaces must remain the same. The drawback of this approach is that a reconfigurable module occupies the full height of the device, including the I/O blocks at the top and bottom of the reconfigurable module. Similarly, driver contentions can occur if a module is written over immediately with another, which can be avoided by first replacing a module with a default empty configuration before loading the next module. Similarly, for large devices, full height of a reconfigurable module may not be desired due to inefficient use of resources. In addition, there is a high probability that a static signal path through a reconfigurable module will be re-routed during reconfiguration, making the design non valid. If the modules are completely independent of other modules or static portion of the FPGA: i.e, no common I/O except clocks; then no special mechanisms such as *bus macros* are needed for inter-module communication.

Communication modules for Partial Dynamic Reconfiguration. *Bus macros* [258] which are relationally placed macros (RPMs), are used to ensure proper communication routing between the static and dynamic regions during and after reconfiguration. They are physical ports that connect a reconfigurable module with other modules in the design. Initial design methodologies recommended tri-state buffer (TBUF) based bus macros which were hard-wired into the architecture of Xilinx FPGAs [257]. However, this resulted in several drawbacks due to the predetermined locations of TBUFs, along with the fact that they were dispersed across the FPGA architecture, introducing increased delays.

Xilinx then introduced CLB based bus macros in [260]. In Xilinx FPGA architectures, CLBs are considered the basic primitives and are abundant in nature, allowing to create bus macros for any architecture along with increased flexibility in their placement. The bus macros provide a unidirectional 8-bit data transfer and have a specific signal direction. A condition for initial

bus macros was that all connections between partially reconfigurable module(s) and static design must pass through the macros, with the exception of clock signals. However, with new bus macros [260], static nets can also cross through reconfigurable modules.

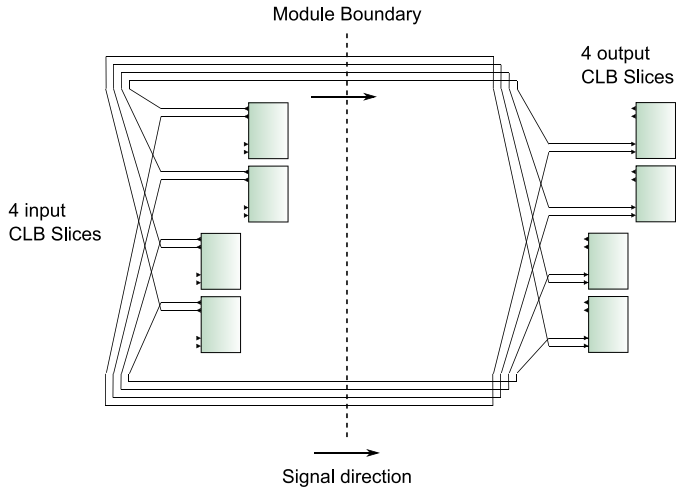


Figure 1.11: Basic 8-input, 8-output left-to-right bus macro

Types of bus macros. Several different types of bus macros are available for each FPGA architecture depending upon their *signal direction* and *width*. Regarding *signal direction*, for the Virtex-II/Pro FPGAs, the bus macros can be either *left-to-right* or *right-to-left*. Virtex-IV series FPGAs also include *top-to-bottom* and *bottom-to-top* directional bus macros. Whether a bus macro act as an input/output to the reconfigurable module depends directly on its signal direction and placement. For example, a *right-to-left* bus macro placed on the right side of the reconfigurable module acts as an input, while the same bus macro placed on the left side acts as an output. The *width* of a bus macro can be varied as well. Bus macros can either be *narrow* or *wide* in nature. A *narrow* bus macro is constructed of 2 CLBs as compared to a wide bus macro constructed of 4 CLBs. Here *width* refers to the physical width of a bus macro and not of the data bandwidth. Wide bus macros can be nested or chained together in a single CLB column for increased bandwidth. For more complex designs with many I/O pins on the reconfigurable modules, wide bus macros are desirable to save vertical space. Finally bus macros can be either *synchronous* or *asynchronous* in nature. Synchronous bus macros register signals passing through them and provide better timing performance, and are recommended by Xilinx. Lastly, bus macros provide an optional bus macro enable control signal. When this signal is de-asserted, i.e., set to 0, the bus macro outputs are set to 0 as well. Since output signals for a partially reconfigurable module are unpredictable during partial reconfiguration, it is recommended that the enable signal be de-asserted and asserted before and afterwards loading a partial bitstream respectively.

Bus macro placement. Bus macros are normally placed in a manner that one CLB is placed inside the reconfigurable region while the other is outside in the static region. For modern state of the art Virtex-V FPGAs, single CLB based bus macros are also available [261]. Figure 1.11 shows an example of a bus macro for Virtex-II/Pro FPGAs. It consists of 2 CLBs, one on each side of a module boundary between a static and a dynamically reconfigurable region. Each Virtex-II/Pro CLB contains four logical slices, with each slice implementing two unidirectional connections (from left to right in case of the Figure 1.11). Details about these bus macros can be found in [260]. Virtex devices also support the feature of *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, with the exception of LUTRAMs and SRL16 primitives [189]. This limitation was removed in the Virtex-IV family. With the introduction of EAPR flow tools, this problem has also been resolved for Virtex-II/Pro FPGAs and designers do not have to explicitly exclude these resources from the reconfigurable module(s).

Third party bus macros. [190] presented a modular approach that was more effective than the initial Xilinx methodologies and were able to carry out 2D reconfiguration by placing hardware cores above each other. The layout (size and placement) of these cores was predetermined. They made use of reserved static routing in the reconfigurable modules which allowed signals from the base region to pass through the reconfigurable modules permitting communication by using the principle of *glitchless dynamic reconfiguration*.

[122] implemented 1D modular reconfiguration using a horizontal slice based bus macro. All the reconfigurable modules that stretched vertically to the height of the device were connected with the bus macro for communication. They followed by providing 2D placement of modules of any rectangular size by using routing primitives that stretch vertically throughout the device [148]. A module could be attached to the primitives at any location, hence providing arbitrary placement of modules. The routing primitives are LUT based and need to be reconfigured at the region where they connect to the modules. A drawback of this approach is that the number of signals passing through the primitives are limited due to the utilization of LUTs. This approach has been further refined in [46].

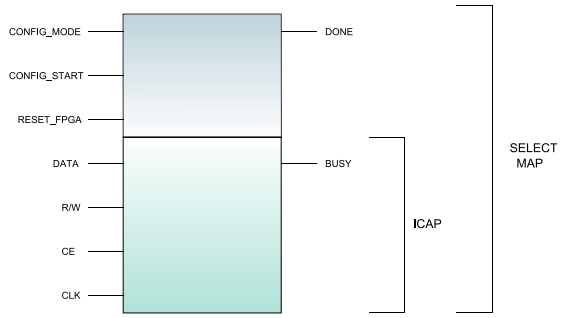


Figure 1.12: An overview of Internal Configuration Access Port (ICAP)

ICAP reconfiguration core for Dynamic Reconfiguration. At the heart of the PDR mechanism lies the *Internal Configuration Access Port* (ICAP) [17], which is a subset of the SelectMAP 8-bit parallel interface as shown in Figure 1.12. While PDR can also be carried out using SelectMAP or the JTAG bit-serial interface, the reconfiguration times are much higher as compared to the ICAP based approach [242]. Additionally, SelectMAP can only be used for carrying out external partial reconfiguration.

The ICAP is an integral component that permits to modify the FPGA configuration memory at run-time. It is only used for partial dynamic reconfiguration and cannot be used for full FPGA configuration; and thus enables self reconfiguration. Special care must be taken during dynamic reconfiguration of a particular device so as not to reconfigure the ICAP circuitry. The *granularity* of reconfiguration is also of importance. In the Virtex-II and Virtex-II Pro series FPGAs, the smallest unit of reconfiguration granularity is a frame. The number of bits present in a frame is directly proportional to the height of the device that is measured in CLBs. For example, for Virtex-II series FPGAs, the number of bits per frame ranges from 832 (the smallest device) to 9152 (the largest device) bits per frame. In the Virtex-IV series FPGAs, the smallest unit of reconfiguration granularity is a bit-wide column corresponding to 16 CLBs (or multiples, and this unit is independent of the different device sizes or families). A configuration frame of a Virtex-IV series FPGA contains forty one 32-bit words (1,312 bits per frame). The smaller granularity size allows more than one dynamically reconfigurable module to be placed vertically in the same region of FPGA. This is not possible in the earlier series FPGAs such as Virtex-II/Pro.

The ICAP is present in nearly all Xilinx FPGAs ranging from the low cost Spartan-3A(N) to the high performance Virtex-V FPGAs [22]. For Virtex-II and Virtex-II Pro series, the ICAP furnishes 8-bit (1 byte) input/output data ports while with the Virtex-IV Series, the ICAP interface has been updated with 32-bit input/output data ports and the width can be alternated between 8 and 32 bits [1]. The ICAP cannot be directly connected to a system bus as it requires a controller to manage the data flow coming from the reconfiguration controller. A classical

ICAP controller is presented inside Xilinx's EDK OPB HwICAP module [262], which serves as a wrapper for the ICAP core and allows its interfacing with a global system controller such as an embedded PowerPC hardcore processor. Hence it abstracts the low level details of the ICAP and is connected to the *On-Chip Peripheral Bus* (OPB) [12] as a slave peripheral. Another version is the PLB ICAP [44] attached to the *Processor Local Bus* (PLB) [13], however this dissertation focuses on the former version. For Virtex-II/Pro series FPGAs, only one ICAP core is present which is located on the lower right side of the FPGA. For Virtex-IV and more modern FPGAs, two ICAPs are presented that are normally located in the center of the FPGA. A study related to design space exploration of ICAP architecture has been presented in [146].

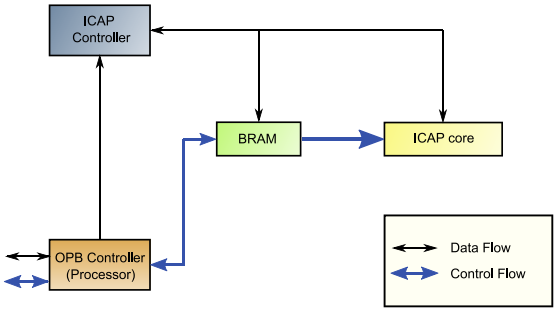


Figure 1.13: An overview of OPB HwICAP core

Read-Modify-Write mechanism. The ICAP utilizes a read-modify-write (RMW) [147] mechanism that enables the global controller (a hardcore PowerPC or a softcore Microblaze processor) to modify the bitstream related to the reconfiguration module dynamically with the help of the HwICAP module. The OPB HwICAP module provides access to the ICAP core for reading or writing configuration data from the configuration memory. It also provides the C/C++ software libraries for the ICAP core. The software library includes functions that permit initializing, reading configuration data, writing a frame; and writing a partial bitstream, to the ICAP. The module internally consists of an ICAP controller, a Block-RAM (BRAM) memory and the ICAP core. The ICAP controller regulates the data and control flow between the ICAP core, the reconfiguration controller and the BRAM memory. The exchange of data (configuration packages) is carried out by the reconfiguration controller and the ICAP core via the BRAM. Usually the BRAM memory has sufficient capacity (several KBs) [148, 205] to at least store one configuration frame related to the region being reconfigured.

The BRAM is used to store the configuration data read from the device. A configuration frame related to some FPGA resources is read out from the device's configuration memory and stored in the BRAM. Afterwards, the reconfiguration controller manipulates bits inside the frame, related to the FPGA resources. Finally the partially modified frame is written back to the configuration memory. The combination of the ICAP with an internal reconfiguration controller allows to build a self controlling dynamically reconfigurable system [17].

Reconfiguration time during dynamic switching is directly proportional to the average throughput of the ICAP. The theoretical speed of the ICAP is 100 MHz [43], however, in reality it is not always the case. For example in Virtex-II/Pro FPGAs, when the ICAP is clocked over 50 MHz (100 MHz in Virtex-IV FPGAs), it is necessary to respect the ICAP's handshaking (busy) signal [44]. Thus maximum theoretical throughput cannot be achieved, as compared to the real throughput which is about 94% to 96%.

1.3.4.3 Early Access Partial Reconfiguration Flow

In March of 2006, Xilinx introduced the *Early Access Partial Reconfiguration (EAPR)* [260] design flow along with the introduction of CLB based bus macros and possibility of creating 2D

[12] http://www.xilinx.com/products/ipcenter/OPB_Bus_Structure.htm
[13] http://www.xilinx.com/products/ipcenter/PLB_Bus_Structure.htm

reconfigurable modules, thus resolving the drawbacks present in the earlier modular design methodology. The concepts introduced in [190] and [122] were integrated in this flow. The restriction of full column dynamic modules was removed allowing reconfigurable modules of any arbitrary rectangular size to be created. The EAPR flow also supports static nets to cross through the partially reconfigurable region(s) without the use of bus macros. This improves timing performance, clock tree management; and simplifies overall design construction.

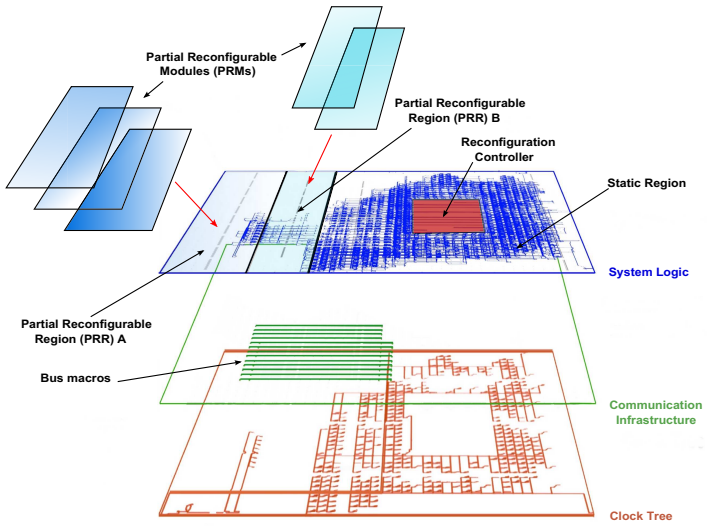


Figure 1.14: An overview of Partial Dynamic Reconfiguration: physical implementation in three different layers

Terminologies related to EAPR. In this dissertation, respecting the terms as specified by Xilinx, a region of the FPGA to be reconfigured dynamically is termed as PRR or *Partial Reconfigurable Region*. A PRR can have several possible implementations or *Partial Reconfigurable Modules* (PRMs). An important point to consider is that all the PRMS of the PRR have the same external interface to ease compatibility. We now utilize these terms during the course of this dissertation. Using the principle of glitchless reconfiguration, no glitches will occur in signal routes as long as they are implemented identically in every reconfigurable module for a PRR. The only limitation of this approach is that all the partial bitstreams (PRMs) to be executed on a reconfigurable region (PRR) must be predetermined. Additionally, the output frequency and phase shift of a digital clock manager (DCM) can be modified as well.

Usually an initial bitstream is loaded on to the FPGA which consists of the static portion as well as an initial PRM for the PRR(s). Afterwards, the controller only has to load the partial bitstream related to an alternate PRM for the same PRR. Using glitchless reconfiguration and the RMW mechanism, the *difference* between the two PRMS is noted and written back by the controller resulting in implementation of the new PRM. Figure 1.14 shows an abstract overview of a partial reconfigurable system.

The partial bitstream files to update the configuration memory can either be provided from outside or inside the system. External Flash memory can be used to permanently store the partial bitstreams depending upon their size. Bitstreams of small sizes can be placed on on-chip BRAM memories, where as large bitstreams for complex systems can be stored on off-chip memories such as Flash and SDRAM memories. As the Flash memory is usually very slow (it can only deliver several bits at small frequencies: such as 8 bits at 10 MHz in the case of Virtex-IV FPGAs), SDRAM memories are preferred. During initial bootup, the partial bitstreams from a Flash memory can be transferred to a SDRAM to reduce reconfiguration times, however this complicates the design process and reconfiguration management.

1.3.4.4 Research related to PDR

We now detail some works in the domain of partial dynamic reconfiguration. This is not an exhaustive collection and mentions just some significant contributions. Works related to PDR can be categorized in several categories: Some research works try to elevate design abstraction levels, such as providing specifications in system level languages like SystemC[14]; for decreasing the complexity related to creation of dynamically reconfigurable systems. Others deals with optimization directly at the *Register Transfer Level*[15] (RTL) level by introducing new tools and methodologies. We now provide an overview of some of these works.

Elevation of design abstraction levels. The MoPCoM project [3] aims to target modeling and code generation of dynamically reconfigurable embedded systems using the MARTE UML profile for SoC Co-Design [127]. However, the targeted applications are extremely simplistic in nature, and do not represent complex application domains normally targeted in the SoC industry. Similarly, while the authors claim that they are capable of creating a complete SoC Co-Design framework, in reality, the high level application model is converted into an equivalent hardware design, with each application task transformed into a hardware accelerator in a target FPGA. Additionally, while the project permits modeling of the targeted FPGA architecture at the UML level as inspired from the works presented in [203, 205], they are only capable of generating the *microprocessor hardware specification* file (.mhs) for input in Xilinx EDK tool for manual manipulation of the PDR flow. Moreover, IP re-use in not possible with this methodology.

In the OverSoC project [218], the authors also provide a high level modeling methodology for implementing dynamic reconfigurable architectures. They integrate an operating system for providing and handling the reconfiguration mechanism. The global platform is conceptually divided into *active* and *reactive* components representing the reconfigurable architecture (an FPGA) and the OS respectively. The OS is executed on a general purpose processor (GPP) interfacing with the FPGA. The active component is further composed of several sub components that represent the computation and reconfiguration components, the former relating to FPGA resources such as CLBs, LUTs, etc., while the latter corresponding to the ICAP core. Finally, SystemC was used for simulation and verification of the OS for managing the reconfigurable aspects. However, final implementation on FPGAs has not been carried out, and the OS determines whether an application task should be executed on the GPP or the FPGA depending upon its required resources. A more complex OS is presented in [168], as embedded uCLinux as an RTOS is used for managing PDR. A customized device driver has been created to mange the ICAP core, allowing users to carry out dynamic configuration in traditional Linux shell programs. However, the bitstreams are generated manually using the FPGA editor tool, raising chances of design errors.

[38] uses a SystemC based design flow for implementing PDR. The SystemC kernel was modified for the integration of reconfiguration operations for activation/dis-activation of the reconfigurable modules. Initial simulation is carried out using a SystemC model, which is then converted into a HDL RTL model for actual implementation and comparison. The drawback of this approach is that the reconfiguration time related to module is predetermined by the designers. Additionally, with respect to PDR, the system only provides on-off functionality for the modules resulting in a simplified design. In contrast, [170] use HandleC in order to implement PDR for Software defined Radio (SDR), however, they only provide the design methodology and no actual implementation is carried out.

Optimizing details at RTL. Works such as [22] and [130] focus on implementing softcore internal configuration ports on Xilinx FPGAs such as the pure Spartan-3 that do not have the hardware ICAP core rendering dynamic reconfiguration impossible via traditional means. In [130] a soft ICAP known as JCAP (based on the serial JTAG interface) is introduced for realizing PDR while [22] introduces the notion of a PCAP, based on the parallel SelectMAP interface, providing improved reconfiguration rates as compared to the JTAG approach. However this approach is only suitable to reconfigure very small regions of FPGA and since the design is not

[14] http://www.systemc.org/
[15] http://en.wikipedia.org/wiki/Register_transfer_level

an embedded one, it is impossible to retrieve bitstreams from an external memory. This issue has been addressed in [78], where a complete reconfigurable embedded design on a Spartan-3 board has been implemented using a reconfigurable co-processor. The results show that this achieves a compromise between the works presented in [22] and [130].

In [44], a new framework is introduced for implementing PDR by the utilization of a PLB ICAP. The ICAP is connected to the PLB as a master peripheral with direct memory access (DMA) to a connected BRAM as compared to the traditional OPB based approach. This provides an increased throughput of about 20 percent by lowering the process load. However, no support for the read-modify-write mechanism is presented, nor it is possible to access the ICAP in a virtual manner. An improved version of the PLB ICAP was introduced in [43] with a speedup factor of 18 to 58 times depending upon the reconfiguration scenario. However, it also suffers from the same disadvantages as its precedent version. [4] provides another flavor of a PDR architecture by attaching a reconfigurable hardware accelerator to a softcore Microblaze processor via a *Fast Simplex Link*[16] (FSL) bus. This resulted in an area economization of 29.6% with a performance loss of 1.5%, as compared to placing all hardware cores on the target FPGA. However, this architecture is only valid for the chosen JPEG encoder and cannot be used for different range of applications. In comparison, [49] advocates the use of OPB co-processors that are attached to the Microblaze controller. The authors use OPB bus for connecting the modules as compared to the normally preferred FSL bus. The advantage being that microprocessor uses regular read/write instructions in contrast to specific FSL instructions; which is balanced by the complexity of the design and increased reconfiguration times.

In [1], a customized ICAP controller is presented in order to speed up the reconfiguration process depending on a specific reconfiguration scenario. This controller can be implemented as either a PLB or an OPB ICAP and offers the possibility of different memory implementations: slices or BRAMs. The ICAP controller has an additional FIFO that stores the incoming data consisting of 4 bytes (in case of 32-bit data transfer) that are sent to the ICAP core one by one for processing. Nevertheless, the metrics related to the customization only take the architectural aspects into account.

In [53], the authors present new *Xilinx Definition Language* (XDL) based bus macros, similar to the approach specified in [122, 148]. The advantage of this approach is that customized data width bus macros can be generated. While the works focus on generation of bus macros, their placement is user dependent; and does not reduces the complexity related to the creation of a dynamically reconfigurable system.

In [131], the REPLICA filter has been introduced which uses the SelectMAP interface for bitstream manipulation, for implementing PDR in the RAPTOR2000 platform. However, the reconfigurable modules only have a mono-dimensional shape extending to the height of the targeted FPGA. As well, the framework uses traditional TBUF bus macros; and the configuration manager responsible for the configuration switch is implemented in a CPLD inside the platform. This introduces additional reconfiguration overhead. Similarly, [115] introduces the notion of *Dynamic Hardware Plugins* (DHPs). However this methodology targets the earlier Virtex-E series FPGAs, and has not been tested on modern commercially available FPGAs.

Works such as [208] use ICAP to connect with a Network-on-chip (NoC) to allow distributed access to speed up reconfiguration time. However the read-modify-write mechanism is not supported which is an important factor to speed up the time period. This limitation has been resolved in [45] where an ICAP communicates with a NoC using a light weight RMW method. The reconfiguration times have been reduced to about 40 microseconds per frame of the multiple FPGA nodes in the NoC architecture. However, the works focus primarily on architectural aspects and the targeted application domains are not mentioned.

Tools and technologies for supporting PDR. Several tools have also been introduced for facilitating the implementation of PDR in FPGAs. Initial tools such as PARBIT [114] and JBits [156] manipulate FPGA configuration bitstreams. However they lack support for modern FPGAs and require tedious manual low level manipulations. PARBIT allows to relocate reconfigurable modules but with an extremely high reconfiguration overhead. XPART [33], a Xilinx proprietary tool, seemed promising for PDR, but was never actually released. In [133], the *ReCoBus-Builder*

[16] http://www.xilinx.com/products/ipcenter/FSL.htm

has been introduced, which permits to generate communicating infrastructure at run-time for the reconfigurable modules; and incorporates final bitstream generation features. Similarly, in [223], the *ReconfGenerator* has been developed, resulting in a completely automatic design flow as compared to Xilinx EAPR flow. Bus macro placement is done automatically by the generator resulting in optimized results. Similarly, synthesized netlists for reconfigurable modules are translated, mapped, placed and routed automatically without user interaction. In [52], *CombitGen* is illustrated, which improves on the Xilinx's *Multi Frame Write* (MFWR) mechanism of writing frames; and enables writing of single unique frames resulting in shorter bitstreams and improved reconfiguration times. The only disadvantage of this approach is that if every frame is to be configured in a column of configuration memory, the tool offers no optimization over the current EAPR flow. The main problem related to all these above mentioned tools is that they are not currently open source or compatible with each other.

Other issues related to PDR. There is a large number of issues related to implementing PDR. Runtime relocation of partially reconfigurable modules has been addressed as well, which allows PRMs of different regions to be swapped by one other. Detailed works related to this approach and related problems have been presented in [24, 123, 209]. Similarly other issues such as compression of partial bitstreams for reducing reconfiguration times have been addressed in works such as [191].

1.3.4.5 Summary related to partial dynamic reconfiguration

As seen in the section related to partial dynamic reconfiguration and the related research, nearly all of the current efforts have been focusing on the low level architectural details of the targeted reconfigurable platforms. The designers normally concentrate on the problems related to the platforms and forget about the applications which are to be executed on these architectures. While some of these architectures are custom built to focus on specific applications, their drawback lies on the strong dependency on the required application. Hence, they are not able to execute a wide range of applications. There is thus a strong need to switch the designer perspective from the architectural aspects towards an application driven approach. Only [48] shares the same perspectives as this dissertation, and tries to present an application-centered high modeling level approach using Simulink HDL coder[17]. However, in reality, this modeling level is still not abstract enough to be totally independent of low technological details.

1.4 Challenges for SoC Co-Design

1.4.1 Productivity issues

According to Moore's law, rapid evolution in hardware technology doubles the number of transistors in an Integrated Circuit (IC) nearly every two years. As the computational power increases, more functionalities are expected to be integrated into the system. 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 facing SoC Co-Design. The fallout of this complexity is that the system design (particularly software design) does not evolve at the same pace as that of hardware due to issues such as development budget limitations, reduction of product life cycles and design time augmentation. It has been estimated that the productivity of developers does not augment more than 30% per year. Current hardware technology allows to integrate more than twenty million gates in a single chip, making it quite possible that the chip will be under utilized by the targeted application. This evolution of balance between production and hardware/software design has become a critical issue and has finally led to the famous *productivity gap*.

According to the ITRS (International Technology Roadmap for Semiconductors) [121], this productivity is based on the following elements: communication between hardware/software designers, component re-utilization, validation and tests, platforms for SoC conception and

[17] http://www.mathworks.com/products/slhdlcoder/

implementation; and finally the SoC design processes. SoC design tools maintainability and evolution is also one of main concerns for SoC designers.

System reliability and verification are also the other issues related to SoC industry and are directly affected by the design complexity. Design correctness is an important factor in SoC design, since it has a great impact on system performance, overall results, time-to-market, cost, etc. For example, incorrect decisions taken at the SoC mapping design level can have catastrophic performance results. For critical systems, ignoring design correctness can cause disasters resulting in loss of human lives.

However, SoC design correctness always remains one of the most difficult challenges to overcome, as SoCs are becoming more complex. Validation accounts for about 70% of the overall chip design cost, even so, design teams always deliver chips late and miss projected deadlines due to verification problems. Simulations and tests offer a compromise between the verification quality and cost. They are considered as partial solutions to the SoC productivity problem nowadays. It is evident that new and effective SoC Co-Design methodologies are required that increase the productivity of SoC designers.

1.4.2 Reconfigurability issues

Reconfigurability in SoCs adds another level of complexity to the already challenging SoC design aspects. System level modeling of reconfigurable SoCs must be efficient in order to combat factors such as time-to-market and fabrication costs. Effective methodologies need to be developed in order to manage the hardware/software resources for full or partial dynamically reconfigurable SoCs while offering seamless interfaces to the users. Overheads typically associated with reconfiguration such as reconfiguration times, load balancing, scheduling reconfigurable tasks must be reduced.

Robust control mechanisms for managing reconfiguration must be developed in order to take advantage of SoC designer needs as well as QoS choices. These choices can be: changes in executing functionalities, e.g., color or black and white picture modes in a video processing application; 2) changes due to resource constraints of targeted hardware, 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 mechanism must be generic enough to be applied to both software and hardware design aspects.

Although PDR is fastly growing and gaining popularity in the domain of reconfigurable computing, there is not sufficient initial information for a novice designer to implement this feature. Also the tools for implementing PDR are still in the development phases: for example, the EAPR flow is still not available for the Xilinx Virtex-V and the recently released Virtex-VI series FPGAs [263]. Similarly, as seen earlier in the chapter, dynamic reconfiguration is more architecture driven as compared to being application oriented, and is influenced by target platform low-level details. These issues such as bus macro placement, have a drastic impact on overall reconfiguration times, performance etc. Similarly the choice of selecting an appropriate location for a particular partial reconfigurable region (PRR) is designer dependent and also impacts the overall results.

These architectural details introduce increased complexity for the construction of dynamically reconfigurable FPGA based SoCs, resulting in exponentially escalating system development times. Similarly, designers need to master various different tools in order to implement PDR resulting in increased time-to-market, system complexity as well as errors. Thus, there is a critical need to abstract the low level details for addressing these issues.

1.4.3 Responding to challenges of SoC Co-Design

Many partial solutions to tackle the aforementioned issues have been proposed such as system-level hardware/software Co-Design, IP reuse, behavioral synthesis, softwarization (memory, processor), etc. However, neither of these approaches can provide a complete optimal solution.

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.

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 [77]. 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 reused across different software products.

Current SoC Co-Design practices mix many of these approaches to obtain the maximum design efficiency. For example, IP re-use, hardware/software Co-Design, high-level abstraction can be proposed in a framework in order to benefit from the advantages provided by the three approaches: IP re-use helps to separate concerns so that unacquainted work can be accomplished by certain experts of that domain; hardware/software Co-Design enables concurrent design, moreover, hardware design becomes similar to software design, for instance, software programming languages are extended for the hardware design (e.g., HandelC, SystemC, etc.), which reduces the complexity by using the same programming languages; high-level abstraction contributes in designing a system without too many implementation details.

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) [186] 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* [237] can be carried out to generate executable models from high level models. MDE is supported by several standards and tools.

Dynamic reconfigurability also possesses a challenge. A effective reconfigurable SoC must have design methodologies which allow to express some reconfigurability aspects at high abstraction levels as compared to manipulation at lower technology levels. Unfortunately, due to tool limitations of current dynamically reconfigurable FPGA based SoCs and lack of knowledge related to these proprietary architectures, it is often difficult to bridge the gap between high abstraction levels and low implementation details.

This thesis provides some answers related to the issues associated with SoC productivity and integration of reconfigurable features. A model-driven high level component based methodology seems promising as it integrates advantages provided by several partial solutions such as IP reuse and elevation of design abstraction levels. With respects to current problems plaguing the dynamic reconfiguration domain, we propose an alternative approach to shift the designer focus from purely architectural point of view to application based perspectives.

1.5 Conclusions

This chapter gives a brief overview of SoCs, their reconfigurability features and issues regarding to their design. As mentioned previously, SoCs offer numerous advantages over traditional computing systems, which include reduced energy consumption, small physical size, improved performance and throughput, etc. Thus, SoCs are increasingly found in real-time embedded systems. However, traditional SoCs offer less flexibility with regard to general computing systems, particularly when new applications are to be integrated into SoCs. This limitation can be removed by adapting aspects of reconfigurability in modern SoCs. Partial dynamically reconfigurable FPGA based SoCs are the wave of the future and address issues such as limited hardware resources, hardware/software evolution and fault tolerance. These SoCs follow the principles of hardware and software Co-Design approaches. The SoC validation is carried out thorough repetitive simulation and verification, until the systems are completely considered to be validated.

In spite of the widespread usage, SoCs also face several challenges. The main challenge is the productivity issue, which is caused by the unbalanced evolution of hardware as well as software. Another dilemma is the validation issue, which is the most time-consuming stage in the SoC design. Integration of reconfiguration aspects, while offering tremendous long term advantages,

complicate the already delicate design process. Despite these challenges, new methodologies and technologies are continuously proposed to meet the requirements of SoC design. Particularly, component based system design and elevation of design abstraction levels are highly encouraged. In the context of this thesis for addressing reconfigurable SoCs; a design approach based on component based methodology and MDE driven high abstraction levels is taken into account, which are presented in the following two chapters.

Chapter 2

Component Based Design

This chapter provides a brief overview of component based design which is an existing and popular design methodology mainly used in the software engineering domain. As mentioned in the previous chapter, design complexity of SoCs can be reduced by developing the interior parts as components or modules in order to separate the different concerns. Features such as system hierarchy are more evident with this approach. We first provide a brief introduction of components, followed by some key concepts such as component models and component frameworks. Afterwards, reconfigurability in component based design is addressed which is normally present in different granularity levels. Moreover, a brief overview of different component models existing in literature is given with special focus on heterogeneity and adaptation. Finally we discuss the usage of component based design in the conception of embedded systems and the related challenges.

2.1 Components

Components are widely used in the domain of *Component Based Software Development* (CBSD) or *Component Based Software Engineering* (CBSE). The key concept is to visualize and structure the system as an explicit composition of units or *components* [77]. A component based design approach permits to increase software *productivity*, as it reduces efforts to conceive, develop, update and maintain large complex systems. A component technology offers the following crucial advantages:

- ***Modular based approach:*** A component based approach gives a modular structure to the overall system design. The system can be divided into several small subsystems that enables their independent design, thus making system debugging, verification and maintenance more tractable. A component should be clearly (formally) specified and must be comprehensible. This modular approach also achieves separation of concerns related

to the system at hand. A specific component can be used to implement non-functional services, enabling separation of functional and non-functional concerns.

- ***Re-usability:*** Components can be re-used across different software products and applications, many of them not yet existing. New software applications can be created from existing components, without being designed from scratch. The re-used components may be integrated with slight modifications or parameterizations. Some components may need wrapping code for system integration purposes. A component should be easy to deliver and implement, and easy to replace. In some cases, it may not be possible to re-use the component, but only its interface, which needs to be refined and implemented again. This type of partial re-use can also speed up the development life cycle.

- ***Expandability:*** System verification and validation are easier if they incorporate a clear structure, i.e., a system composed of clearly defined components. For component based legacy systems, future upgrades and maintenance is less troublesome.

Component based design approach can be viewed as a qualitative evolutionary jump in software engineering, comparable to the progression from low level assembly language programming to high level problem oriented languages in the 1970s, or the progression from procedural programming to *Object oriented programming* (OOP) in the 1990s. A widely accepted definition of components in software domain is given by Szyperski in [236]:

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 is used as a building block for designing a complex global system. A component can require or provide *services* to its environment via well-specified *interfaces* [77]. Examples can be a telemetry subsystems module in an artificial satellite or an engine in an automobile. 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 enables separation of 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, byte code, etc.; depending upon the component model. This notion allows to link the component to user defined or third party implementations or intellectual properties (IPs). In pure software development, a component can be viewed as a software implementation that can be executed on a physical or logical device. This view includes components present in high level languages and permits design-time composition. A component *interface* represents the properties of the component that are externally visible to other parts of the system, and are utilized in the design and development of the system. The interfaces may provide additional information related to a component's interaction with its environment or with other components; or about extra-functional properties such as throughput, execution time etc. This allows a more precise determination of system properties in the initial design phase.

2.2 Component models and infrastructure

Two basic prerequisites permit integration and execution of components. A *component model* defines the semantics that components must follow for their proper evolution [77]. A *component infrastructure* or *component framework* is the design-time and run-time infrastructure that permits interaction between the different components and manages their assembly and resources. Obviously, there is a strong correspondence between a component model and the supporting mechanisms and services of a component framework.

2.2.1 Component model

A component model determines the behavior of components within a component framework. This model states what it means for a component to implement a given interface, it also imposes constraints on components, such as defining communication protocols between interacting components etc., [77]. We have already briefly described the use of components in software engineering. There exit many component models such as COM (Component Object Model), DCOM, CORBA, CCM, EJB and .NET. Each of these component models have distinct semantics which may render them incompatible with other component models. As component models prove more and more useful for the design, development and verification of complex software systems, research is being carried out by hardware designers at a steady pace 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.

2.2.2 Component infrastructure/framework

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 [77]. It also provides the underlying mechanisms that allow component interactions and final assembly. Components can either be classified as *homogeneous* or *heterogeneous*, in the context of the framework. Examples of homogeneous components can be found in systems such as grids and cubes of computation units. In systems such at TILE64 [25], 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. These complex systems may contain ten of hundreds of components. Thus a robust component infrastructure is needed that regulates the critical interoperational aspects. 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 permits to address different aspects such as timing and power consumption [74]. The semantics related to component assembly can be selected by designers according to their system requirements. The component assembly can be either static or dynamic in nature. This feature is discussed subsequently in section 2.3.

2.2.3 Concepts related to component technology

2.2.3.1 Architecture Description Languages

The *software architecture* of a computing system can be generally defined as:

The structure or structures of the system, which comprise of software components and connectors, the externally visible properties of those components and connectors; and the relationships among them [137]

The *architecture* of a system is an initial design decision and helps in determining the global system parameters and constraints related to system functionality, maintainability, resource management, performance etc. As compared to *design*, architecture casts non-functional decisions and partition functional requirements, whereas design is a principle through which functional requirements are accomplished.

Typically, in languages such as *Architecture Description Languages* (ADLs), description of complex system architectures is carried out via compositions of hardware and software modules or objects. These components follow a component model; and the interaction between the components is managed by a component infrastructure [137]. The common concepts shared by nearly

all ADLs are *components, ports, connectors*, etc. They can also specify different types of component properties, which are principally expressed via the component interfaces. ADLs are mainly rooted in the *solution space*. Figure 2.1 shows the graphical notation of hardware and software concepts as expressed in *AADL* or *Architecture Analysis and Design Language* [5], an ADL for real-time embedded systems.

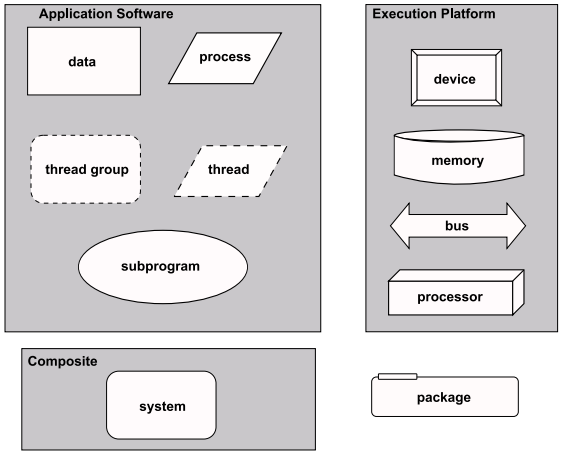


Figure 2.1: Graphical notation of system components in AADL [5]

2.2.3.2 Software vs. System components

By their very nature, software components differ from components used in embedded systems, which can be termed as *system* components as they are used to present both hardware and software perspectives of a complete system, and are addressed in this dissertation. For these types of components, several critical properties such as timing, performance and energy consumption, depend on characteristics of the underlying hardware platform. In [134], a distinction has been proposed between software and system components. Extra functional properties such as performance cannot be specified simply for just a software component in isolation. These properties must be elaborated in relation with the targeted hardware platform for its design, or paramaterized based on the properties of the underlying platform. Thus, a *system component*, can be a self contained hardware and software subsystem or module with related functional and non-functional properties.

2.3 Adaptability in component based design

For software engineering, system adaptation or reconfiguration [91] is considered to be a crucial aspect as well as a challenge. Systems must be flexible enough to evolve in order to keep up to pace with changing functional or non-functional requirements, or the environment itself. The need for adaptivity may arise at any time in a software development life cycle: during the development, implementation or maintenance phases. Adaptivity is considered to be more simplified if the software applications are component oriented in nature rather than complex monolithic blocks of codes. Adaptation or reconfiguration actions may involve changing the assembly of components, insertion of new components, deletion or replacement of components. For example, in SCORPIO [21], structural adaptation of software components is carried out. Components are restructured in order to match heterogeneous structures when integrating new components. The adaptation can be static in nature; where changes are carried out during development or component implementation times. Changes applied during execution time without halting the system are collectively termed as dynamic adaptation. Different types of adaptation techniques have been studied with regards to component based design, such as reconfigurations of component assemblies [39, 73], adapters for components [23] and *Aspect Oriented Programming* (AOP) techniques [251].

Adaptation can be either viewed as having a *safe* or *unsafe* nature [145]. Unsafe adaptation typically involves disruptive communication between components. An adaptive mechanism involving inter-communicating components may disrupt normal functional communication between the adapted component and the rest of the system, introducing system inconsistencies. It is up to the component infrastructure to ensure a safe adaptation and synchronization between the static and dynamically reconfigured components. The semantics related to component infrastructure must take into consideration several key issues when dealing with adaptation: instantiation and termination of these components, deletion in case of user requirements, etc. Likewise, communication mechanisms such as message passing and operation calls can be chosen for inter and intra communication when the components are hierarchically composed. The component framework should constantly monitor and analyze the behavior of the components to be adapted. The framework is generally based on adaptation policies and mechanisms, for deciding a correct and safe adaptation in a particular scenario.

In component based software development, system adaptation can be defined at three levels of granularity: the *global level* that deals with system level adaptation and component bindings, a *component level* related to non-functional properties of individual components; and finally the *implementation level* which allows to change the implementations related to a component:

- ***Global System/Architecture level adaptation:*** This granularity deals with adaptation of the whole system architecture, which represents how the components are binded together. The relationships between the components are also clearly presented. During a global system/architecture level adaptation, a reconfiguration and recomposition of a component assembly takes place. Thus components can be added or replaced, component hierarchy can be modified, connectors can be inserted between components for providing interactions, and so on. For example, in [228], specific adapters which are connectors are used to mediate interactions between different components. Adaptation at this level can be considered as purely structural in nature, as only assembly or composition between the components can be changed, as compared to their internal behavior.

- ***Component level adaptation:*** A component can be seen as a functional entity associated with a set of non-functional properties. In architecture level adaptation, it is possible to replace a component with another provided that their interfaces are compatible and follow similar protocols. However a mismatch can still occur, related to non-functional properties like temporal requirements, reliability, security, etc. This makes composition impossible. Management of these properties are handled by so-called *containers* or *membranes* [224] in different component models such as EJB, CCM and Fractal [39], which embody *interception* based mechanisms such as *reflection*[1] or AOP.

- ***Implementation/Program level adaptation:*** At this lowest granularity level, sequence of operations or programs are considered as entities which are encapsulated by components. Component behavior or implementation can be changed at this level, which may influence the overall behavior of the system.

Normally in component infrastructures, the components can be binded together in a non stringent manner termed as *weak-coupling* or *loose coupling*. In weak coupling, a component does not have information about its neighboring components or their functionalities at design time. The information is determined at runtime either by the component itself or another one. Weak-coupling offers flexibility in terms of adaptation but has low levels of abstraction, prohibiting complex designs. However, for complex embedded systems, the components may be heavily coupled together due to their extra non-functional properties. This may result in a rigid assembly where reconfiguration may not be easy to carry out.

A set of components that work together to provide certain services can be viewed as a system *configuration*. This configuration can be global or for an intermediate composition level. A configuration can be seen as a specific assembly of the inter-communicating components along with their associated implementations. For a system, a global configuration can be either correct or incorrect. A system can only operate correctly when it is in one of its safe correct con-

[1] Reflection is a process by which a computer program can monitor and modify its own structure and behavior.

figurations. A system moves from one configuration to another by performing configuration switching operations.

Dynamic adaptation or reconfiguration in component based design depends on the context required by designer and can be determined by different QoS criteria. Self adaptation is also present in some component models. Usually this type of adaptation is executed by an adaptation manager or *controller*, typically a separate process or component that is responsible for managing the adaptations. The controller communicates with adaptation agents attached to processes involved in the reconfiguration. An agent receives adaptive commands from the controller, performs adaptive actions, and reports the status of the local process to the controller.

One of the most popular dynamic component models is Fractal [39] as shown in Figure 2.2, it provides a dynamically adaptable structural component platform, using *aspects* as adapting tools. Components are considered as run-time entities and can be thus manipulated. Fractal allows architectural *introspection* for system monitoring and *intercession* for dynamic reconfiguration. Different types of controllers can be built within the infrastructure which help to control the encapsulated components. Fractal has been used in THINK [125] tool that defines an environment for development of dynamically reconfigurable systems. In [145], the authors propose the usage of integrity constraints for assuring consistent safe reconfigurations in the Fractal component model. The key characteristics of Fractal are its openness and light-weight nature. It thus allows designers to introduce various new extensions with minimal restrictions. Last but not least, a large number of tools supports exist for this component model.

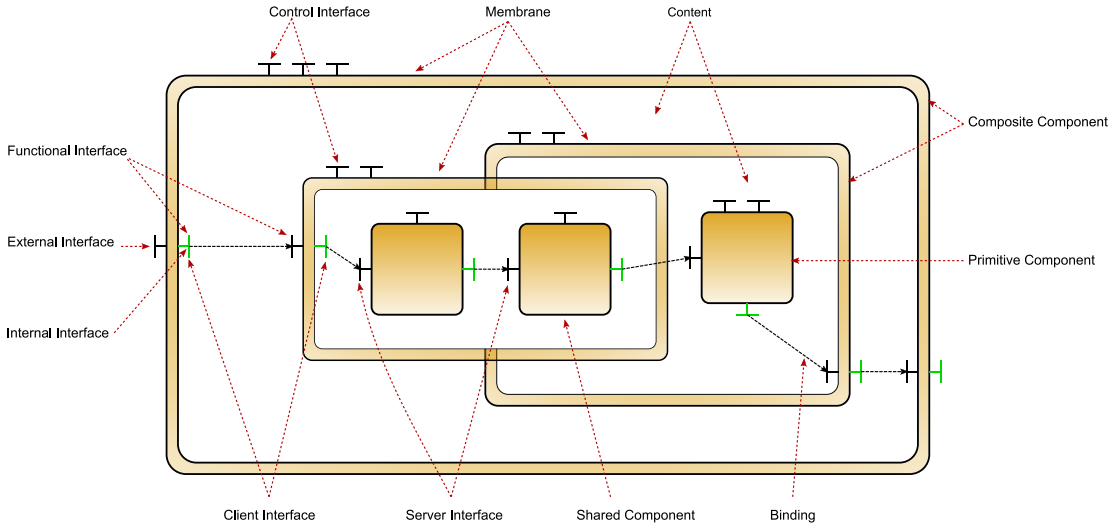


Figure 2.2: Fractal concepts

In [73], a dynamic self-adaptive *K-Component* model is introduced which enables individual components to evolve with changing environments through a complex decentralized coordinated model. The adaptation mechanism provided is aware of any modifications and can adapt the base system via *structural reflection*. Kermeta [120], proposes a model that supports verification of heterogeneous components. This approach has been used for the definition of self reconfigurable applications in order to build *Dynamic Software Product Lines* (DSPLs) [93]. Similarly SPEEDS! [232] enables the development of embedded systems and provides a uniform framework; designers can integrate heterogeneous modeling paradigms due to the existence of a robust component framework providing rich interfaces with components. However this project focuses only on static systems and is not able to handle dynamic reconfigurable components. SOFA [41] is another distributed component model, however has a limited support for dynamic reconfiguration. These limitations have been addressed in SOFA 2.0, which proposes reconfiguration patterns to avoid unsafe reconfigurations. *Microcomponents* are present in the framework, which are parallel to Fractal controllers.

In case of real-time embedded systems such as SoCs, a suitable example can be of FPGAs as presented earlier in section 1.3.3. 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 dynamic reconfiguration, a controller component concept can be integrated into the system for managing aspects of reconfigurability.

2.4 Overview of different component models and infrastructures

With the boom in component based design, a plethora of component models is emerging. Each component system has its specific characteristics and particularities reflecting its desired focus and the targeted application domain. Comparisons of component models can be found in numerous surveys such as in [42, 56]. It is not possible here to give a detailed description of the different component models. We thus provide a brief overview related to some existing examples in literature:

- ***Component Models for run-time composition***: This category deals with components models and platforms which are not primarily developed for real-time embedded systems, but in which the component implementation and composition is carried at run-time. Examples include Sun Microsystems Java Beans and Enterprise Java Beans (EJB); Component Object Model (COM), Distributed Component Object Model (DCOM) and COM+; the .NET framework; OMG's CORBA and CORBA Component Model (CCM); and Real-time CORBA which is a CORBA extension for applications with real-time requirements.

- ***Component models for embedded system design***: This category lists some of the component models and infrastructures that have been developed for applications targeting embedded systems. Typically, component implementations are given in a compilable language (such as C/C++) and are composed before compilation. Real-Time Operating Systems (RTOS) or run-time executives manage the execution semantics. Some examples are: Koala [187] (a component model developed by Philips for consumer electronic devices); Rubus Component Model [6] (an RTOS developed by Arcticus Systems AB); and PECOS [171] (a component model targeting smart cell phones, PDAs and industrial field devices). In [65], the SysWeaver component modeling tool is illustrated for embedded real-time applications, while [215] proposes a UML 2 profile for separating the functional and non-functional (domain-specific) aspects of components. Other component models include examples such as Robocop[2] (being spearheaded by Nokia, Philips and other research institutes); Procom[3][227] and the Pin component technology[4]. Fractal as defined previously, is a modular and extensible component model that has been used in the development of real-time distributed dynamic embedded systems [125]. Similarly, a SOFA HI profile based on SOFA component model has been proposed in [197], for specification of high-integrity embedded systems.

- ***Infrastructures for heterogeneous system design***: This category lists some infrastructures or platforms which are used to model systems composed largely of heterogeneous systems. The system can contain interconnected components, which can be expressed in different languages, modeling paradigms or formalisms. Examples include MetaH [113] (a domain specific ADL for avionics systems created by DARPA and United States Army), Metropolis [235] (where the components are composed at a model level); and Ptolemy II [69] (a infrastructure with special focus on heterogeneous embedded systems) with an associated software environment used for a broad range of applications including parallel processing and signal processing.

- ***Hardware/Software modeling languages***: This category provides a brief overview of some languages which are not component models, but are used to model and design real-time

[2] http://research.nokia.com/research/projects/trust4all/index.html
[3] http://www.mrtc.mdh.se/index.php?choice=publications&id=1467
[4] http://www.sei.cmu.edu/library/abstracts/reports/05tn001.cfm

embedded systems in a modular way. VHDL is an HDL for the design of integrated circuits at the Register Transfer Level. It can be used in wide range of contexts such as implementation in reconfigurable architectures, i.e., FPGAs. VHDL supports modular semantics; and abstract behavioral models can hide implementation details. A part of VHDL is synthesizable and can be implemented on target FPGAs. Designers can construct structural designs in VHDL using *entities*, *ports*, and *signals*. An entity can be instantiated within other entities as a *component*, allowing structural design hierarchies. Ports and signals permit data communication between the components; and have specific data types. Finally, concurrent behaviors are modeled via *processes*. SystemC is a system description level language at a higher abstraction level as compared to HDLs like VHDL or Verilog. It permits system modeling above the Register Transfer Level. Similar to HDLs, modular structures in SystemC can be created by the usage of *modules*, *ports*, and *signals*. One of the challenges in providing a system-level design language is that there is a broad range of design abstraction levels and *Models of Computations* (MoC).

Summary. As seen in this section, there exist a wide variety of component models, some are more generic in nature and are purely used in software engineering domain, while others can be used to model both hardware/software aspects. Some of these component models in turn have been used in the specification and development of real-time embedded systems with special focus on heterogeneity and adaptivity.

2.5 Towards embedded systems and reconfigurable SoCs

2.5.1 Present issues

Design of SoC based real-time embedded systems must consider constraints that do not apply to traditional large component and object-based systems such as data processing systems. Some of these constraints are detailed below:

- These critical systems must satisfy constraints on a diverse range of properties, such as extra-functional properties: timing (e.g., latency, missed deadlines), QoS (e.g., performance, consumed area), and dependability (including reliability, security and safety) among others.
- These real-time systems often operate with scarce resources such as low processing power, memory, communication bandwidth, etc.
- Some functional and extra-functional properties can be statically predicted, specifically if the system is deemed to be safety-critical.
- Dynamic reconfigurability: Reconfiguration and especially dynamic reconfiguration is still a complicated issue for SoC based component models. While component models like Fractal [39] and its extensions such as FracL [229] target dynamic reconfiguration, they are mostly used for distributed systems.

Therefore, definitions and hypothesis that hold true for large traditional component based systems have to be reconsidered for real-time embedded systems. As stated before, [236] defines components with contractually specified interfaces, completely explicit context dependencies, independent deployment and third-party composition. While this definition fulfills the requirement for component models used in non-critical, non-real time environments, it is not completely suitable for component models where component implementation is carried out at run-time; and specially for real-time embedded systems. Embedded systems also vary in range, for e.g. from small smart phones to huge complex systems, such as illustrated in Figure 1.7, with a wide range of requirements. A large embedded system may offer more resources and can provide better prerequisites for using the most widely used component technologies, as compared to smaller systems. Interfacing of these systems is also of utmost importance.

In current popular component technologies, the interfaces are usually created as object interfaces supporting *polymorphism* by *late binding* [77]. While this mechanism permits assembly

of components which are unaware of each other besides the connecting interface, this flexibility comes with a performance penalty that may be difficult to integrate for small embedded systems. The need for a run-time environment may arise as well, to support the component infrastructure with a set of offered services, such as management of dynamic reconfiguration.

Taking into account, all the present constraints for real-time embedded systems, there are several reasons to perform component implementation and composition at design time rather than run-time. Global optimizations can be carried out: e.g., in a static component composition or design-time, interconnections between components could be transformed into function calls instead of dynamic event notifications. Composition tools can generate a monolithic firmware for the device from the component-based design. Similarly, verification and prediction of system requirements can be done statically from component properties.

For small real-time systems, component technologies have been developed for particular classes of systems. Often, these have been done within development organizations, and their adaptation outside these organizations is limited. To avoid heavy-weight run-time platforms, they mostly do not support run-time deployment of components and lack many services. Composition of components into a (sub)system is rather performed in the design environment, prior to compilation, thus enabling static prediction of system properties and global optimizations.

Thus for embedded systems, a component model should incorporate different features: it should be light-weight in nature. This is because heavy-weight component technologies are normally complex to implement and incur large overheads. These component models must have adequate mechanisms to support reconfiguration, should incorporate aspects related to component implementations, and may have different abstraction layers to allow designers having different expertise to work at their respective layers.

2.6 Challenges related to component based design

While there has been a lot of research carried out in the domain of component based design and especially targeted to embedded systems, the approach still faces some challenges. Some of which are described below:

- ***Wide variety of approaches***: A common trend in component based design related to the real-time industry is to start using more widely adopted component technologies for embedded systems. Examples are Fractal, COM and CORBA/RT-CORBA. This allows cost savings in terms of resources, by using parts of these technologies needed by the designers. The advantage of this trend is *interoperability*, as the developed system can interoperate with other systems using the fundamental technologies. The main disadvantage of this trend is that the underlying technologies do not, a priori, support extra-functional properties essential for real-time embedded systems.

- ***Absence of a common standard***: In the previous section we have also included design tools such as MetaH and Ptolemy II, in which systems are designed by putting together pieces that might be viewed as components. The advantage of these tools is that they support a wide variety of design notations. However, these components can be assembled only in the supporting tool, meaning that all the different developments must be developed in the same environment. In this perspective, these tools have similarities to tools like SCADE[5] or UML based tools.

- ***Independent and incompatible definitions***: There are many efforts undergoing for defining component technologies for real-time embedded systems. However, examples such as Koala and PECOS component models have not spread very rapidly outside the parent organization in which they were conceived. An advantage of these models is that they can be customized for their application domain. Disadvantages are the lack of synergy across diverse range of application domains, as it is costly to develop tool support, and such development is difficult to justify for proprietary component technologies.

[5] http://www.esterel-technologies.com/products/scade-suite/

- ***Integrating aspects of reconfigurability***: Currently most component models for embedded systems do not offer concrete features to specify reconfiguration aspects related to SoCs. This issue must be tackled along with the other mentioned issues.

2.6.1 Standardization efforts and specification standards

This section provides a brief overview of some specification standards and implementation technology standards relevant to component based design. Over the last decade, there has been an increasing emphasis on the development of component based design aiming towards high abstraction levels, in order to handle the issues related with a system's complexity. The Unified Modeling Language [179] (UML) was one of the first modeling languages to standardize software engineering concepts, and it and related Object Management Group (OMG) standards such as MDA [161, 174] are currently the de-facto standards for incorporating new concepts into the software industry. UML offers a component based approach, and tends to use components as a higher-level modeling artifact that can be used irrespective of the nature of the high level models (specification design, implementation, etc.). Components are used throughout the system development till component implementation. The implementation part of a component becomes one of its aspects only relevant for the implementation stage. Details related to these aspects are provided in the next chapter.

2.7 Conclusion

It is obvious that in the context of embedded systems and specially SoC, there is a critical need for widely adapted component models and infrastructures. Also many current component technologies are rather tightly bound to; and thus make sense to, a particular platform (either runtime or design platform). Thus platform independent models must be developed. Similarly, adequate tool support should be present, as the adaptation of a component technology depends on the development of its tools support. Information related to hardware platforms must also 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.

Component verification and certification are so far unsolved problems. There seem to be no standardized procedures for ensuring component trustworthiness. There are several suggestions for how to handle functional and extra-functional properties of system design (timing, QoS, etc.). Widely accepted techniques for specifying functional and extra-functional properties of components remain to be developed. Managing reconfigurability and adaptation in a component infrastructure is also a complicated issue. In order to meet the resource limitations of embedded systems, there is a desire to be able to adapt components to use exactly the resources and services that are needed in a particular service.

Finally, the MDA approach is of particular interest to the real-time community for resolving the problems arising in component based design. The following chapter details the usage of components in MDA and the gained benefits.

Chapter 3

Model-Driven Engineering and MARTE

In chapter 1, we have illustrated the advances in SoC design, particularly with respect to the hardware aspects, with special focus on reconfigurability. On one hand, due to technological evolution, computing capability of a processing unit increases very rapidly; on the other hand, parallel architectures play more important role, with regards to criteria such as system performance. This leads to the gap between software development and the hardware computational capacity, as the former does not benefit from the same advancement rhythm as the latter. Similarly, the level of complexity of these systems is increasing continuously to new heights as integrated heterogeneous components become more and more common in these systems. Classical programming languages are becoming increasingly difficult to be adapted in these complicated system designs. Moreover, integrating reconfigurability features add new complexity layers to these already convoluted systems. However, new design and development methodologies for current system designs are emerging continuously. In chapter 2, traditional component based design practices have been briefly explored to resolve the above mentioned issues plaguing real-time embedded systems and specially SoC Co-Design. While interesting in nature, due to a wide variety of approaches, incompatible methodologies and lack of common standards; alone they are not sufficient enough to tackle SoC complexity.

Among the diverse range of intensive research activities that are dedicated to address fast and efficient software design issues, MDE [157, 222] stands out as one of the most promising approaches related to the design and development of real-time embedded systems. In section 3.1, MDE is briefly discussed, with emphasis on its principles. Afterwards, usage of MDE in the field of embedded systems is discussed with special emphasis on the MARTE profile. Subsequently, a brief comparison of MARTE with other industry standards and UML profiles is given, followed by exploration of reconfigurable concepts specification in MARTE.

3.1 Model-Driven Engineering

According to Wikipedia[1]: *"MDE pertains to software development, which refers to a range of development approaches that are based on the use of software modeling as a primary form of expression."*

As development in MDE and associated tools and technologies is increasing, it has become a promising approach not only for software engineering but for hardware as well as system engineering, attracting much attention in industry and academia. Above all, MDE plays a very important role, which contributes to modeling, automatic code generation and bridging between different technologies.

The key integral concept in MDE is a *model*. Two core relations are related to a model in MDE. The first is *representation* (a model is a representation of a system); and secondly *conformance* (a model conforms to a metamodel) [32]. These two relations are separately presented in section 3.1.1 and section 3.1.2. Another key notion of MDE, *model transformation* is also discussed in section 3.1.3. The advantages of MDE are illustrated successively following the introduction of the previous concepts. MDE in practice is also discussed in this chapter.

3.1.1 Models

A model signifies a representation of some reality or system with an accepted level of abstraction, i.e., all unnecessary details of the system are omitted for the sake of simplicity, formality, comprehensibility, etc. A model has two key elements: concepts and relations. Concepts represent *things* and relations are the *links* between these things in reality. A model can be observed from different abstract point of views (*views* in MDE). The abstraction mechanism avoids dealing with details and eases re-usability.

However, the model concept is not a novelty. According to Favre [89], the notion of models dates back to ancient times, approximately more than five thousand years ago. The alphabet of *Ugaritic cuneiform* (3400 B.C.) already introduced a similar notion by defining a set of abstract representations (characters) and rules (pronunciations) that permitted the expression of some reality (sentences). Most recently, in information technology domain, programming languages, relational data bases, semantic web, etc., are all based on the same fundamental principle, where a set of predefined and linked (or concatenated) concepts represent some reality once they are given certain interpretations.

Intuitively, according to different granularity degrees of detail, there may exist several levels of abstraction. But these accepted levels of abstractions may not be unique, which is determined according to specific system requirements. However the choice of a good abstraction level does not imply a simple and non trivial task. First, the evolution of abstraction levels in software and hardware design is briefly discussed, which partly explains the notion of models in MDE.

3.1.1.1 Raising abstraction levels in software design

The history of software evolution can be viewed as a history of raising design abstraction levels. Since the very beginning, machine code (first generation languages) helped people to escape from direct manipulations of physical elements in a machine. However it was a tedious task, because programming with large set of binary numbers: "1"s and "0"s; implied a dull and daunting procedure.

The later assembly languages (second generation languages) somewhat helped in combating this tediousness by substituting the binary numbers by some literal instructions, but it still remained a challenge for programmers because they needed to know precise hardware instructions for correct program execution. This turned out to be a huge obstacle for designers who did not have any experience or knowledge about the hardware aspects.

Efforts were undertaken so that programming languages become independent from specific machines and platforms. High-level languages (third generation languages), such as FORTRAN, LISP and C, have made this goal more possible. Developers can put their focus on functionalities, which appear more interesting to them. Some of these languages, e.g., APL (A

[1] http://en.wikipedia.org/wiki/Model-driven_engineering

Programming Language), C, PROLOG, and ML (ML stands for metalanguage) are major language paradigms still in use in industry nowadays.

However, the increasing software complexity has resulted in the software crisis, which involves factors such as development time, cost, etc. *Object-Oriented Analysis and Design* (OOAD) has been developed to partly address this problem. The basic principle of OOAD is that a system is composed of a set of interacting objects; which are independent from one another in the sense that their local states are private; and can be only accessed by some provided operations. This object independence makes their re-usability possible. OOAD also involves the definitions of object, class and their relations. These classes are independent from implementation concerns. They share several features, such as *modularity*, *encapsulation*, *inheritance* and *polymorphism*. At run time, objects are created dynamically according to their class definitions.

These notions, such as *class* and *object*, imply a clear separation between *specification* and *implementation*. However, the design of *classes* is still restrained by object-oriented languages and the virtual machines on which their implementing *objects* execute. OOAD helps to address the software complexity problem to some extent. Based on the practices of object-oriented development, recent software research focuses on software/hardware modeling, domain-specific modeling, heterogeneous system integration, high-level abstraction, etc., where OOAD is not well adapted.

With respect to *Component Based Design* or *Component Based Software Engineering*, an object in OOAD can be represented as a component subsystem. A component can be defined as a package of objects and is specified using type or class definitions. These definitions collaborate in offering a set of services, grouped into one or more interfaces. Class definitions can be *wrapped* in order to conform to the component technology of choice. Analogously, at the OOAD level, objects can be wrapped to conform to the notion of components. It has been argued that this strategy offers advantages as compared to traditional object-orientation. Basically, it forces good design practices upon developers, i.e. strict visibility rules, encapsulation and explicit dependencies [136, 255].

3.1.1.2 Raising abstraction levels in hardware design

Since the advent of *Integrated circuits* (ICs) in the 1950s, the evolution in hardware technology has directly elevated the design abstraction levels. Initial electronic circuits were crafted by hand and the schematics were drawn on paper. Not surprisingly, this handcrafted way of designing was prone to numerous errors and was extremely time-consuming. In response, *Computer Aided Engineering* (CAE) was introduced in the late 1960s - early 1970s; and rudimentary logic simulators were introduced. Proprietary text based gate-level netlists were utilized and *Test Vectors* (test benches) were applied on inputs as stimuli for getting the simulation results.

However, as the IC evolution increased the number of integrated gates from mere ten or hundreds to thousands, the simulation time shot up exponentially. Thus long hours were required for simulation and verification purposes. Layout editors by companies such as Calma in the early 1970s offered digitized schematic designs. These digital files were subsequently used in the creation of photo masks; which were then used in the fabrication of actual silicon chips. These editors can be considered as ancestors of modern *Computer-Aided Design* (CAD) tools.

In 1980, the publication of *Introduction to VLSI Systems* [158] advocated chip design with programming languages that compiled to silicon. This resulted in an immediate hundredfold increase in complexity of chips that could be designed, with optimized access to design verification tools using logic simulation. The chips were easier to lay out (placement and routing), but were more accurate as well, as their designs were thoroughly simulated before fabrication. 1981 marks the beginning of *Electronic Design Automation* or EDA as an industry, in order to bring together all the CAE and CAD design tools for increased synergy and compatibility.

In 1986, Verilog was first introduced as an hardware description language by Gateway; followed by VHDL in 1987 by the US. Department of Defense. Following these introductions, HDL simulators were quickly developed, permitting direct simulation of chip designs: executable specifications. Within few more years, back-ends were developed to perform logic synthesis.

Electronic System Level (ESL) design and verification is an emerging high abstraction level electronic design methodology. According to [19], it is defined as: *"the utilization of appropri-*

ate abstractions in order to increase comprehension about a system, and to enhance the probability of a successful implementation of functionality in a cost-effective manner." At present, ESL is an established approach at most of the world's leading SoC design companies. The basic objective is to model the behavior of the system using a high-level language such as MATLAB, C/C++. System level languages such as SystemC permit abstract system modeling at a higher level of abstraction. Similarly, using EDA tools and processes such as *High Level Synthesis* (HLS) and embedded software tools, rapid and correct-by-construction implementation of the system can be automated, although much of it is performed manually today. From its conception as an algorithm modeling methodology with *no links to implementation*, ESL has rapidly evolved into a set of corresponding methodologies which enable embedded system design, verification and validation, testing and debugging through to the hardware and software implementation of system-on-board, custom SoC, system-on-FPGA and entire multi-board systems.

For SoC conception, currently following HLS approaches are utilized: the behavioral description of the system is refined into an accurate RTL design for SoC implementation. An effective HLS flow must be adaptable to cope with the rapid hardware/software evolution and maintainable by the tool designers. The underlying low level implementation details are hidden from users and their automatic generation reduces time to market and fabrication costs, as compared to hand written HDL based implementations. However in reality, the abstraction level of the user-side tools is usually not elevated enough to be totally independent from low level implementations. Each particular implementation of the system (application/architecture) requires a particular specification which is usually in SystemC or a similar language resulting in several disadvantages. Immediate recognition of system information such as related to hierarchy, data parallelism and dependencies is not possible; differentiation between different concepts is a daunting task in a textual description and makes modifications complex and time consuming, resulting in increased time-to-market.

3.1.1.3 Machine-recognizable models

Model based approaches play a significant role in software evolution, particularly related to system analysis and design. Several modeling approaches that should be cited include: Merise [243] (1970), Structured Systems Analysis and Design Methodology (SSADM) [85] (1980) and Unified Modeling Language (UML) [179] (1995). Similarly hardware evolution is also beginning to benefit from model based design approaches. These approaches aid in the comprehension of the current MDE model concept. Each of these approaches proposes certain concepts, semantics and notations to describe the system to be designed. The common point in all these approaches is that in each stage of the system life cycle, a set of documents composed of some diagrams are created; that allow designers, developers, users, etc., to share their system designs.

These approaches have some crucial advantages. The abstraction achieved by a modeling approach permits to abstractly emphasize the overall system structure, while bypassing details related to implementation and associated technologies. This enables conception and development of huge complex systems in a speedy and efficient manner. These approaches enable representing a system or a part of the system with different point of views, which permit system separation by aspects related to specific domain views. These approaches can also be found in aspect oriented programming (AOP) and *Domain-Specific Languages* (DSLs) [160].

However, these approaches have been criticized [89] for the heaviness and lack of flexibility in rapid system design and development. The resulting models of these approaches, called *contemplative* models by Favre et al., are essentially only used for communication and apprehension. They remains passive with regards to production, although ironically the first concern of information technology is to produce the artifacts interpretable by machines [89]. Hence, in order to be productive so as to accelerate system design and implementation, machine-recognizable models, which are not only human-recognizable, become indispensable and critical.

3.1.1.4 Models and MDE

MDE [157, 222] has emerged to mainly satisfy the requirements of two communities, i.e., programming language community (particularly OOAD community); and the system analysis and modeling community. *Models*, which are the key concept in MDE, are utilized systematically

throughout the whole system life cycle. Unlike classes and objects found in OOAD, models in MDE are flexible, as they are not required to take implementation aspects into account. The modeling method proposed by MDE also makes up for the deficiency of traditional modeling approaches through the proposition of machine-recognizable models. MDE provides a development framework, where models transit from a contemplative state to a productive state. Thus, models become the first class elements in a software development process, that aims to improve its portability and maintainability through separation of concepts, particularly separation of concepts of specific domains or technologies in order to boost productivity and quality. MDE enables re-utilization of these models by the associated tools and design patterns and helps to keep the models human readable.

Models: transition from solution space to problem space As compared to traditional programming languages that focus on *solution space*, MDE aims to concentrate on the overall structural and behavioral modeling of a system; without being influenced due to some specific domains of computing technologies. This analysis and modeling is effectively carried out by using the concepts of application domains that form the problem space, i.e., analysis and modeling of the problem itself, not a solution to the problem. Several distinct advantages arise from this transition such as the capability of representing large-scale complex systems and capacity of handling heterogeneous systems. First, a designer can focus on the given problem without knowing too many specific computing technologies. As a result, the accessible system complexity can be increased gradually.

Secondly, MDE enables system level (software and hardware) modeling at a high specification level permitting several abstraction layers. Thus a system can be viewed globally or from a specific point of view of the system, allowing to separate the system model into parts according to relations between system concepts defined at different layers. This *Separation of Views* (SoV) gives a designer the opportunity to focus on a domain aspect related to an abstraction layer. Also, when seen from a hardware design perspective, MDE can be viewed as a *High Level Design Flow* containing several *Internal Representations* (IR), which bridge the gap between high and low abstraction layers [102].

Thirdly, a model can be a composition of several other models, with each model adapted to a specific formalism appropriate to a particular domain. The composition is possible because all the models can be expressed in a common uniform language. Thus, different development teams from different application domains can co-operate on the same heterogeneous system with their appropriate domain concepts, where implementations of different computing technologies are not involved. Using a graphical modeling language i.e. UML (Unified Modeling Language) for system description also increases the system comprehensibility. This allows designers to provide high-level descriptions of the system that easily illustrate the internal concepts (task/data parallelism, data dependencies and hierarchy). These specifications can be reused, modified or extended due to their graphical nature.

3.1.2 Metamodel and metamodeling

In order to be interpretable by a machine, the expression, with which a model is represented is pre-defined formally. This is achieved by a *metamodel*. In MDE, A metamodel is a collection of concepts and relations for describing a model using a model description language; and is used for defining the *syntax* of a model. A metamodel can be viewed as an internal representation in a high-level synthesis flow.

Each model that is designed according to a given metamodel is said to *conform* to its metamodel at a higher level. This relation is analogous to a text and its language grammar. Here level does not signifies an abstraction level, but a definition level. A metamodel itself is also a model, thus it also conforms to another metamodel. However, in order to define a model, it is not convenient to define an infinite succession of metamodels, with each one conforming to an other at a higher level. One formal solution to this issue is the definition of a metamodel, which conforms to itself, i.e., it can be expressed only by using the concepts it defines. Currently, widely used metamodels, such as Ecore [80] and MOF [175], are examples of such kind of metamodels or *metametamodels*.

Figure 3.1 represents the relation between models and metamodels. One of the best known metamodels is the UML metamodel. The **M0** level is the representation of some reality (a computer program). In this example, several variables (*Number* and *Balance*) take values that are assigned to them. The **M1** level is the lowest level of abstraction, where the concepts can be manipulated by developers. In this example, declarations are found for the variables used at the **M0** level and the notion of *Account*, which contains these variables. The model at the **M1** level conforms to the metamodel at the level of **M2**. The concepts manipulated by developers at **M1** are defined and situated at this level. *Account* is a *Class*, whereas variable declarations are *Attributes* enclosed in the *Class*. Finally, a metamodel at the **M2** level conforms to a metametamodel (at the level of **M3**). The latter conforms to itself. In the example, the concepts, such as *Class* and *Attribute*, are metaclasses, whereas the containing relation is a metarelation. The metametamodel can describe itself, e.g., *metaclass* and *metarelation* are still metaclasses; and relations such as source and destination are metarelations.

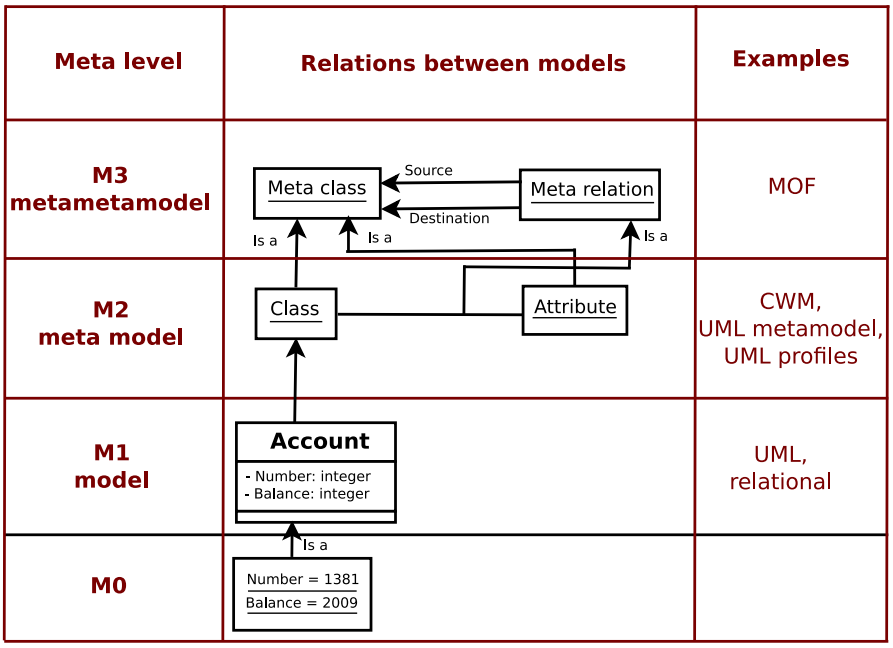


Figure 3.1: Different levels of modeling in MDE

If the highest-level metamodel has been defined so as to conform to itself in a formal way, and the syntax and semantics of this metamodel are described explicitly, then the models that conform to this metamodel can be interpreted by a computer. Once significations of the concepts in this metamodel are programmed, a computer will be capable to read any model that conforms to this recursive metamodel directly or indirectly. However, a metamodel is only composed of structural information in relation to its models, no semantics are involved formally. A model makes sense with the help of its interpretation, either by users through a provided specification, which includes the concepts of the metamodel, or by a machine during the transformation of the model.

3.1.3 Model transformations

Models in MDE are not only used for communication and comprehension but using model transformations [219], produce concrete results such as executable source code. With the help of metamodel(s), to which these models conform to, models can be recognized by machines. As a result, they can be processed, i.e., a model is taken as input (source) and then some models (target) are generated. This process is called a *model transformation*, as shown in Figure 3.2; it is a compilation process that transforms a source model into a target model and allows to move from an abstract model to a more detailed model. The condition for a successful model

transformation is that both source and target models must conform to their explicitly specified respective metamodels.

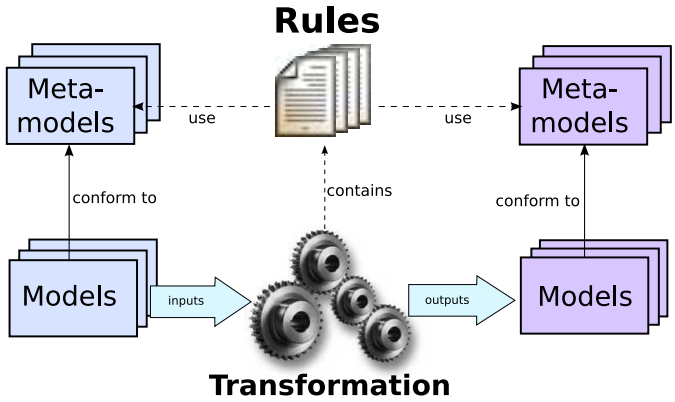


Figure 3.2: A model transformation allows to transform source models into target models via a set of rules. These rules are defined by using the concepts of metamodels, to which the source/-target models conform to

3.1.3.1 Classification of model transformations

MDE model transformations can be classified according to different point of views. Several proposed classifications are briefly presented below. According to the homogeneity and heterogeneity of the models on the two sides (source and target sides) of a transformation; two kinds of transformations: namely *exogenous* and *endogenous* can be distinguished [159]. An endogenous transformation only considers a single metamodel, i.e., the same metamodel is used for the source model and the target model. An exogenous transformation uses different source and target metamodels. According to the abstraction level of source and target models, a transformation can be a *vertical* one, when two levels are different, or a *horizontal* one when the models are situated at the same abstraction level.

In addition to the unidirectional transformation, whose direction is implied by a *source* and *target*, a transformation can also be bidirectional in nature. In the unidirectional transformation case, only source mode can be modified by designers; and the target model is re-generated accordingly. However, in case of a bidirectional transformation, the target model is also modifiable, requiring the source model to be modified in a synchronized way. Consequently, bidirectional transformations always lead to model synchronization issues. [233] presents a survey on bidirectional transformations. The above mentioned transformations can be termed as *model-to-model* transformations, as compared to the *model-to-text* transformations, which convert models to executable code for eventual implementation.

3.1.3.2 Transformation rules

Model transformations are always implemented by an engine that executes the transformations based on a set of *rules*. The rules can be either *declarative*: where outputs are obtained from some given inputs; or *imperative* (how to transform). Declarative rules, in general, are expressed in three parts: two *patterns* and a *rule body*. The two patterns are the source and target patterns respectively in a unidirectional transformation or the same pattern acting as source/target in a bidirectional transformation. A source pattern is composed of some necessary information about part of the source metamodel, according to which a segment of source model can be transformed. Correspondingly, a target pattern consists of some necessary information about part of the target metamodel, according to which a segment of target model can be generated. The link between these two patterns is the rule body (or a logical part according to [60]), which defines the relation between the source pattern and the target pattern.

Declarative rules can be composed in a sequential or hierarchical manner. Thus, flexibility and re-usability in transformations can be achieved. In a sequential case, all the rules can be

executed one by one, hence, all the source patterns of these rules cover the source model and all the target patterns cover the target model. In the hierarchical case, the root rule can have *sub-rules*. The corresponding source/target patterns directly cover the whole source/target models. Transformation are in general, mixed-style in nature (having both declarative and imperative rules) so that complex transformations can be implemented.

3.1.3.3 A multi-level approach in modeling and transformation

Between the different abstraction levels of a modeled system and its resulting code, intermediate levels can be created. At each intermediate level, a model and its corresponding metamodel are defined, hence a complete model transformation turns into a compilation chain, consisting of successive transformations. These intermediate models do not increase the overall workload. On the contrary, they are added when it is difficult to bridge the gap between two models directly. At each intermediate level, implementation details are added to the model transformations. Usually, the initial high level models contain only domain specific concepts, while technological concepts are introduced seamlessly in the intermediate levels.

A typical example is the *Platform-Specific Model* (PSM) defined in *Model-Driven Architecture* (MDA) that is situated between *Platform-Independent Model* (PIM) and the resulting code. This multi-level approach contributes in reducing the complexity of transformations. For instance, the information needed to transform a high-level model to a low-level one is divided into several portions, each of which is included in a transformation.

New rules in a model transformation extend the compilation process and each rule can be independently modified; this separation helps to maintain the compilation process and facilitates in making the transformations modular. The advantage of this approach is that it enables to define several model transformations from the same abstraction level but targeted to different lower levels, offering opportunities to target different technology platforms. Another advantage is that the development of a chain of transformations can be concurrent, once intermediate models are defined.

3.1.3.4 Traceability

In some cases, model transformation information is expected to be logged for reasons related to verification and debugging. For instance, relations between the elements of source and target model and modifications or debug information of the target model, are needed to be logged in order to backtrack and find the corresponding elements in the source model. Traceability in the model transformations consists of finding the transformation relation between the elements in source/target pattern. For instance, a *trace* can be observed and saved in the execution of a transformation [9], which enables the traceability. However, traceability is still not well supported in current transformation tools.

3.1.3.5 Productivity issue

The modeling approach proposed in MDE and its corresponding model transformation helps to address the productivity issue. As mentioned in section 3.1.1, high-level modeling reduces the complexity of system design, hence it contributes in improving productivity. Moreover, one of the distinct features of MDE over other modeling approaches is: models can be directly used to generate implementation-level results (e.g., executable source code) from high-level models. This production is achieved by the model transformations.

3.1.3.6 Transformation tools

Currently, the only standard related to model query and transformations is the *Meta-Object Facility Query/View/Transformation* (MOF QVT) [176], proposed by Object Management Group (OMG). However, there exists a large number of transformation languages and tools such as the ATLAS Transformation Language (ATL) [119], Kermeta [120] and ModelMorf [244]. The drawback of these tools is that they are very specific in nature and thus not suitable to be adapted as

a standard. ATL is a model transformation language (a mixed style of declarative and imperative constructions) designed according to QVT. Kermeta is a metaprogramming environment based on an object-oriented DSL. Also, since the standardization of QVT, none of the investigated tools are powerful enough to execute large complex transformations. Similarly, none of the above mentioned engines is fully compliant with the QVT standard. However, new tools such as SmartQVT [246] and QVTO [184], while also partially compliant to the QVT specification, are more effective then the above mentioned tools. They are also being evolved to be fully compliant with the QVT standard.

An alternative solution to QVT is the Eclipse Modeling Framework Technology (EMFT) project[2], which was initiated to develop new technologies that extend or complement the Eclipse Modeling Framework (EMF)[3], for model creation and modification. Its query component offers capabilities to specify and execute queries against EMF model elements and their contents. EMF Java Emitter Templates (JET) [82] is a generic template engine for code generation purposes. The code generation largely consists of gathering the information contained in a model, and injecting it into different text files after an analysis. The JET templates are specified by using a JSP (JavaServer Pages) like syntax and are used to generate Java implementation classes. Finally these classes can be invoked to generate user customized source code, such as Structured Query Language (SQL), eXtensible Markup Language (XML), Java source code or any other user specified syntax. Similarly, Acceleo[4] is a promising tool capable for automatic code generation from models. Its syntax is similar to the syntax proposed in the upcoming MOF model to text standard (Mof2Text) proposed by OMG. Other code generation tools exist as well, such as Xpand[5] from OpenArchitectureWare.

3.1.4 MDE in practice

MDE is still not completely well-rounded, and there still exist propositions and initiatives. In literature, we can find them under many different names, such as *Model-driven Architecture* (MDA), *Model-Driven Development* (MDD), *Model Integrated Computing* (MIC) and *Model-Driven Software development* (MDSD). We insist on the essence of all proposals that form a foundation of principles and concepts, rather than the subtle nuances implied by these different names.

3.1.4.1 MDA

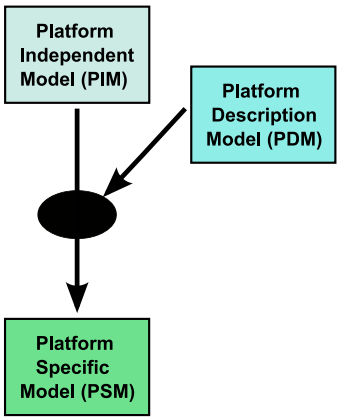


Figure 3.3: A global overview of the MDA approach

One of the best known MDE initiatives is MDA [161, 174], which is proposed by OMG [173]. In MDA, three types of models are distinguished according to the abstraction levels as shown in Figure 3.3: *Platform Independent Model* (PIM), *Platform Description Model* (PDM) and *Platform*

[2] http://www.eclipse.org/emft
[3] http://www.eclipse.org/emf
[4] http://www.acceleo.org/pages/home/en
[5] http://www.eclipse.org/modeling/m2t/?project=xpand

Specification Model (PSM). The first model generally expresses the structure of the application independent of the platform, while the PDM is the model description of the targeted execution platform. Finally the PSM is the model of the application specific to a platform or a particular technology. A PSM can be an executable model itself, or be used to generate certain source code. Transformation specifications are also proposed by OMG to bridge these types of models, such as MOF QVT [176].

The goal of MDA is to drive the system development through platform independent models which can be semi-automatically translated into any platform specific language for which a standard mapping has been defined. Thus, platform independence is obtained along with greater flexibility, with regards to final implementation.

3.1.4.2 UML

UML is considered as one of the main unified visual modeling languages in MDE. The UML metamodel [179] was standardized in 1997 by OMG. Since its standardization, UML has been widely accepted and adopted in industry and academia. UML now provides support for a wide variety of modeling domains, including real-time system modeling. It has been proposed to answer the requirements of modeling specification, communication, documentation, etc. It integrates the advantages of component re-use, unified modeling of heterogeneous system, different facet modeling of a system, etc. The proposition of UML is based on several languages, such as OMT, Booch and OOSE, which had a great influence on the object-based modeling approach, as shown in Figure 3.4. Consequently, UML is very similar to object-based languages. As UML is widely utilized in industry and academia for modeling purposes, a large number of tools[6] have been developed for its support.

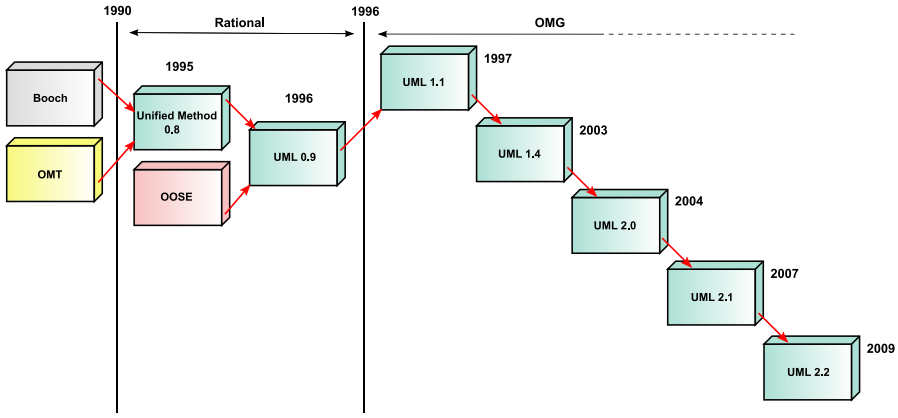


Figure 3.4: History of UML through the years

UML distinguishes between structural and behavioral modeling. The fist one concentrates on the static structure of a system, which involves constructs, such as *class*, *component*, *packages* and *deployment*. The second one focuses on behavioral aspects of the system, which can be expressed by *activities*, *interactions*, *state machines*, *sequence diagrams*, etc.

Unfortunately, the success of UML has its drawbacks, resulting in a bloated and complex language. Its expressivity and precision are not always well defined in certain cases for the specification of some specific systems.

There are also discussions on the semantics of UML. The specification of the language (a metamodel of its abstract syntax with weakly defined semantics) has also become difficult to manage and hard to understand due to its size and complexity. Some also believe that its semantics are not well defined. In particular, the semantics of UML behavioral modeling brings certain ambiguities [90]. This problem cannot be addressed by *Object Constraint Language* (OCL) [7], that is dedicated to the specification of static syntactic constraints on UML constructs. From

[6] http://en.wikipedia.org/wiki/List_of_UML_tools
[7] http://www.omg.org/technology/documents/formal/ocl.htm

this point of view, the validation of UML applications cannot be achieved in a precise manner. Related works [100, 225] have been carried out to give a clear and formal semantics to UML.

UML components

A UML component is a self-contained, modular and reusable entity that is considered as an autonomous unit in a system or a subsystem. It is also replaceable in its environment at design time and run-time if its fungible component has compatible interfaces. A component is provided only with its specified interfaces or ports; and the functionality that it provides. Its implementation is concealed, and its behavior is generally defined in terms of its interfaces.

As a subtype of *class*, a UML component has an external view (or *black-box* view) through its publicly visible properties and operations. Moreover, a *behavior*, such as state machines, can be associated with the component in order to express a more precise external view. A component also has an internal view (*white-box* view) via its private properties and realizing classifiers. This view shows how the external behavior is realized internally [178]. The UML component model is very close to the component model described in chapter 2, thus permitting to very easily use UML as modeling language or ADL for supporting a component based design methodology.

3.2 Profiles for real-time and embedded system design

3.2.1 Profiles

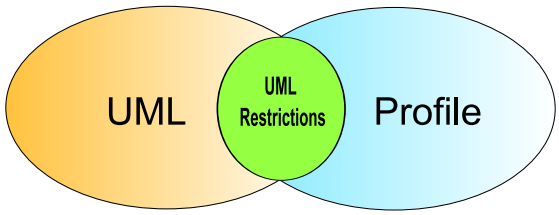


Figure 3.5: UML specialization by the profile mechanism

When UML is utilized in the framework of MDE, it is usually carried out through by its internal metamodeling mechanism termed as a *profile*. A profile is a collection of extensions and eventually restrictions. A profile is created when the UML metamodel is not sufficient enough to model the concepts related to a specific domain. Creation of a profile permits to use UML as a DSL. An extension at the profile level is called a *stereotype*. It specializes one or several UML classes and can contain supplementary attributes known as *tagged values*.

With a profile, a designer has an increasing level of design freedom at his fingertips, as he can model his targeted domain concepts via existing or user customized UML concepts. Additionally, crucial or lacking features related to a domain can be added to the profile. Also, UML profiles benefit from an extremely large presence of graphical UML modeling tools for manipulating UML models. This is also one of the reasons that the metamodels related to a model are also graphical in nature. This allows the designers to work with available tools and carry out their proper extensions. With aid of model transformations, it is possible to pass from an UML model conforming to a UML profile to another intermediate model conforming to its respective metamodel (on which the model transformations are based).

3.2.2 Profiles for Real-Time Embedded Systems (RTES)

Several profiles for the design and development of real-time embedded systems have been developed and many of them have even been standardized. We separate these profiles into two major categories. The first category regroups some of the profiles that are oriented towards low level implementations and are interested in electronic circuits and the realization of these components. These goal of the profiles is to generate the code from UML diagrams and to use

UML as an HDL or system design language. UML for SoC [183] and UML for SystemC [211] are among these profiles as shown in Figure 3.6. Each of these profiles has modeling concepts corresponding to SystemC notions in order to guarantee automatic SystemC code generation, or at least a skeleton of the code. However, the drawback of these profiles is that they are not abstract enough and very close to the low level implementation details.

The second category regroups some of the profiles that model the systems with a functional point of view. Parallel to the implementation, it is necessary to specify the functional models which are abstract and comprehensible enough; before communicating the design intent, the development of the software, carrying out different analyses and proceeding to the allocation. For example, a scheduling analysis requires a view which sees the system as a collection of processors, memories etc. Several UML profiles provide such a description, an example being of the UML profile for Schedulability, Performance and Time (SPT) [182]. All these profiles permit to annotate the functionality of a component depending upon its nature (computation, memory, communication resource, etc.). However, irrespective of their abstraction levels, these profiles are inadequate for the precise analysis or simulation of the system. For example, in order to calculate the *Worst Case Execution Time* (WCET), we have to take in account the micro-architecture details related to a processor. However this aspect has not been addressed in any of these profiles.

UML Profiles	*Principal Concepts*
UML for SoC	SoCModule, SoCChannel, SoCConnector, SoCPort ...
UML for SystemC	sc_module, sc_chanel, sc_port ...

Figure 3.6: UML profiles for RTES: strongly attached to low level implementation details

UML Profiles	*Principal Concepts*
SPT	Device, Processor, CommunicationResource ...
ACOTRIS	ECU, ECUGate, Channel ...

Figure 3.7: UML profiles for RTES: system modeling with functional aspects

Also as seen from these different profiles, there are currently too many specific approaches, languages and tools. They can be sometimes redundant in nature, but generally are complementary to each other. Another disadvantage is that few of these approaches have interoperability capabilities. Hence for a designer, it is mandatory to be an expert in all these approaches, creating the same problem as in traditional UML based object oriented approaches. Thus a unified modeling standard for RTES is required that can offer useful advantages. A designer can focus more on a single language than several dedicated languages. Tool interoperability and tool choices should also be readily available. This in turn, ensures that the designer requires an expertise in a single unified language as compared to several dedicated ones. Afterwards, he can choose the tool best suited for his needs. This while increases the competition between the tool vendors to create the best tools for supporting the UML RTES standard, also gives the end users better tools and choices. From a business point of view, RTES engineers will require less training efforts. We now briefly summarize two of the most popular UML profiles currently being used for the design, development and analysis of real-time embedded systems.

3.2.2.1 SysML

System Modeling Language (SysML) [177] is the first UML standard for system engineering proposed by OMG that aims at describing complex systems. SysML allows describing of the traceability requirements, and provides means to express the behavior and composition of the system blocks. This profile also provides the designer with parametric formalisms which are

used to express analytical models based on equations. The key contributions of the SysML standard are as follows:

- ***Architecture organization:*** The modeling concepts related to expressing architectural aspects. The concepts of *View*, *Viewpoint* and *Rationale* are most significant.
- ***Blocks and Flows:*** Blocks allow to represent complex systems in a composed manner. Flows in SysML enable modeling of data/control flow as well as physical flows, such as electrical flows.
- ***Behavior:*** SysML refines the UML common behavior concepts (such as state machines, activities among others) for modeling continuous systems.
- ***Requirements:*** System requirements can also be modeled via SysML. These requirements can be presented either in graphical or tabular form and help with model traceability.
- ***Parametrics:*** SysML allows designers to describe analytical relations and constraints in a graphical manner.

However, while SysML is used in the RTES community for SoC design, it was not mainly created for modeling of embedded system designs. Non-functional properties such as timing constraints, latency and throughput that are crucial for the design of RTES are absent in this profile. This is not the case of the UML profile for Modeling and Analysis of Real-Time and embedded Systems [181] (MARTE).

3.2.3 MARTE

MARTE [181] (Modeling and Analysis of Real-Time and Embedded Systems) is an upcoming industry standard UML profile of OMG, dedicated to model-driven development of embedded systems. MARTE extends UML along with added extensions (for e.g. performance and scheduling analysis).

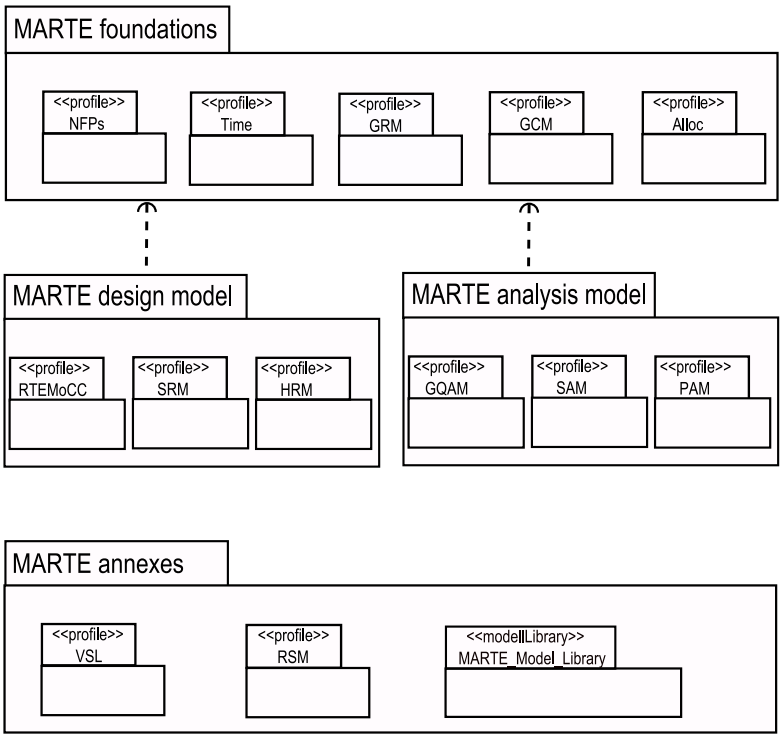


Figure 3.8: Global architecture of the MARTE profile

The MARTE profile extends the possibilities to model the features of software and hardware parts of a real-time embedded system and their relations. It also offers added extensions, for example to carry out performance and scheduling analysis, while taking into consideration the platform services (such as the services offered by an OS). Figure 3.8 presents the global architecture of the MARTE profile and its decomposition in packages. The profile is structured in two directions: first, the modeling of concepts of real-time and embedded systems and secondly, the annotation of the models for supporting analyses of the system properties. The organization of the profile reflects this structure, by its separation into two packages, the *MARTE design model* and the *MARTE analysis model* respectively. These two major parts share common concepts, grouped in the *MARTE foundations* package: for expressing non-functional properties (*NFPs*), timing notions (*Time*), resource modeling (*GRM*), components (*GCM*) and allocation concepts (*Alloc*). An additional package contains the annexes defined in the MARTE profile along with predefined libraries.

3.2.3.1 Overview of the MARTE specifications

We briefly give an overall summary of the MARTE concepts present in both the MARTE metamodel and the MARTE profile.

- ***Core Elements:*** Describes the basic core concepts in the MARTE specifications, such as *Models*, *Model Elements* (such as *Classifiers*) and their associated *behaviors*. The behaviors can be any thing such as state machines, activity diagrams, interactions (sequence diagrams/timing diagrams), automatas, etc.

- ***Non Functional Properties (NFPs):*** They allow to describe properties which are not related to functional aspects such as energy consumption, memory utilization, consumed resources, etc. As specified in chapter 2, this is a critical aspect for a detail description of an RTES.

- ***Time Modeling:*** The MARTE Time package allows concepts mainly used in synchronous domain as well as in discrete real-time systems such as FPGAs. The most important notion in that package is the usage of time constraints on UML behavioral models such as sequence diagrams and state machines.

- ***Generic Resource Modeling (GRM):*** This package introduces the concept of *Resource* and *Resource Services*. Resources can be classified into types, such as computing, storage and synchronization resources. Resources can also be managed and brokered and can be scheduled. This notion allows to model shared and mutually exclusive resources.

- ***Allocation:*** The allocation package permits allocating the application model onto the architecture model. With combination of the *distribute* concept introduced in the repetitive structure modeling (RSM) annex, the allocation of multiple tasks on multiple/single processing units can be carried out. The allocation can either be *spatial* or *temporal* in nature.

- ***Generic Component Modeling (GCM):*** Permits to define concepts such as *components*, *ports* and *instances*. The SysML concepts of block diagrams and *flow ports* have been integrated in MARTE.

- ***High Level Application Modeling (HLAM):*** The high level application modeling package in MARTE permits describing features and functionalities related to real-time systems or RTS. Quantitative features such as deadlines as well as qualitative features related to communication and behavior can also be addressed and expressed.

- ***Detailed Resource Modeling (DRM):*** This package is divided into two sub packages:

  - ***Software Resource Modeling (SRM):*** The SRM package in MARTE is used to describe parts of standardized or designer based RTOS APIs. Thus multi-tasking libraries and multi-tasking framework APIs can be described with this package.

  - ***Hardware Resource Modeling (HRM):*** The hardware concepts in MARTE allow to represent hardware architectures in several views. The views can be either functional, physical or hybrid in nature.

- ***Generic Quantitative Analysis Modeling (GQAM):*** The GQAM package enables designer to focus on analysis via the MARTE profile. The analysis can be for the software behavior (such as schedulability and performance) as well as other aspects such as power, energy, fault tolerance, etc. The GQAM package gives the description of how the system behavior uses the available resources. This package helps to determine timeliness of a response (such as hard/soft deadlines) and other statistical measures such as average delays. Similarly, memory and power usage are also addressed in this package.
- ***Schedulability Analysis Modeling (SAM):*** The SAM package extends the GQAM package for the purposes of schedulability analysis. Schedulability of a system can be either related to the system itself or related to a sub-module, for example to meet certain constraints such as related to time (e.g., deadlines, miss rations). Schedulability analysis also helps in the optimization of the system. A system can be analyzed under different scenarios or input values in order to observe the differences.
- ***Performance Analysis Modeling (PAM):*** The PAM package extends the GQAM package for the analysis of temporal properties of real-time embedded systems.
- ***Value Specification Language (VSL):*** The language which is to be used in NFP constraints. VSL defines data types, parameters, constants, enumerations and expressions. These VSL expressions can be used to specify non-functional parameters, values, operations, values and dependencies between different values in a model.
- ***Repetitive structure package (RSM):*** Enables compact expressions to represent massively parallel applications and architectures. Grid and cube topologies can also be expressed, as well as interconnection topologies present in NoCs and multistage interconnection networks [199, 206].
- ***Clock Handling Facilities:*** This annex provides the abstract syntax for specifying clock dependencies and clocked values.
- ***MARTE Libraries*** This annex defines predefined MARTE libraries for primitive types, extended data types, as well as a time library that defines enumerations for time concepts.

3.2.4 Comparing MARTE with other existing standards and profiles

In this section, we provide a brief overview of MARTE with some popular existing profiles and standards. This allows interoperability between different designers, research teams and industrial partners and helps in the collaboration efforts. If there are compatible concepts present in the profiles or standards, it can allow designers to port their modeling specifications to the MARTE profile. This allows them to keep up to pace with current industry evolution and to take advantage of associated modeling tools and technologies.

3.2.4.1 MARTE and existing profiles

While there exists a large number of UML profiles for modeling of real-time embedded systems and SoC in particular, we have only focused on some of the most popular profiles associated with the SoC domain.

Comparing MARTE with SysML SysML and MARTE are complementary: SysML allows to model requirements in the early design phases while MARTE defines concepts to model the timing and the non-functional aspects and is more suitable for the later design phases. SysML permits description of the traceability requirements, and provides means to express the behavior and composition of the system blocks. This profile also provides the designer with parametric formalisms that are used to express analytical models based on equations. A detailed comparison of the two profiles has been presented in [84].

Comparing MARTE with UML for SoC and UML for SystemC As described before, some of the existing UML profiles for modeling of real-time embedded systems such as UML for SoC and UML for SystemC are too closely linked to the execution platforms; and lack the high level abstractions required for the design and development of complex SoCs. As compared to these profiles, MARTE provides a light-weight component based UML profile at a higher abstraction level. In addition, MARTE enables specification of non-functional properties of SoC components, which is not possible with these profiles.

Comparing MARTE with UML profile for SPT The UML profile for SPT was OMG's first UML profile for the specification of real-time systems. The profile included support for performance analysis with mechanisms such as queuing theory and petri nets. It also provided schedulability analysis, but lacked mechanisms for modeling hardware and software platforms. Additionally, the constructs present in the profile were found to be too abstract in nature. MARTE overcomes all these limitations, and incorporates concepts such as schedulability analysis and performance observation, via the GQAM, SAM and PAM packages.

3.2.4.2 MARTE and existing standards

There exist a large number of modeling frameworks that support the design specification, development and analysis of real-time embedded systems. Each of these frameworks has its specific strengths and weaknesses. We compare MARTE to some of these frameworks, in order to determine common concepts that will permit increased synergy between designers of different domains. Additionally, interesting concepts present in these frameworks can be the basis of future extensions of the MARTE profile.

MARTE and AADL. AADL, that has its origins in the avionic domain, is a SAE[8] standard for the development of real-time embedded systems. In AADL, the design can be represented in the forms of processes and threads which can interact via port connections, program calls and shared data access. Once the application has been modeled, it is mapped or binded to a target platform. In [88], the authors compared the relationship between AADL and MARTE. By utilizing MARTE for modeling AADL applications, designers can model their applications at earlier design stages. Similarly models can be conceived for different views related to time properties, performance and scheduling. Once the applications have been developed, designers can take advantage of existing AADL validation and verification techniques and tools. These validation/verification aspects would come as a compliment to current MARTE aspects.

MARTE and AUTOSAR. AUTOSAR (Automotive Open System Architecture) [83] is a standardized and open automotive software architecture framework, developed jointly by different automobile manufacturers, suppliers and tool developers. Its main goal is to define a complete software architecture with standardized APIs and configuration files for automotive applications as well as the basic software, that permits exchanging parts of the system's software in ways that programmers know from manipulating Java or C++. With regards to AUTOSAR, MARTE already covers many aspects of timing. One important example is the specification of *over-sampling* and *under-sampling* in end-to-end timing chains (commonly found in complex control systems).

3.3 Modeling reconfiguration concepts with MARTE

As described in the previous chapter, reconfiguration is an important aspect of a component based design methodology dealing with modern real-time embedded systems. While standards such as AADL [5] introduce the notion of *modes* related to a system, these concepts are not well integrated in the MARTE profile and up to the writing of this dissertation, lack corresponding modeling tools support.

[8] Society of Automotive Engineers: http://www.sae.org/servlets/index

In AADL, each mode is associated with a system configuration, and events can trigger a change from one configuration to another. Hence, dynamic reconfiguration of the system is possible by the utilization of operational modes and mode transitions. Similarly SysML makes heavy usage of the traditional UML state machines to determine the different states of a system. While the MARTE profile introduces abstract concepts for modeling system behavior, it is up to the modeling tools and the underlying model transformations to accurately interpret these concepts for code generation.

Additionally, although MARTE provides adequate modeling semantics for describing computing resources such as processors and ASICs, it is not enriched enough to provide a detailed FPGA model at the high abstraction levels. Figure 3.9 shows the profile concepts related to computing resources as present in the MARTE *HwComputing* sub-package of the HRM package.

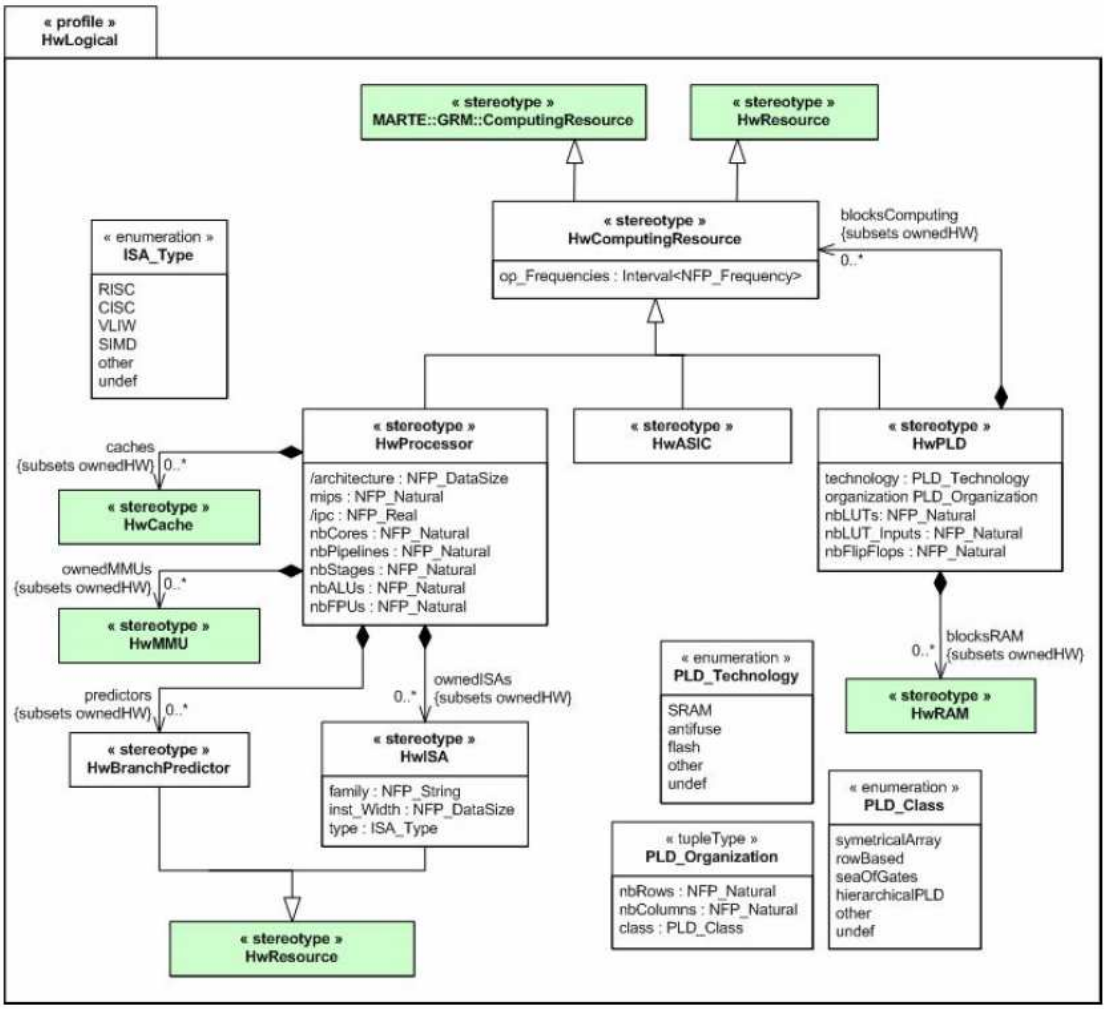


Figure 3.9: Concepts related to the FPGA modeling in the MARTE HRM package

The *HwComputing* sub-package in the HRM functional view defines a set of active processing resources pivotal for an execution platform. A `HwComputingResource` symbolizes an active processing resource that can be specialized as either a processor (`HwProcessor`), an ASIC (`HwASIC`) or a PLD (`HwPLD`). An FPGA is represented by the `HwPLD` stereotype; it can contain a RAM memory (`HwRAM`) (as well as other `HwResources`) and is characterized by a technology (SRAM, Antifuse etc.). The cell organization of the FPGA is characterized by the number of rows and columns, but also by the type of architecture (Symmetrical array, row based etc.). A processor may contain some instruction set architectures (ISAs); and can have some `HwCaches`, `HwMMU` (main memory units) along with zero or more branch predictors (HwBranchPredictor). Firstly, the concepts related to representing a processor are not sufficient for a complex SoC

on FPGA design, in which a complex processor can either be implemented as a softcore IP or integrated as a hardcore IP. Thus additional concepts are needed to address this limitation. Similarly, the concepts for `HwPLD` can be used for FPGA modeling; however, more details need to be added to this concept such as number of DSP blocks, registers, Block-RAMs etc. Another solution would be the definition of a specific *'FPGA'* metaclass as a stereotype in the profile, that inherits from the `HwPLD` concept and adds the additional attributes to that class. Similarly when communication between hardware components takes place, the latency and bandwidth needs to be treated. While HRM takes bandwidth into consideration, the latency has to be addressed as well.

The main limitation for modeling of reconfigurable architectures such as FPGAs is the ability to bridge the gap between high level models and the commercial tools provided by FPGA vendors for implementing features such as partial dynamic reconfiguration. The high level models are usually too abstract in nature and do not provide adequate mechanisms to take into account issues such as floor planning, placement of reconfigurable modules, etc.; which have been briefly detailed in chapter 1. Similarly, automatic generation of processors along with their instruction set simulators[9] (ISS) is a daunting task from high level models. While a MARTE based approach has been presented regarding this aspect in [27], they are only able to produce code for the TLM PA abstraction level. Additionally, albeit MARTE provides a layout mechanism for modeling hardware architectures in grid like models, these specifications are not supported by current modeling tools. Evolution of these modeling tools may make it possible to take aspects such as floorplanning into account.

As this dissertation makes use of the MARTE profile for modeling reconfigurable FPGA based SoCs, it is crucial to specify some modeling methodologies to express reconfigurability at the high modeling levels. Regarding this, we have proposed an initial extension of the MARTE profile for the high level modeling of reconfigurable FPGA based SoCs [203, 205], that is presented in appendix A. These works were the inspiration of the research presented in [127]. However, as explained in chapter 1, the authors are only able to generate a textual description related to the hardware components in the modeled FPGA, which is taken as input by the FPGA tools for eventual manual manipulation.

Hence, due to current limitations of the current MARTE profile to effectively model reconfigurable architectures such as FPGAs at high modeling levels; and our initial motivation to provide an application driven partial dynamic reconfiguration design flow, this dissertation primarily focuses on the modeled applicative part which is afterwards transformed into a hardware functionality as explained in chapter 7. Dynamic reconfiguration is still the main aim for this dissertation, and an application driven high modeling approach for introducing system configurability has been introduced later on in the dissertation.

3.4 Conclusions

As previously mentioned, MDE has several advantages: the possibility of platform-independent modeling without involvement of implementation details; re-usability and productivity of models; modeling and specification of different facets of a system from different points of view and rapid automatic model transformations. We have also briefly compared different UML profiles and specially those related to RTES. While SysML stands out as a strong candidate, MARTE is slowly becoming the preferred de-facto industry standard for the modeling of RTES and SoCs. MARTE shares common concepts with ADLs such as AADL and other standards and UML profiles. This will enable designers to port their design models in MARTE, in order to benefit from its advantages and available tools and methodologies.

However, MARTE while suitable for modeling purposes, lacks means to move onto execution platforms. Suitable modeling concepts need to be introduced to address this lack. In the subsequent chapter, we introduce a SoC Co-Design framework that bridges the gap between MARTE and targeted platforms and technologies. Similarly reconfiguration aspects are drastically lacking in MARTE, and appropriate high level modeling mechanisms must be introduced in the profile. This issue has been addressed in chapter 6.

[9] http://en.wikipedia.org/wiki/Instruction_set_simulator

Chapter 4

Gaspard2: An MDE-based framework for SoC Co-Design

In this chapter we present the Gaspard2 SoC Co-Design framework, as a solution to the development of high performance embedded systems; addressing the aforementioned challenges related to SoC Co-Design mentioned in chapter 1. The usage of a component based approach in combination with high abstraction levels as specified in the last two chapters can aid in reduction of design development time and the inherent complexity. Thus this framework offers high benefits for the design, development and eventual implementation of complex SoCs.

Gaspard2 (Graphical Array Specification for Parallel and Distributed Computing) [2, 63] is a MDE oriented SoC Co-Design framework that utilizes a *subset* of the MARTE profile currently supported by the SoC industry. Here, by framework, we mean an environment that provides designers with at least the following means: a formalism for the description of embedded systems at a high abstraction level, a methodology covering all system design steps; and a tool-set that supports the entire design activity. In Gaspard2 as in MARTE, a clear *separation of concerns* exists between the hardware and software models. Gaspard2 provides an *Integrated Development Environment* (IDE) dedicated to the visual co-modeling of embedded systems; and has been developed within the DaRT project-team at INRIA Lille-Nord Europe. The framework enables fast design and code generation with the help of UML graphical tools (e.g., MagicDraw UML[1] and Papyrus[2]) and Eclipse EMF.

Figure 4.1 shows the global architecture of the Gaspard2 framework. The main features of Gaspard2 are classified into three categories:

[1] www.magicdraw.com/
[2] www.papyrusuml.org/

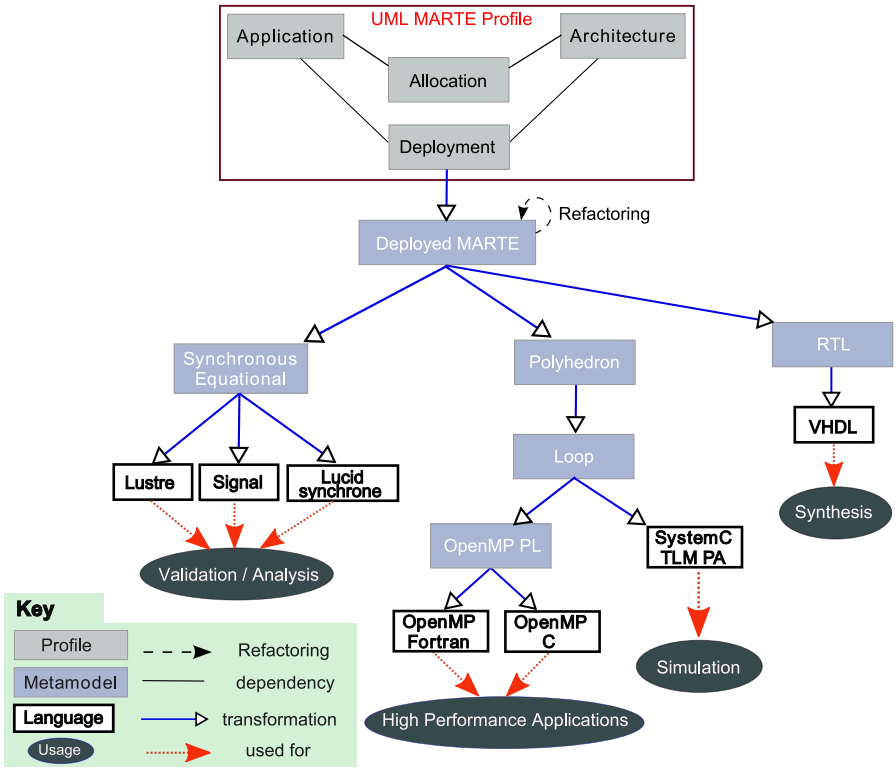


Figure 4.1: A global view of the Gaspard2 environment for SoC design

High-level co-modeling: It enables a co-joint specification of the application and the architectural parts of a system using the MARTE profile at a high level of abstraction, which provides various packages for the modeling concepts related to software, hardware, allocation, etc.

Model transformations: MDE transformation chains have been developed to generate executable code for different domains and targets.

Usage of the generated code: Gaspard2 is capable of targeting different execution platforms and technologies. Code can be generated for different goals, such as *validation* (with synchronous languages), *simulation* (at *Transaction Level Modeling Pattern Accurate* or TLM PA level [47, 71], *synthesis* (at *Register Transfer Level* with VHDL code) and *execution* (with OpenMP or Fortran).

Gaspard2 has strongly contributed to the development of the MARTE UML profile and its corresponding metamodel. The *Repetitive Structure Modeling* (RSM) package in MARTE and its *Model of Computation* (MoC): Array-OL [36], have been inspired from Gaspard2 . The *Hardware Resource Modeling* (HRM) package also inspires from architectural aspects present in Gaspard2. Similarly, certain aspects have also been integrated into the MARTE *Allocation* package.

4.1 Application domain of Gaspard2

Gaspard2 aims to improve the design of SoC based embedded systems with a strong focus on intensive signal processing applications. *Signal processing* can be considered as one of the most important SoC application domains. It concerns the interpretation, analysis, storage and manipulation of *signals*, which can be signals of sound, image, video, radar, etc. A signal is the carrier of the information of interest. According to the different signals to be processed, these applications can be classified into: *analog signal processing* (where signals are captured by sensors, which are not yet digitized) and *digital signal processing* (digitized signals that can be processed

by SoCs or computers directly). Only digital signal processing is considered here, which includes filtering, removal of noise, information extraction, compression and decompression, etc.

Among various types of digital signal processing, we are interested in *intensive signal processing* (ISP), which is always decomposed into two steps: *systematic signal processing* (SSP) for the first step and *intensive data processing* (IDP) for the second. SSP mainly consists of a chain of filters and involves regular processing of large amounts of signals, which are independent of signal values. It results in a characterization of the input signals with values of interest. Whereas, IDP is considered as *irregular* processing because the processing results rely on signal values. Figure 4.2 shows the relation between these classifications of digital signal processing.

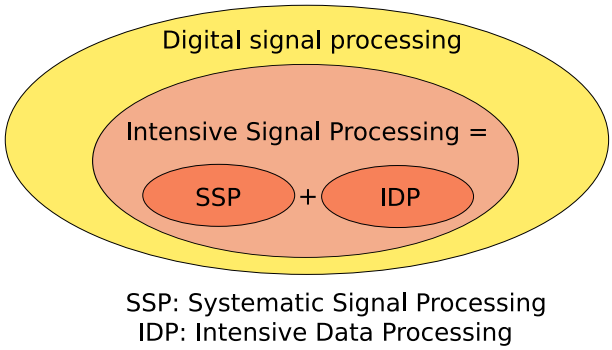


Figure 4.2: Global overview of Digital Signal Processing

Some application examples. Some typical examples of intensive signal processing are presented here, which include:

- ***Anti-collision radar systems:*** Collision avoidance in a system (such as a moving vehicle) requires the use of an antenna (or antennas). The antenna sends a signal as a data flow containing information relating to the presence of obstacles, i.e. an echo. This information is masked by noise (interference) and is spread over the temporal data flow. A pre-treatment systematic signal processing filters the signal in order to bring out the interesting features (the presence of obstacles). The next step is more irregular and consists of analyzing this data flow to validate the presence of an obstacle, trigger an action (such as performing an emergency brake) or to continue normal functioning if necessary. An overview of the above mentioned mechanism is illustrated in Figure 4.3.

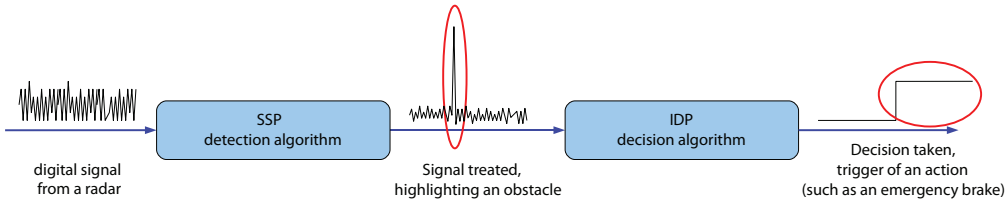


Figure 4.3: Mechanism related to an anti-collision radar system

- ***Sonar signal processing:*** A submarine is equipped with several hydrophones, which are used for listening to and recording underwater sounds. A classical sonar signal processing chain is composed of several stages. The first stage involves systematic signal processing, which includes FFTs. FFTs add a frequency dimension to the processed signals. The results are used in the subsequent stages for communication or object detection purposes.

- ***Image encoding/decoding:*** JPEG 2000 is a wavelet-based image compression standard. The encoder [8] can also be divided into several stages. The first few stages are considered to be systematic processing, such as *color components transformation* and *tiling* stages.

The subsequent stages involve irregular processing, such as *wavelet transform*, *quantization* and *coding*. The decoder works in an inverse way: irregular phases are followed by systematic phases.

- ***Aspect ratio converting:*** the conversion of a high-definition video format (16:9) to a standard-definition (4:3) [150] can also be divided into two stages: the first one consists of line processing of the original 16:9 video signals in order to create pixels through interpolation, the results are then processed by removing some lines so that the final ratio is set to 4:3.

These examples show how signal processing can be divided into stages, such as SSP and IDP. As SSP is independent from signal contents, it is possible to use certain *generic* models for processing specification. However, IDP involves the processing of signal contents, which may vary from one to another according to the signal contents. Hence it is not appropriate to use some generic models for the computing specification.

Multidimensional arrays. *Multidimensional arrays* are often used as the main data structures in SSP applications. As signal contents are not involved in the processing, it is appropriate to abstract them, which facilitates the modeling of SSP. Consequently, array *type* and array *shape* are sufficient for the modeling.

The previous examples also illustrate various semantics related to signal dimensions (2-Dimensional images, temporal dimensions, frequency dimensions, etc.). For instance, the temporal dimension can be represented by an infinite dimension. The dimension number of a signal can also be changed (increased or decreased) in the processing. Moreover, some applications can have signals with *toric* dimensions, i.e., data stored in these dimensions are processed in a *modulo* way.

Some languages for signal processing. There exist numerous languages for the specification of signal processing applications. Here, we only provide a brief overview of some of these languages; and a detailed comparison between the existing languages and their underlying model of computations can be found in [97]:

- *StreamIt* [248] and *Synchronous Data Flow* (SDF) [144] are stream processing languages, but they are not considered to be multidimensional languages for signal processing. StreamIt is an object-oriented imperative language that is intended to allow maximum optimization for the specification of synchronous dataflow at a high level of abstraction. The extension of SDF, MultiDimensional SDF [167] is a multidimensional language, whose applications are described using oriented acyclic graph. The nodes, called actors in the graph, consume and produce data, called tokens.

- The A*lpha* language [155] is a functional language, whose applications are composed of systems of recurrent equations. Alpha is based on *polyhedral* model, which is extensively used for automatic parallelization and the generation of systolic arrays. Alpha is a multidimensional language with single assignment specification.

- *High-performance Fortran(HPF)* [111] is a language dedicated to scientific parallel computing. It takes high levels of abstraction into account. HPF uses multidimensional arrays in parallel loops, where operations are carried out on sub-arrays. HPF also enables regular data distributions.

- Synchronous dataflow languages also define arrays in order to deal with specific algorithms and architectures. For instance, in *Lustre*, array concept has been introduced in order to design and simulate systolic algorithms [106]. This work led to the implementation of their results on circuits [214]. More recently, an efficient compilation of array in Lustre programs has been proposed [164]. It is similar to the S*ignal* language. In contrast, concept of array of processes [29] has been introduced in Signal, which is adapted to model systolic algorithms.

- Finally, Array-OL [36, 37] is also a multidimensional language for the specification of intensive signal processing, which is the underlying MoC of the *Repetitive Structure Modeling* package present in the MARTE profile. Gaspard2 makes heavy use of the RSM package, and tries to remain compatible with Array-OL semantics.

4.2 High-level co-modeling for SoC design

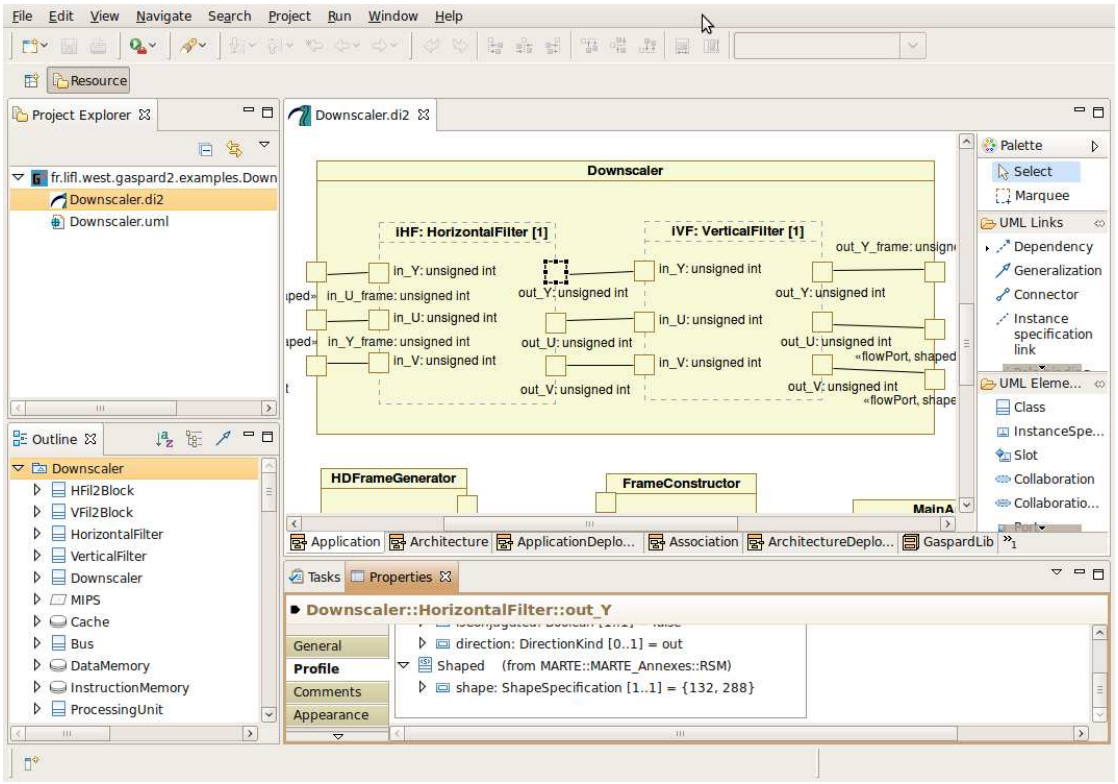


Figure 4.4: The Gaspard2 SoC Co-Design environment

With the goal of making the Gaspard2 framework more complaint and *standard*, to help facilitating in interfacing with other tools; and to profit from the related existing tools and technologies, efforts have been carried out within the DART team to render Gaspard2 completely compatible with MARTE.

One of the most important features of Gaspard2 is its ability for system co-modeling using the MARTE profile at a high level of abstraction. More precisely, it enables to model *software applications*, *hardware architectures*, their *allocations* and IP *deployment* separately, but in a unique modeling environment. This concept is partially based on the Y-chart (Figure 4.1 and [63]). In Gaspard, models of software applications and hardware architectures can be defined concurrently and independently. Then, software applications can be mapped onto hardware architectures via an allocation.

Although MARTE is suitable for modeling purposes, it lacks the means to move from high level modeling specifications to execution platforms. Gaspard bridges this gap and introduces the notion of IP deployment. This level associates every elementary component, of both the hardware and the application, to an implementation, thus facilitating IP reuse. Until the deployment level, the integrated high abstraction level models are platform-independent, i.e., they are not associated with a specific execution platform or technology.

Gaspard2 also profits from the recent development of the open source Papyrus UML editor which is integrable in the Eclipse environment. Using this approach, all the current development in Gaspard2, from high level modeling to automatic code generation, is structured

around the Eclipse platform. Thus added extensions in Gaspard2 can be developed and easily integrated in Eclipse. Figure 4.4 shows a screen shot of the Gaspard2 environment with modeling of a system in Papyrus.

4.2.1 Component based modeling

Gaspard2 adopts a component based approach based on the UML MARTE profile. For this, it uses the MARTE *Generic Component Modeling Package* (GCM) as its core foundation.

A *Structured component* [181] in the GCM package as shown in Figure 4.5 can be defined as the UML *Classifier* concept [178] and relies mainly on UML structured classes. Thus it can be used for the modeling of both class and component structures in their respective diagrams. A structured component may also contain several properties such as *assembly parts* and *interaction ports*. An assembly part can be viewed as an instance of another structured component, defined elsewhere in the specification.

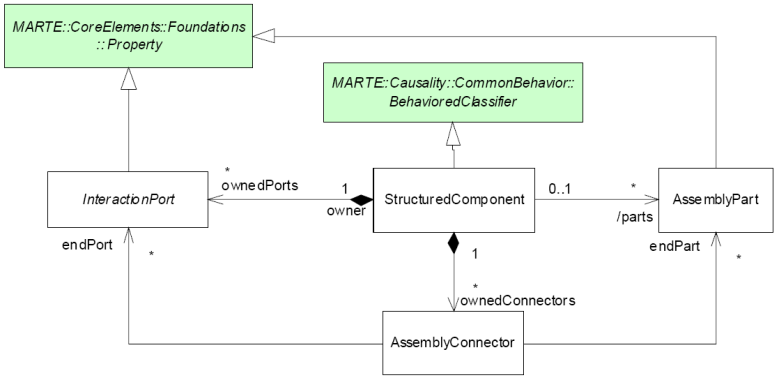


Figure 4.5: Concepts related to the MARTE *StructuredComponent* in the GCM package

A structured component can also have internal *connectors*. These connectors are termed as MARTE *assembly connectors*. MARTE does not distinguish explicitly between the connector types in comparison with pure UML specifications. The assembly connectors can thus play the role of either delegate connectors, which permit connecting the ports of a structured component to the ports of its sub components; or typical assembly connectors (which allow connecting the input/outputting ports of different sub components).

An *Interaction Port* defines a point of interaction via which different components interact and are linked by means of an assembly connector. It is not necessary for a component to have an interaction port. An interaction port can be classified into two major types:

- ***Flow Port:*** mainly used for flow oriented communications schema e.g. control/data flow. It can have several directions (*in*, *out* or *inout*). This concept bears resemblance to the flow port concept defined in the SysML profile. In Gaspard2, mainly flow ports are used for the modeling purposes.

- ***Message Port:*** Message Ports are used for message oriented (request/reply) communication schema. They can support provided or/and required services.

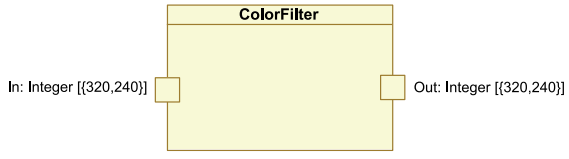


Figure 4.6: An example of a Gaspard2 component modeled with the MARTE profile

Finally, Figure 4.6 illustrates an example of a MARTE structured component in Gaspard2 environment. The `ColorFilter` component has respective input and output ports with associated *types* and *shapes*.

4.2.2 Repetitive structure modeling

One of the key MARTE packages, the *Repetitive Structure Modeling* (RSM) package is inspired from Gaspard2. Gaspard2, and in turn RSM, are based on the Array-OL [36, 37] model of computation 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. RSM permits to describe the regularity of a system's structure (composed of repetitions of structural components interconnected in a regular connection pattern) and topology in a compact manner. In order to fully comprehend the RSM package and consequently the Gaspard2 framework, we provide a brief summary about Array-OL, the model of computation for RSM. Detailed information related to Array-OL can be found in [36, 37].

4.2.2.1 A high-level data dependency model: Array-OL

Array Oriented Language (Array-OL) was first proposed by Alain Demeure ([68] in French and [67] in English) at THALES Underwater System (TUS) in 1995. It is dedicated to the specification of intensive signal processing where large number of signals are regularly processed by a set of repeated tasks. Its typical applications include radar/sonar signal processing and multimedia (image, audio and video) processing.

The first thing to remember is that Array-OL is not a programming language, but only a specification language. Thus no execution concerns are involved in the language, and no rules for executing an application written with Array-OL are present. Instead of specifying certain specific scheduling of parallel tasks, only data dependencies between these tasks are specified. Some tasks that achieve some computing functionalities, such as filters and FFT, are referred to as *elementary* tasks, which are considered as *black boxes* provided with interfaces.

Basic characteristics. The basic goal of Array-OL is to provide a mixed graphical-textual language for modeling of multidimensional intensive signal processing applications. As these applications treat a massive amount of data with tight real-time constraints, effective use of potential parallelism of the application or parallel hardware architectures is obligatory.

As mentioned before, detailed characteristics of this domain-specific language have been presented in [36, 37]. Here we only mention some key points:

- *Multidimensional arrays*: Data manipulated in Array-OL are in the form of multidimensional arrays, which have at most one possible *infinite dimension*. These arrays can be specified with a certain type specification, such as an array *shape*. Nevertheless, data types, e.g., *Integer* and *Boolean*, are unnecessary, because data values stored in the array are not handled. Consequently data values are concealed. These features imply that only array spatial manipulations are involved in the language. Moreover, these arrays can be *toroidal*. This characteristic enables to model some spatial dimensions that represent some physical tori (e.g. hydrophones around a submarine). Other examples are some frequency domains obtained by FFTs.

- *Data dependency expressions*: Array-OL expresses true data dependencies in order to describe maximum parallelism in the application. In such a way, except for the minimal partial order, which results from the specified data dependencies, no other order is *a priori* assumed.

- *Patterns*: Access to data is carried out in the form of sub-arrays called *patterns*.

- *Spatial and temporal specifications*: The spatial and temporal dimensions are treated in the same manner, in the form of arrays. Particularly, time is expanded as one dimension of arrays. This is a direct consequence of the single assignment property of Array-OL.

Array-OL is not a data-flow language, but can be projected as such. The language does not manipulate flows, but instead focuses on multidimensional arrays. The environment or an execution platform can impose an order and a granularity on the array elements in order to treat them as a flow, but the choice of computation granularity is left to the compiler (or the designer) and not the person doing the modeling (for example, with the same Array-OL specification, a video represented in the form of a 3D array of pixels can be viewed as a flow of images, flow of lines or even a flow of pixels).

Array-OL is not limited for the specification of signal processing, other similar processing, e.g., data-intensive processing, are also its application domain. Hence, we call all these kinds of processing *data-parallel intensive processing* (DIP), which defines the application domain of Array-OL and Gaspard2, and is one of the contexts of this thesis.

Array-OL utilizes the multidimensional array data structure for the specification of intensive data, which benefits from several advantages: toric arrays can be specified for some special applications, such as sonar signal processing and frequency processing; temporal and spatial dimensions are processed in the same way, hence a maximum parallelism is specified, which can be refactored according to an architecture when the application is mapped onto the architecture.

ADO (*Array Distribution Operators* in English, *Opérateurs de Distribution de Tableaux* in French) operators permit a high-level data dependency specification (such as patterns) with regards to the manipulation of arrays indexes, as patterns are also arrays. These kinds of dependencies enables the specification of multi-granularity degrees, which make the application specifications flexible. Array-OL only specifies data dependencies, and is independent from any execution model, which contributes to a fast application specification. Properties, e.g., single assignment, are defined in Array-OL to guarantee the correctness of specification. These characteristics make Array-OL distinct from other languages in the same application domain, such as SDF, Alpha, StreamIt, synchronous languages and HPF. However, as Array-OL is a specification language, it is possible to project it onto the execution models provided by the previously mentioned languages [11, 76, 241, 265].

4.2.2.2 Overview of RSM

We now present some of the basic core concepts of the RSM package. The available RSM mechanisms are oriented towards two aspects:

RSM enables the possibility to specify the *shape* of a repetition, by a multidimensionality, and also permits to represent a collection of potential links such as a multidimensional array. This repetition can be specified for an instance or a port of a component. The advantage is double fold: For hardware modeling, RSM presents a clear mechanism for expressing the links in a topology, as well as increasing the expression power of the mechanism for describing these complex topologies [199, 202]. Secondly, RSM provides a method of adding information about topological relations between entities during their specification. It also permits to express topological links between the entities at run-time execution. Complex regular, repetitive structures such as cubes and grids can be modeled easily via RSM, in a compact manner. Similarly for application aspects, RSM helps to determine different types of *parallelism*.

Gaspard uses the RSM semantics to exploit the inherent *parallelism* included in repetitive constructions of both hardware and software elements (such as application loops). Large regular hardware architectures (such as multiprocessor architectures) can be modeled in a condensed precise manner. For an application functionality, both *data parallelism* and *task parallelism* can be expressed easily via RSM. A *composite* component contains several parts (subcomponents) and is viewed as an acyclic dependency graph. 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 such a component. 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).

The repetitive component is thus viewed as a *repetition context task* (RCT), which defines an repetition context for a certain *repeated task* (RT), i.e., the RT is repeated in the RCT. An RCT and its RT do not have the same interfaces, thus ADOs connect an RCT with its RT; and define how the input/output arrays of the RCT are regularly accessed by the RT. Repetitions (or instances

according to different context) of an RT are supposed to be independent from one another in general. Finally *elementary* components can also be specified via RSM. An elementary task is atomic (a black box) in nature and can be a part of a library.

Finally, the integration of RSM in the MARTE allocation package enables expressing temporal and spatial allocation of the application onto the hardware platform. Figure 4.7 shows the basic core concepts of the RSM package.

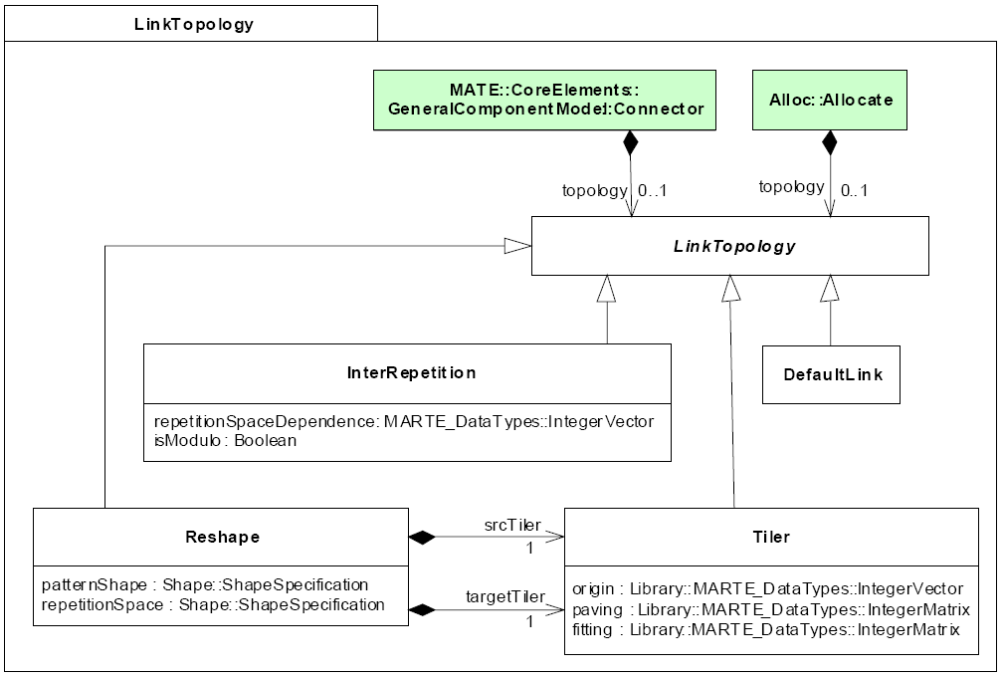


Figure 4.7: RSM package of the MARTE profile

The concept of *Shape* can be assigned to an instance or a port of a component. It is specified through the *multiplicity* property of the instance or the port, which implies a collection of the corresponding elements. This collection is defined in the form of a *multidimensional array*, whose elements are ordered positive integers indicating the maximum number of elements stored in the corresponding array dimensions. For instance, a shape of $\{40, 30\}$, defined for a repeated task in a repetition context indicates that the task is repeated 40×30 times. This shape is also called the *repetition space* of this task; and is also a multidimensional array. The product of all the elements in the repetition space determines the number of repetitions of this RT. This concept is inspired by the bound notion of parallelly nested loops, which are present in high-performance computing languages [36]. The same shape $\{40, 30\}$ on a port indicates an $[40, 30]$-array that is processed by the component owning the port. This shape is termed as a *pattern shape* that basically represents the form of a pattern. A shape is extended to have a special dimension, i.e. infinite dimension, which is the result of mapping some discrete time computing (or dataflow) onto a space model.

A *Tiler* represents a special connector, used in a repetition context, and is associated with some topological information for array processing. A tiler describes how an array can be cut into sub-arrays with the same shape in a regular way or how some sub-arrays are used to build an array. These sub-arrays can also be multidimensional arrays, which are inputs/outputs of the RT. Whereas, the whole arrays are inputs/outputs of the RCT. In order to distinguish the different usages of these subarrays, they are called *tiles* in the case that they are a part of an array that belongs to an RCT, in contrast to *patterns*, which are taken as inputs/outputs of an RT. A tiler defines: a *fitting* matrix describing how array elements fill tiles; an *origin* of the *reference pattern* and a *paving* matrix describing how tiles cover arrays.

A *Reshape* connects two arrays, and can be considered as an array transformation between these two arrays. The values stored in the arrays remain unchanged after the transformation.

A reshape has two tilers at each end, which explain how to *displace* a tile from the source array to the target array. A reshape represents run-time links between the source and target array and enables to represent complex link topologies, in which the elements of a multidimensional array are redistributed in another array.

An `InterRepetition` dependency (IRD) 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 task and its owner RCT. Particularly, an interrepetition dependency specification leads to the sequential execution of repetitions of the repeated task. It connects one of the outputs of an RT with one of its inputs in the condition that the type of this input and output must be identical. From the point of view of the RT itself, this interrepetition dependency makes it self-dependent. However, in the repetition context of the RT, a *dependency vector* associated to the interrepetition dependency is used to ensure that one repetition of the RT relies on another one (or some repetitions rely on some other ones), i.e., it expresses the dependency relation in terms of a vector. If the depended repetition is not defined in the repetition space, a default value is then chosen.

A `defaultLink` connector provides a default value for the repetitions of a task which are linked with an interrepetition dependency, with the condition that the source of the dependency is absent.

For RSM, the multidimensional modeling requires the specification of vectors and matrices for the connectors (such as tiler, reshape, etc.) of the repetitive structures; and the *shapes*, for multidimensional arrays and repetitions. For this, the RSM package makes use of the concepts present in the MARTE library.

As illustrated in Figure 4.8, an `IntegerVector` is a vector of integer values as defined in the MARTE library. This vector is defined as an ordered list of integer values (including having a possible value of zero). An `IntegerMatrix` represents a matrix of integer values and is represented by a list of integer vectors corresponding to the definition stated previously and have the same shape. In MARTE, the matrices are written in a column by column manner. For example, a matrix of $\left(\begin{smallmatrix}1&2\\3&4\\5&6\end{smallmatrix}\right)$ is represented by $\{\{1,3,5\},\{2,4,6\}\}$. The integer vectors and matrices are used to determine the attributes related to the concepts present in the RSM package: such as tilers and interrepetition connectors.

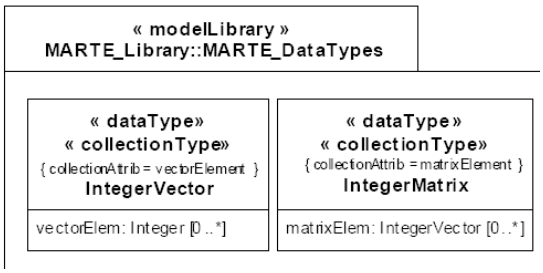


Figure 4.8: An extract of the MARTE data types present in the MARTE Library

4.2.3 Application modeling

In MARTE, a vast variety of applications can be modeled related to different domains. However, as in Gaspard2, we are mainly interested in only data-parallel intensive processing (DIP), the application components in Gaspard2 are not stereotyped explicitly as compared to the hardware components. As this dissertation is mainly interested in the high level application aspects, we thus explain the underlying modeling concepts.

4.2.3.1 Modeling of task parallelism

Figure 4.9 illustrates task parallelism on the top level component of a matrix multiplication application (this main component can be compared to the *main* of a C program). This

`MatrixMultiplicationMain` component is composed of four sub components or parts/instances. The two instances of `initMatrix` (`iM1` and `iM2`) initialize the matrices (for example: either by random initialization or due to data read from a file). These matrices are then provided to the `MatrixMultiplication` component instance which will calculate the matrix multiplication. Finally, the `saveMatrix` component instance will store this matrix in a file, for display or verification purposes.

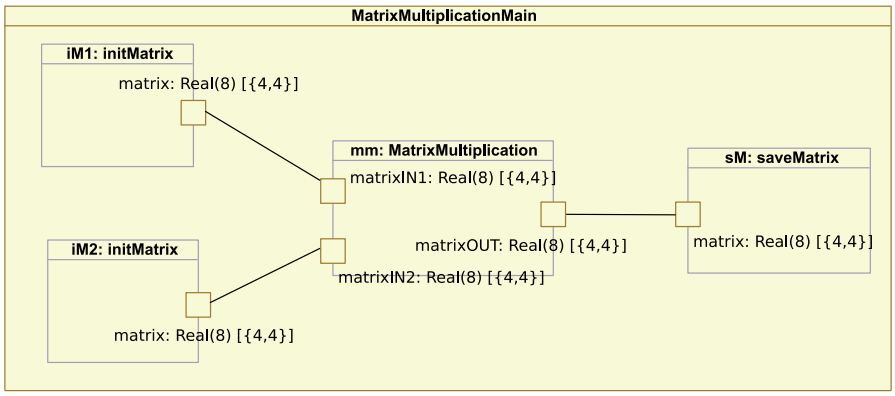


Figure 4.9: Illustration of *Task Parallelism* in Gaspard2 with the MARTE profile

4.2.3.2 Modeling of data parallelism

The modeling of data parallel Gaspard2 applications is based on the expression of data parallelism present in Array-OL. Figure 4.10 represents the algorithmic expression of row/column multiplication of two matrices[3].

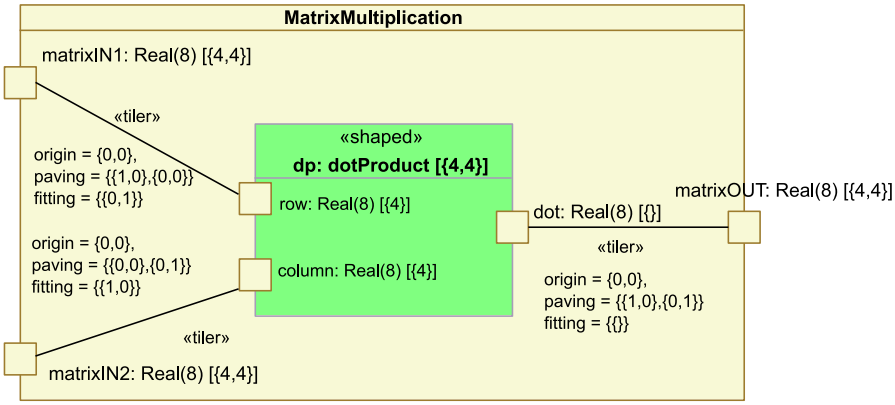


Figure 4.10: Representing *Data Parallelism* in Gaspard2 with the MARTE profile

The component `MatrixMultiplication` is composed of a repeated task: the `dP` instace of the `dotProduct` component. This repeated task represents the computing task, which takes one row and column; and produces one element in the final produced matrix. This task is elementary in nature and is thus represented differently from other tasks; and can be henceforth deployed. The `tiler` connectors express how the repeated task consumes and produces the patterns by the indexes of the repetition in the repetition space. The input ports `matrixIN1` and `matrixIN2` represent the two input matrices, while the `matrixOUT` output port represents the produced matrix.

[3]This algorithm is illustrated on small {4, 4} matrices for a simple example, however it be equally applied on large matrices

4.2.4 Architecture modeling

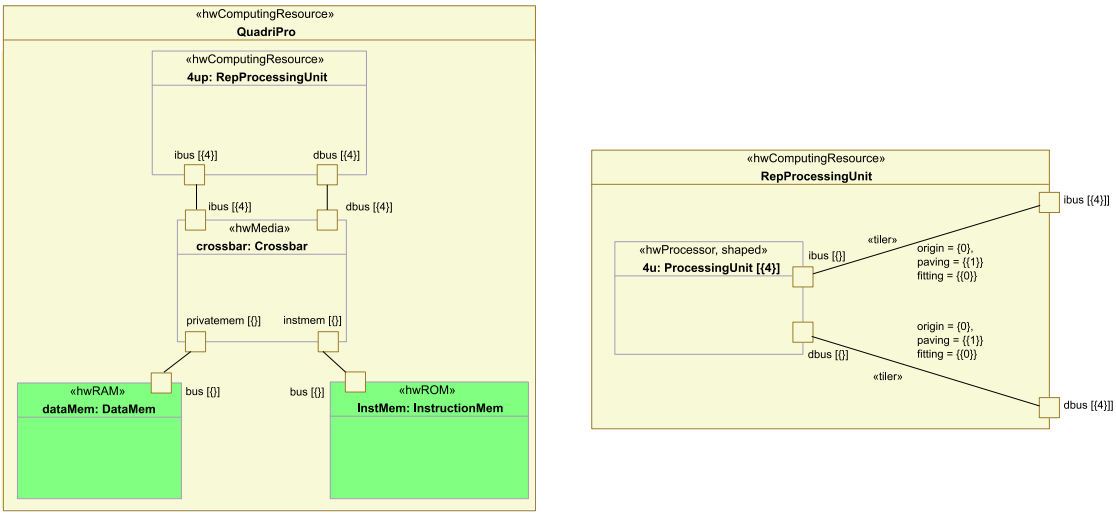


Figure 4.11: Modeling of an architecture QuadriPro with shared memory

Gaspard2 uses the HRM concepts in MARTE for modeling complex hardware architectures. HRM consists of several views, a functional view (*HwLogical* sub-package), a physical view (*HwPhysical* sub-package) or a merge of the two. The two sub-packages derive certain concepts from the *HwGeneral* root package, in which *HwResource* is a core concept that defines a generic hardware entity. A HwResource can be composed of other HwResource(s) (for example a processor containing an ALU). This concept is then further expanded according to the functional or physical specifications. The functional view of HRM defines hardware resources as either *computing*, *storage*, *communication*, *timing* or *device* resources. The physical view represents hardware resources as physical components with details about their shape, size and power consumption among other attributes. Gaspard2 currently only supports the functional view. The HRM also exploits the NFP package for specifying non-functional properties and quantitative annotations with measurement units. The NFP package provides a rich library of basic types like *Data size*, *Data Transmission Rate* and *Duration*.

Figure 4.11 represents the modeling of a QuadriPro architecture with shared memory. The global architecture is modeled on the left side of Figure 4.11. This component is composed of a repetition of processors connected with instruction and data memories via a crossbar. The repetition of the processors is modeled on the right side of Figure 4.11. It is composed of repetitions of the same processor, being repeated 4 times. Here the tilers represent the interconnecting topology between the ports of the processors and the crossbar (via the hierarchy). Detailed information about hardware modeling in Gaspard2 can be found in [27].

4.2.5 Allocation modeling

An allocation permits to associate the applicative part of the system onto the available hardware resources (for e.g. mapping of a task or data onto a processor or a memory respectively). Allocation plays an important role in the overall performance of the system. An allocation can be of two types.

- ***Simple allocation:***

  An allocation can be viewed as an operation consisting of associating an element to an other. In the case of a single task, it can be allocated to a processor which will be responsible for its execution. In the MARTE profile, this operation is realized with the help of a dependency from the source task to the target processor. This dependency uses the `allocate` stereotype present in the profile. This operation is always carried out between two elements having the same multiplicity or shape.

- ***Distribution:***

  The distribution primarily consists of distributing repetitions of one element on to the repetitions of another element. The basic principle of a distribution is to select a pattern in the repetition space of the repeated task; and then to place this pattern on to the repeated processors. This operation must be repeated the number of times it is necessary to at least place once, each repetition of a task on to the processors.

  Further details regarding effective allocation of applications onto hardware architectures in the Gaspard2 framework can be found in [194].

4.2.6 Deployment modeling

Although MARTE is suitable for modeling purposes, it lacks the means to move from high level modeling specifications to execution platforms. Gaspard bridges this gap and introduces additional concepts and semantics to fill this requirement for SoC Co-Design.

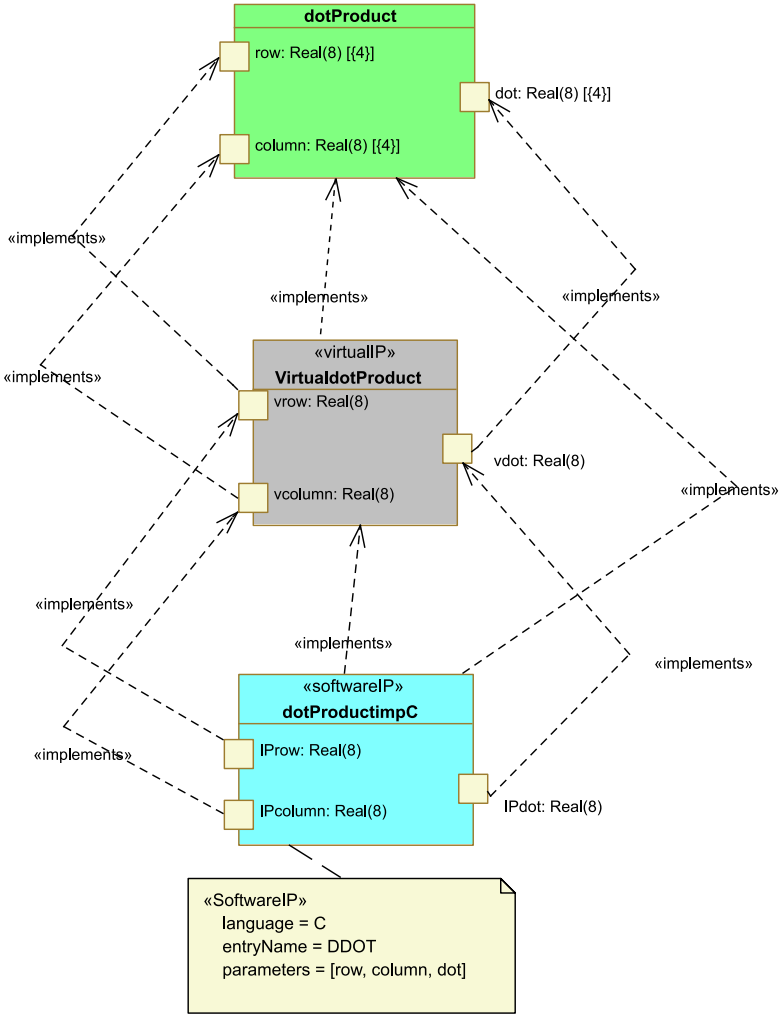


Figure 4.12: Deployment of an elementary component dotProduct

Gaspard defines a notion of a *Deployment* specification level [16] in order to generate compilable code from a SoC model. This level is related to the specification of *elementary* components (ECs): basic building blocks of all other components having atomic functions. Although the notion of deployment is present in UML, the SoC design has special needs, not fulfilled by this notion. In order to generate an entire system from high level specifications, all implementation

details of every EC have to be determined. Low level behavioral or structural details are much better described by using usual programming languages instead of graphical UML models.

Hence, Gaspard extends the MARTE profile to allow deploying of ECs. To transform the high abstraction level models to concrete code, detailed information must be provided. The deployment level associates every EC (of both the hardware and the application) to an implementation (code) hence facilitating Intellectual Property (IP) re-use. Each EC ideally can have several implementations. 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 assembler or C/C++, or implemented as a hardware accelerator using HDLs or SystemC. Hence the deployment level differentiates between the hardware and software functionalities; and permits moving from platform-independent high level models to platform dependent models for eventual implementation. Deployment provides IP information to model transformations to form a compilation chain in order to transform the high abstraction level models (application, architecture and allocation) for different domains: formal verification, simulation, high performance computing or synthesis. Hence deployment can be seen a potential extension of the MARTE profile enabling a complete flow from model conception to automatic code generation. We now present a brief overview of the deployment concepts.

A `VirtualIP` expresses the behavior (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 concept of `CodeFile` is used to specify, for a given IP, the file corresponding to the source code and its required compilation options. The `CodeFile` thus identifies the physical path of the source code. It should be noted that the modeling of a code file is not possible in the *UML composite structure diagram* but is carried out in the *UML deployment diagram* in the current UML model tools such as Papyrus. The desired IP is then selected by the SoC designer by linking it to the EC through an `implements` dependency. As compared to the deployment specified in [16], the deployment level has been modified to respect the semantics of traditional UML deployment.

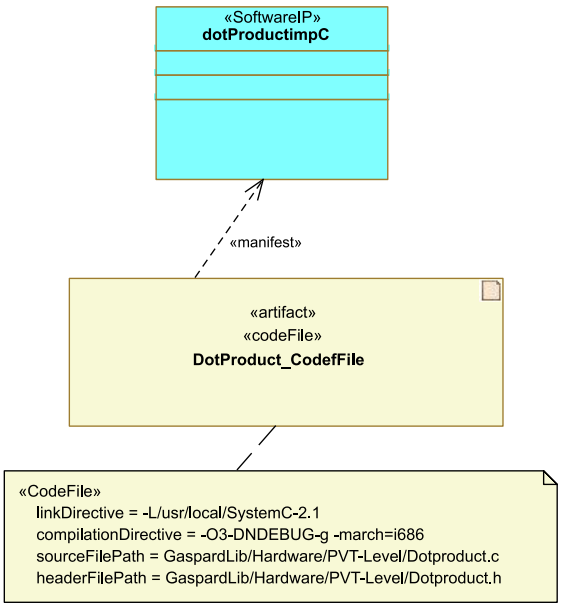


Figure 4.13: Linking an IP to its associated CodeFile

Figures 4.12 and 4.13 show a clear description of the deployment level by deploying an elementary component `dotProduct` of the matrix multiplication application shown in Figure 4.10. At the deployment level, this elementary component can have several possible implementation choices. These choices can be for the same execution platform (and same abstraction level) in a given language, or can be for different ones (different abstraction levels or execution platforms). In the illustrated example, the `dotProduct` has an implementation available for

simulation in SystemC. The *implements* dependency from the `dotProductimpC` component to the `dotProduct` component permits linking the IP (a `softwareIP` in this case) to the elementary component.

A fundamental limitation of the current deployment level is that for final compilation to an execution platform, an elementary component can be linked to *only one* implementation (IP). While this does make sense with regards to platforms where dynamic nature is not relevant, in the case of reconfigurable and especially dynamically reconfigurable SoCs, this is an significant limitation. Thus additional semantics are needed to address this issue.

4.2.7 GaspardLIB

The high level modeling of software and hardware IPs as described precedently is concretized by the development of an IP library termed as *GaspardLIB* [27, 194]. The hardware components of SoCLib [230] as well as internally developed hardware and software IPs have been integrated in this library in order to construct complete complex SoCs. Figure 4.13 illustrates an example of linking an IP to its associated source file present in GaspardLIB.

4.3 Metamodels and model transformations in Gaspard2

This section introduces some of the intermediate metamodels and model transformations present in the Gaspard2 framework.

4.3.1 Domain-specific metamodels in Gaspard2

Once high level models have been specified using the MARTE profile, the Gaspard2 environment provides several intermediate domain-specific models. Each of these models conform to their respective metamodels for eventual model transformations. The metamodels bridge the gap between high level notions and some specific technologies. These metamodels include: *Deployed MARTE*, *Polyhedron*, *Loop*, *OpenMP PL*, *Synchronous Equational* and the *RTL* metamodel.

- Deployed MARTE metamodel bridges the gap between the MARTE profile diagrams and the lower lever metamodels. It is a composition of several metamodels (application, architecture, allocation and deployment) [198].
- Polyhedron metamodel [194] is intended to implicit the allocation in Gaspard2 through polyhedral technique, which enables the representation of a spatial allocation of computing resources (processors) to task repetitions through parametrized polyhedral concepts.
- Loop metamodel [126] has been proposed to refine the polyhedron metamodel for code generation. This metamodel is very closely related to the Polyhedron metamodel. Loop statements are described in this metamodel, compared to polyhedra information in the polyhedron metamodel. Loop statements indicate how certain repetitions of a task are executed by processors. They are therefore parametrized by the processor indexes. SoC simulation and high-performance computing is the targeted application domain.
- OpenMP PL metamodel [126] enables representation of the essential parts of some procedural languages, e.g., Fortran and C, accompanied with OpenMP statements [188]. This metamodel is used for the code generation of high-performance computing on shared memory computers without communication between the processors.
- Synchronous metamodel [104] has been proposed to carry out formal validation and verification of application modeling in Gaspard2. The verification is carried out via synchronous equations.
- RTL metamodel [143] has been proposed to describe hardware accelerators at RTL, that permits to translate high level modeled applications into hardware functionalities for eventual implementation in a target integrated circuit. This metamodel introduces, e.g., the notion of clocks in order to manipulate some of the usual hardware design concepts.

The RTL metamodel is independent from any HDL such as VHDL or Verilog. During the course of this dissertation, this metamodel has been heavily modified to incorporate aspects of dynamic reconfiguration.

4.3.2 Model transformations

Once Gaspard2 models are specified in a graphical environment, model transformations are carried out via a transformation tool. However, as described in chapter 3, since the standardization of QVT, few of the QVT transformation tools are capable to execute large complex transformations such as present in the Gaspard framework. Also none of these engines is fully compliant with the QVT standard.

In order to solve this dilemma, In 2006, an initial transformation tool called MOMOTE (MOdel to MOdel Transformation Engine) was developed internally in the team that was based on EMFT QUERY [81]. MOMOTE is an enhanced Java framework that permits to perform model-to-model transformations. It is composed of an API and an engine. It takes source models as input and produces target models with each conforming to some metamodel. Another advantage of MOMOTE over the then existing transformation tools was that it supported external black box calls: e.g. native function calls, rule inheritance, recursive rule call and integration of imperative code. However, since that time, new tools such as SmartQVT [246] and QVTO [184] have emerged that implement the QVT Operational language and are effective for handling the complex Gaspard2 model transformations.

Currently, in order to standardize the model transformations and to render them compatible with the future versions of the MARTE profile; we have chosen QVTO as the future transformation tool for Gaspard2. Current all the existing MOMOTE based transformation rules for each execution platform have been, or are being converted into QVTO based transformation rules. MOCODE (MOdels to CODe Engine) is another internal Gaspard2 integrated tool that allows automatic code generation and is based on EMF JET (Java Emitter Templates) [82]. Similarly, as MOMOTE based transformation chains are being migrated to QVTO based transformations; for code generation, MOCODE based transformation rules are currently in the development phase of being rewritten from scratch in Acceleo.

The Gaspard2 model transformations are organized as several transformation chains for different languages, as illustrated in Figure 4.1.

- ***From Deployed modeling to Deployed Marte.*** This transformation converts the MARTE UML profile diagrams to a deployed MARTE model (a MARTE model, in which elementary tasks are deployed with respective IPs).
- ***From Deployed MARTE model to Loop model.*** Two successive transformations are defined in order to transform a deployed MARTE model into a Loop model (a model, which conforms to the Loop metamodel). The first one involves the transformation of a MARTE model into a Polyhedron model, where repetitions are expressed by *polyhedrons*, data arrays are mapped on to memories, etc. The second transformation generates a model of loop expressions from the Polyhedron model, which conforms to the Loop metamodel.
- ***From Loop model to SystemC/PA.*** This transformation enables the code generation for SystemC at the TLM-PA level, where data access are based on patterns (instead of bytes). The latter helps to speed up the simulation. The transformation generates the simulation of hardware and software application components. The hardware components are transformed into SystemC modules with their interconnected ports. Part of the application that executes on processors, is generated as sets of dynamically scheduled and synchronized activities. The execution semantics of this part of application complies with the execution model defined for the Gaspard2 MPSoC applications.
- ***From Loop model to OpenMP Fortran/C.*** OpenMP Fortran/C code can also be generated from Loop models. Two steps are involved in this transformation: (1). generation of an OpenMP PL model is carried out, where task scheduling, variable declarations and synchronization barrier are addressed; (2). Then, generation of OpenMP Fortran/C code from the OpenMP PL model takes place.

- ***From Deployed MARTE model to Synchronous model.*** This transformation allows to generate a synchronous equation model. Code for synchronous languages: either Lustre, Signal or Lucid synchrone, can be generated from this model. This allows checking of functional properties of modeled applications.

- ***From Deployed MARTE model to RTL model.*** This transformation enables the generation of RTL models from deployed MARTE models. The RTL model takes only modeled application at the MARTE level and permits generation of VHDL code, with the intended goal of synthesis onto FPGAs. This dissertation is mainly concerned with the evolution of this transformation chain.

It should be noted that the different transformation chains: simulation, synthesis, verification etc., are currently unidirectional in nature.

4.4 Related works in SoC Co-Design

This section covers some related works that take into account the aspects related to SoC Co-Design, and propose respective design methodologies. A large number of these researches focus on elevation of design abstraction levels in order to reduce design complexity. While it is not possible to give a detailed description of all environments and tools focusing on SoC Co-Design and particularly MDE, we try to give some significant contributions.

Works carried out at the NEC research lab at University of Lugano were among the first to propose use of UML for SoC Co-Design, using a methodology termed as ACES [140]. The MOPCOM project [3, 127] inspires from MDE fundamentals for the specification and development of SoCs, and uses the IBM Harmony[4] process coupled with Rhapsody[5] UML modeling tool. In the same manner, Sara Bocchio et al., working in the *Advanced System Technology* department of STMicroelectronics equally proposed a SoC conception methodology, termed as *UPSoC* [212]. This methodology largely exploits The UML for System C profile [211], as well as the tools that they have specialized for their utilization.

Similarly SPEEDS! (Speculative and Exploratory Design in Systems Engineering) [232] is another European project for embedded systems development based on SysML and AUTOSAR. While the EPICURE project [129] defines a design methodology in order to bridge the gap between high abstract specifications and heterogeneous reconfigurable architectures. The framework is based on Esterel design technologies and provides verification and synthesis capabilities. However, one of the existing drawbacks of this framework is the lack of available support for a high abstraction level design methodology, in order to reduce design complexity.

In [135], Klaus Kornlof and Ian Loiver, from Nokia, propose means to use executable use cases for the specification of SoCs. Furthermore, in their approach, the textual specifications are not completely excluded, but are expressed with the help of a specialized tool like DOORS[6]. Initiatives such as presented in [210], search to exploit a structural and behavioral modeling with SysML for the generation of executable functional models. The MCSE methodology implemented by the CoFluent Studio tool provides an architectural analysis in a SoC Co-Design flow [54]. This tool permits rapid specification of annotated functional models along with models of execution platforms; and enables analytical evaluation followed by a dynamic one, by simulating the results of a specific choice of projection between the two models. The language used for these descriptions closely resembles UML concepts. The MCSE approach has also been adapted by Nokia in extending the approach proposed in [135]. Works are currently underway in the MARTES project [154], for an effective interoperability between the two languages.

SynDEx which is a system level CAD tool based on the *Algorithm Architecture Adequation* (AAA) methodology [101, 142], is also based on Y schema. This tool, developed by the initiative of INRIA, permits to project a model of functional specifications onto a model of target platform. SynDEx also integrates heuristics for an optimum placement of the functional model on the given execution platform, under the given design constraints. Works are currently undergoing

[4] http://www-01.ibm.com/software/rational/services/harmony/
[5] http://www-01.ibm.com/software/awdtools/rhapsody/
[6] http://www-01.ibm.com/software/awdtools/doors/productline/

in order for SynDEx to take MARTE profile diagrams as input for automatic code generation. In [86], a SynDEx based design flow is presented to manage SoC reconfigurability via implementation in FPGAs, with the application and architecture parts modeled as components. In [239, 240], the authors have contributed in the development of the HRM package of the MARTE profile, and describe hardware platforms with parameterizable characteristics, enabling rapid development of a simulator, before validating their works by execution of an application on the modeled hardware.

While on the opposite side of the spectrum, The ACCORD/UML project[7] uses MDA initiative for the conception of embedded real-time software. The platform also provides several tools in order to help designers for the development of these complex applications. Similarly, in [216], the authors provided an initial UML based methodology from initial requirement specifications to final generation of executable embedded code on a virtual machine, that can be adapted for different types of real-time embedded kernels.

As described earlier in the dissertation, the AADL standard permits design of safety critical real-time systems. AADL models integrate functional and non-functional properties, enabling early analysis of the modeled systems to eventual code generation for the target hardware platforms. In [226], the authors use the *Open Source AADL Environment Tool*[8] (OSTATE), for estimating the power consumption levels at early SoC design phases. Power estimations have been provided for a wide range of computing units, i.e., from GPPs, DSPs to complex microprocessors present in current FPGAs. Additionally, the future extensions of the works aim for a merged AADL/MARTE design methodology.

ROSES [253] is an environment for Multiprocessor SoC (MPSoC) design and specification, however it does not conform to MDE concepts and as compared to our framework; starts from a low level description equivalent to our deployment level. In [15], a simulink based graphical HW/SW Co-Design approach for MPSoCs is proposed, but does not integrate an MDE methodology. In contrast, [92] uses the MDE approach for the design of a Software-Defined Radio (SDR), but they do not utilize the MARTE profile as proposed by OMG and use only pure UML specifications. In [218], the authors propose a UML based design flow to implement partial dynamically reconfigurable architectures. Their abstract models, consisting of an FPGA and a reconfigurable OS were modeled with pure UML specifications. The authors verified the functionality of the OS by simulation in SystemC. However, the design methodology still lacked code generation and synthesis stages. In [166], the authors provide a mixed modeling approach based on SysML and the MARTE profiles to address design space exploration strategies. However, they only provide implementation results by means of mathematical expressions and no actual experimental results have been illustrated.

MILAN [217] is another project for SoC Co-Design benefiting from the MDE concepts but is not compliant with the MARTE profile. The OMEGA European project [245] is also dedicated to the development of critical real-time systems. However it uses pure UML specifications for the system modeling and proposes a UML profile [138], which is a subset of the UML profile for Scheduling, Performance and Time (SPT). It is used with UML graphical editors and tools compatible with the XMI exchange format, such as Rational Rose and IBM's Rhapsody. Other environments such as GRACE++ [99], Metropolis [235], Ptolemy [40] and Artemis [195] are also environments using model based approaches. They offer description semantics respecting a reference metamodel; and offer tools for analysis, simulation or synthesis. The difference between all these above mentioned environments are mainly: the chosen high specification level language, model of computation, targeted application domains and the abstraction levels of the generated code.

4.5 Reconfigurability features in Gaspard2

As illustrated in chapter 1, the evolution of SoC is continuing at a rapid pace, with the result being the escalation of the design complexity at an exponential rate. Hence reconfigurability and integration of dynamic features are becoming crucial to be adapted in the SoC industry.

[7] http://www-list.cea.fr/labos/fr/LLSP/accord_uml/AccordUML_presentation.htm
[8] The SAE AADL Standard Info Site: http://www.aadl.info

Reconfigurable SoCs and especially partial dynamically reconfigurable SoCs are the future as they offer increased flexibility in terms of functionality, as well as resource economization in terms of target platforms. These systems provide advantages such as low energy consumption, fault tolerance and virtualization of the available hardware resources. As Gaspard2 framework is mainly oriented towards the design and development of SoCs, it is thus evident that these features should be addressed and integrated in our framework.

For this purpose, in [139], an initial proposal was presented for expressing dynamic features in Gaspard2. This proposal was mainly in the form of a hypothesis and no concrete semantics or implementation were carried out. The limitations of this research were addressed in [104], in order to refine the design methodology. The authors focused on expressing dynamic features at a high abstraction level using the UML metamodel. They made use of UML state machines and collaborations, for expressing dynamic aspects in a SoC Co-Design framework. In spite of their intended goal, the introduced features are not generic in nature and can only be applied onto the application modeling level in Gaspard2. Additionally, fusion of the above mentioned concepts in the MARTE profile was not undertaken. Furthermore, the works suffer from the same drawback as the earlier research, as no actual model transformations were developed for translating the high level models for eventual mapping onto an actual execution platform. Details related to these works are provided in the following chapter.

Henceforth, currently the Gaspard2 framework does not offer any concrete semantics: from model specifications to automatic code generation, in the context of adaptive SoCs. This dissertation takes these issues into account, and provides some answers to address these challenges. The following chapters inspire to provide a complete methodology for expressing reconfigurability features in a SoC design, from high abstraction levels to eventual code generation. Afterwards the generated code can be used in commercial FPGA tools for the construction of partial dynamically reconfigurable FPGA based SoCs.

4.6 Conclusions

This chapter focuses on Gaspard2, which is a MARTE compliant MDE oriented SoC Co-Design framework. Gaspard2 benefits from several advantages provided by MDE: modeling at different levels of abstraction reduces the complexity in the modeling and model transformations through the intermediate-level models. Additionally, modeling in a uniform language such as UML helps in the integration of heterogeneous systems, technologies, etc. Being based on the MARTE UML profile, Gaspard2 models are easily comprehensible by different designers. Similarly usage of components and the deployment level in Gaspard2 introduces the notion of re-usability, which helps to build complex systems at a reduced cost.

However, Gaspard2 still lacks a complete methodology for integrating aspects of reconfiguration, and especially run time reconfiguration/partial dynamic reconfiguration. As Gaspard2 focuses on SoC Co-Design, and these architectures are moving towards the horizon of reconfigurability, this is a serious drawback for the existing framework. This issue has been addressed in the subsequent chapters.

Part II

Integration into an MDE based SoC Co-Design Framework

Chapter 5

Methodology for global contribution

This chapter presents our design methodology for targeting dynamically reconfigurable SoCs using high abstraction levels. We first recall the motivations behind the dissertation followed by an explanation of our proposed approach; and then provide an overview of the design methodology developed during the course of this dissertation.

5.1 Motivations

As seen in chapter 1, SoC Co-Design complexity is increasing rapidly, necessitating the need to find effective design methodologies that take into account different aspects such as time-to-market, fabrication costs, etc.; while reducing development time. Reconfigurability features in these modern SoCs increase their flexibility to cope with rapidly evolving environments and change in user requirements, at the cost of increased complexity. While different solutions and methodologies are proposed to address these challenges; a component based mechanism seems prerequisite as it enables to view the systems in a precise composed manner, illustrating the design hierarchy and offering separation of concerns, as emphasized in chapter 2. Different components in a system can be designed independently and afterwards composed together to form a final assembly.

Migration of a component based design to high abstraction levels offers additional benefits: a high level model is independent from implementation details and is re-usable and maintainable as highlighted in chapter 3. A SoC Co-Design framework integrating all these aspects while incorporating IP re-use seems a promising approach, as it offers designers to specify their systems (applications, architectures or their allocations) in a graphical manner which increases system comprehensibility. In chapter 4, the Gaspard2 framework is introduced which offers these benefits. However, dynamic reconfiguration is still absent in the framework, necessitating the need to introduce additional semantics. This chapter presents our design methodology to address this limitation, which can be viewed as a global view of our contributions. Once the methodology has been introduced, we move onto addressing the various concepts and implementation details which make it possible to design and implement dynamically reconfigurable SoCs from Gaspard2 environment.

5.2 Proposed approach

Adaptivity and *reconfigurability* are significant issues for the design and implementation of future SoCs which must be able to cope with end user environment and requirements. These SoCs must have robust effective and efficient mechanisms to deal with dynamic characteristics. Normally, the reconfiguration is carried out in these complex systems by means of a *control mechanism* (for example a RTOS or middleware, or a reconfiguration controller as specified in section 1.3.4). This control mechanism is mostly mode based in nature and can depend on QoS choices: such as changes in executing functionalities due to designer requirements, or changes due to resource constraints of targeted hardware/platforms. The changes can also take place to due to other environmental criteria such as communication quality, time and area consumed for reconfiguration; and energy consumption levels.

For a SoC Co-Design framework, this control model must be generic enough to be applied to both software and hardware design aspects. Similarly for a framework incorporating high abstraction levels and specially MDE, the control model must respect all the related criteria. While several control models exist, automata based control [151] are promising as they incorporate aspects of modularity present in component based approaches, for describing SoC in an incremental fashion to build these complex systems.

5.2.1 Inspirations for our design flow

In this dissertation we present a generic control semantics for expressing reconfigurability in SoCs. The introduced control semantics are introduced via a high abstraction level modeling approach, in MARTE profile (and its corresponding metamodel) and subsequently in the Gaspard2 framework. Our design flow is inspired from the works present in [104], which as explained in the last chapter, provided an initial design methodology for expressing dynamic aspects in Gaspard2. Equally, the design methodology also incorporates aspects introduced in [143], where a Gaspard2 application modeled with UML specifications was intended to be converted into a hardware functionality (while keeping its characteristics intact, such as multidimensional arrays and repetitions that were specified at the modeling level). However this design flow was static in nature and did not take dynamic reconfigurability into account. The intention of the design flow was to create a hardware accelerator for final implementation on a target FPGA. However, this approach had several drawbacks.

Firstly the design methodology did not take the MARTE profile into account at the high modeling level, which could have enabled to conform to the soon to be industry standard for modeling/design of SoCs. Secondly the final hardware design (the hardware accelerator) was intended to be implemented in an FPGA as a black box, having a fixed rigid nature. Moreover there is no notion of an heterogeneous system (processors, buses, etc.) in the final implementation that communicates with this hardware accelerator. As mentioned earlier in section 1.1, modern SoCs are becoming more and more heterogeneous in nature. An additional repercussion is that specification of dynamic reconfiguration semantics is not present at the high modeling levels. Finally the model transformations associated with this methodology were not able to completely produce the HDL code for the eventual intended implementation.

5.2.2 Modeling aspects of reconfiguration controller

As seen in section 1.3.4 related to partial dynamic reconfiguration in FPGA based SoCs, a reconfiguration controller is mandatory for dynamic reconfiguration in an FPGA. The controller itself can be implemented in different manners. It can be external or internal; and can be a hardware module written in an HDL acting as a controller. In this case, the reconfiguration is not self reconfiguration as this simple controller is not capable to communicate with the ICAP core for implementing internal partial dynamic reconfiguration. For self reconfiguration, an embedded hard/soft core microprocessor is needed with complex reconfiguration mechanisms to not only handle switching between the partial bitstreams related to a dynamically reconfigurable region, but to also communicate with the ICAP core. In both types of controllers, the switching mechanism for swapping one configuration with another is usually in the form of a state machine. For

a microprocessor based controller, part of the code being executed relates to interfacing with the ICAP core and fetching the necessary frames or bit wide columns (depending upon the FPGA series); while another part is the state machine which basically alternates between the different implementation choices.

The first part can be viewed as an essential low level macro code which cannot be modeled at high abstraction levels; while the 2nd part corresponding to a state machine, depends upon the nature of the functionality, the number of partially reconfigurable regions, their corresponding implementations and the mechanism to change the implementations related to a dynamic region. This state machine part can be modeled at high abstraction levels via introduction of some associated control semantics.

Moreover, the reconfiguration controller can act upon the events provided by an external environment, or depends solely upon the tasks of the functionality present in the static/dynamic portions. During the course of this dissertation, the first approach has been adapted, to provide a basic design template. Equally, it is possible to divide the functionality in several portions, i.e., some parts of the application can be present in the static portion (even in the controller itself), while a part(s) can be dynamically reconfigurable [4]. However, this approach can lead to an increased complex design due to synchronization between the different static/reconfigurable regions. For this reason, in the dissertation, the desired application functionality is treated as a single reconfigurable region.

5.2.3 An application-driven approach

As it has been made evident in section 3.3, due to limitations in the MARTE profile, currently it is not possible to completely specify the details related to dynamically reconfigurable FPGAs. Moreover, due to limitations of the current tools for implementing partial reconfiguration, it is difficult to bridge the gap between specifications of an FPGA modeled with the MARTE profile, and eventual synthesis and implementation. While efforts have been made in research such as in [3, 127], the dynamic reconfiguration is still manipulated manually. Moreover, dynamic reconfiguration should not be dependent on the underlying architectural details, but should be application driven as explained in the introductory chapter. .

Thus, during this dissertation, only the application model (as present in Gaspard2) is taken into account. It is specified at the high abstraction level with the MARTE profile, followed by its respective deployment along with integration of certain control aspects. The goal is to specify part of the partial dynamically reconfigurable system at high abstraction levels: notably the dynamically reconfigurable region and the part of the reconfiguration controller related to a configuration switch. The dynamically reconfigurable region refers to the modeled application, which is successfully converted into a hardware functionality for eventual synthesis. While the controller part of the system refers to a high level control semantics which is introduced subsequently in the next chapter.

Normally in PDR based systems that focus on changing the context of the application (we avoid the discussion related to PDR based NoCs, where the architecture itself can be flexible), we see a trend to change either the *tasks* of an application or the application itself. Additionally, as seen in chapter 4, Gaspard2 offers a deployment level which defines the relationship between elementary components and their IPs. In our design flow, we focus mainly on changing the IPs related to the elementary components of an application. This offers two advantages. An application can retain the same *structure* and the same *functionality* while differing partly in the manner by which it is *implemented*. This implementation choice can arise due to several factors such as the available hardware resources, power consumption, reconfiguration time etc. The other advantage is that by changing the elementary components, it is possible to partly change the functionality of the application

Afterwards, via the model transformations, the code related to the application functionality and the controller can be generated automatically. The state machine code is merged with the macro code to produce the complete source code for execution in the reconfiguration controller. Finally, using commercial tools, a complete PDR system can be developed.

5.3 Our contributions in the Gaspard2 environment

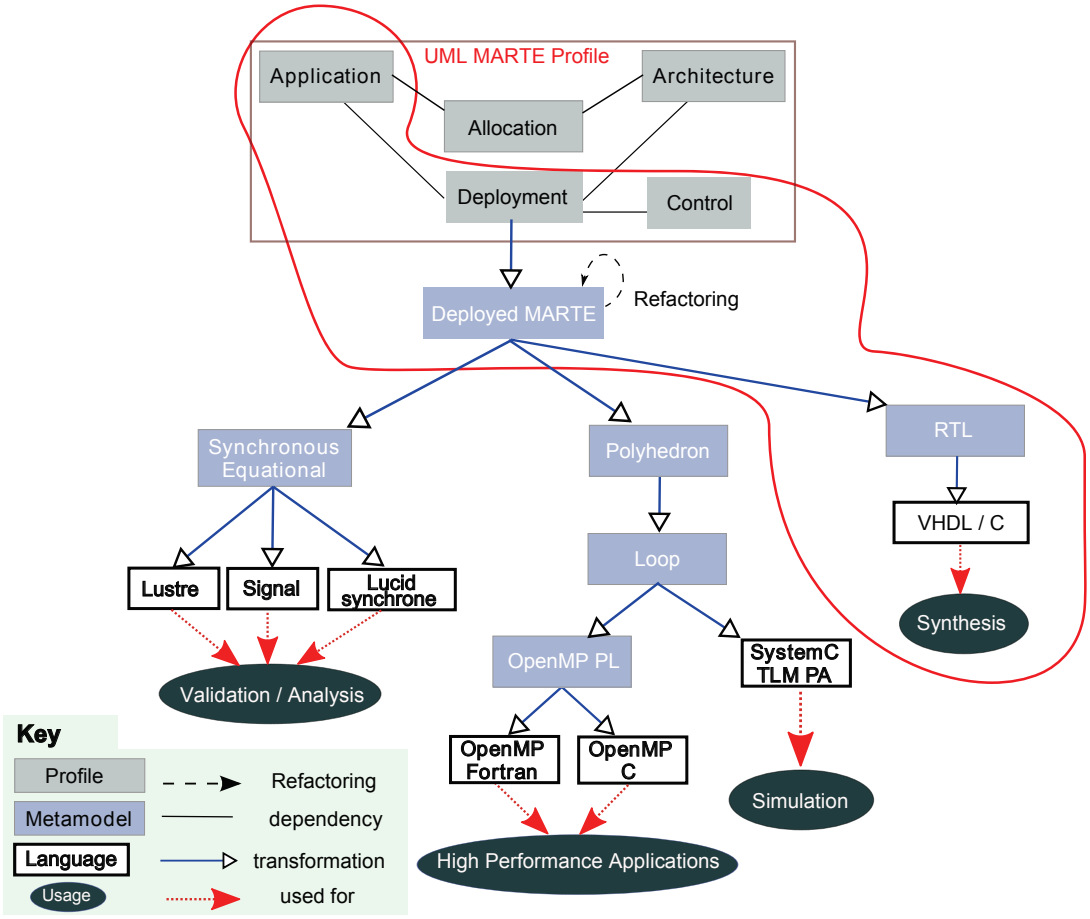


Figure 5.1: The Gaspard2 framework with integrated dynamic aspects

Figure 5.1 illustrates the contribution of this thesis with respect to the Gaspard2 framework. Initially an application can be modeled at a high abstraction level, and afterwards its elementary components can be deployed to respective IPs. A control model has been integrated at the deployment level which introduces QoS aspects in relation to the available IPs. This model in collaboration with deployment, allows to create multiple implementations or *modes* of the modeled application, each with different QoS criteria such as percentage of utilized FPGA resources, etc. The modes of the modeled application depend upon designer requirements and environment specifications. Afterwards, the application is transformed into a hardware functionality (specifically a hardware accelerator having multiple implementations) by the intermediate RTL level, while the control model is transformed into suitable state machine semantics for implementation in a hard/soft core processor acting as a reconfigurable controller in a PDR system. The choice of utilizing a microprocessor based controller over a simple HDL controller has been made to implement self reconfiguration. However, our methodology can also be extended to generate the control code for implementation in a hardware module (for example a VHDL module) acting as a controller. Thus the semantics are generic in nature, until the code generation phase; and depend upon the designer requirements and the final implementation choice of a controller.

5.3.1 The transformation chain

Figure 5.2 shows the global overview of our design flow with respect to the MDE based principles. Initially the application is modeled and deployed, along with the associated control aspects in the Gaspard2 environment with Papyrus; conforming to the UML MARTE profile that has been extended during the course of this dissertation. This modeling is independent from any implementation details until the deployment phase. Afterwards the *UML2MARTE* model transformations enables conversion of the UML model into a *DeployedMARTE* model which conforms to the Deployed MARTE metamodel as specified in chapter 6. Thereafter, the *MARTE2RTL* transformations convert this model into an *RTL* model, which itself correspond to its own metamodel. The RTL model is considered as a low abstraction level with details nearly corresponding to RTL; it provides details related to the hardware accelerators and the control features which can be used for eventual code generation. Finally using the model-to-text *RTL2Code* transformation, we generate the code related to different implementations of a hardware accelerator and the state machine for the reconfiguration controller.

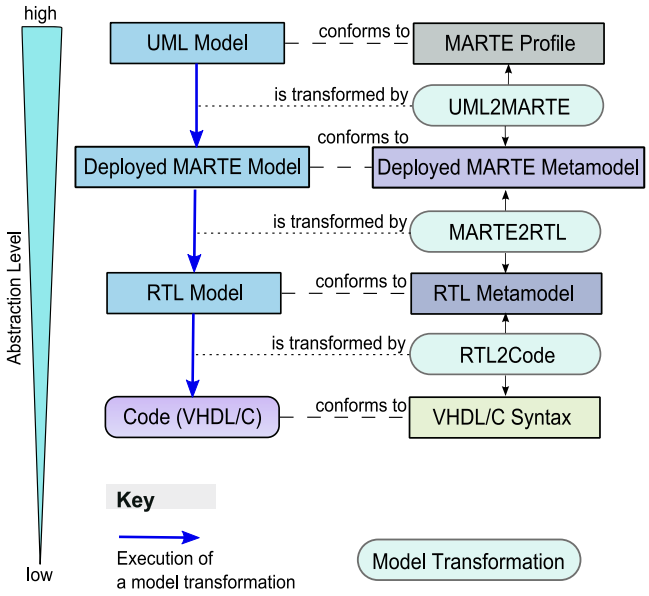


Figure 5.2: An abstract overview of our design flow

Once the source code for the application and the reconfigurable controller (state machine part) is obtained, the reconfigurable FPGA based SoC can be created by using usual Xilinx design tools related to partial dynamic reconfiguration such as Xilinx ISE[1], EDK[2] and PlanAhead [169]. These implementation details have been specified in detail in chapter 9. Depending upon the various deployment choices, the converted form of the modeled application (hardware functionality) is treated as a PRR (Partial Reconfigurable Region) with several PRMs (Partial Reconfigurable Modules), which are unique implementations for a dynamically reconfigurable region. Our design methodology allows to create several PRMs for a PRR on the basis of elementary components and their associated IPs. Each PRM has the same shape and interface respecting the partial reconfiguration semantics. The code generated from the control concepts is used directly in the reconfiguration controller for switching between the PRMs.

Our aim is not to replace the commercial FPGA tools but to aid them in the conception of a system. While tools like ISE and PlanAhead are capable of estimating the configurable FPGA resources (CLBs and in turns the slices) required for implementing the hardware design, this resource estimation is only possible after initial synthesis. In our design flow, the elementary components can be synthesized independently to calculate the consumed FPGA resources. This information can be then incorporated into the model transformations, making it possible to

[1] http://www.xilinx.com/ise/logic_design_prod/foundation.htm
[2] http://www.xilinx.com/ise/embedded_design_prod/platform_studio.htm

calculate the approximate number of consumed FPGA resources of the overall application (at the RTL model) before final code generation and eventual synthesis. Thus the designer is able to compare the resources consumed by the modeled application and the total resources available on the targeted FPGA resulting in an effective *Design Space Exploration* (DSE) strategy. If the application is too big to be placed on the FPGA, the designer can carry out a *refactoring* of the application, which has been detailed in [75, 98, 231]. A refactoring usually involves changing the execution semantics related to the application, such as converting a sequential execution into a parallel one or vice versa. It should be noted that a refactored Gaspard2 application remains a Gaspard2 application and there is no change in its global functionality.

5.4 Limitations of our approach

While this thesis responds to several questions related to the introduction and development of dynamically reconfigurable SoCs from high abstraction levels, certain limitations do exist related to our contributions. One of the initial drawbacks is the absence of architectural aspects at the MARTE profile level, which could help in the conception of a complete SoC. Additionally, MPSoC systems cannot be currently targeted due to this reason. Moreover, while we have successfully presented an application oriented approach, issues such as control/data flow synchronization and dataflow conservation during recnfiguration have not been fully treated. These limitations are due to the underlying model of computation of Gaspard2, as indicated in section 6.2.5.2. Finally, resolution of some of these problems have been presented in the perspectives related to this thesis.

5.5 Conclusion

This chapter represented the global contributions that have been carried out during the course of this dissertation. The contributions relate to creating a complete high abstraction level methodology for defining dynamic aspects in a SoC framework. After presenting our design methodology, we now move onto the different contributions which help to integrate dynamic reconfiguration in Gaspard2. These contributions are specified in the subsequent chapters. We first present the control model that provides the underlying semantics for managing dynamic reconfiguration in Gaspard2, in the next chapter.

Chapter 6

Expressing adaptivity for SoC with MDE

As explained before in chapter 4, Gaspard2 targets data intensive processing (DIP) applications. In reality, the applications targeted in Gaspard2 are mainly data flow oriented and there is no concrete notion of control. This is because the core formalism of data parallelism in Gaspard2 (its MoC or Array-OL) is based on *data dependency* descriptions and repetition operators; for expressing data parallel applications that compute large amounts of data in a *regular* manner. The behavior of these applications is completely fixed statically and cannot be changed at run-time. Thus dynamic behavior is considered to be disadvantageous to the regularity and performance of targeted Gaspard2 applications.

However, as the field of reconfigurable computing is gaining a foothold in the SoC industry at a rapidly increasing pace, dynamic behavior begins to appear more and more in these complex systems; and is vital to resolving issues such as related to resource economization, QoS etc. For integrating dynamic characteristics in a target platform, a control mechanism is integral, as it provides system designers with increased flexibility in terms of design configurations, system performance, consumed platform resources, energy consumption, etc. These design configurations are similar in context to the definition given in section 2.3.

While the MARTE profile permits modeling of behavioral aspects using state machines, activity and sequence diagrams, it is up to the underlying framework to provide proper semantics to translate these high level models. However, unfortunately, Gaspard2 does not provide adequate semantics to model and specify dynamic behavior for the targeted platforms.

In turn, current systems modeled in Gaspard2 are rigid: they are too specific and static. Even a small replacement or modification of a functionality in the system leads to system reconstruction. Thus suitable behavioral modeling concepts are required to address these issues.

A generic control semantics is thus necessary, which can be seamlessly integrated into a SoC Co-Design framework, having compatible semantics for specifying the hardware and software aspects of the framework. Not only these concepts should be effective and robust to remove design faults and errors, but model transformations should be present to generate eventual code for final implementation.

This chapter first presents a generic control semantics to address the above mentioned challenges. Initially, we provide some basic conditions and abstract concepts related to control modeling at a high abstraction level. Afterwards, we investigate the advantages/disadvantages of integrating the semantics at different levels of a SoC Co-Design framework. The investigation results in the integration of the semantics at the deployment level in Gaspard2, which is also presented in this chapter, along with the added extensions carried out during this dissertation. The novel control semantics and the deployment concepts are added to the MARTE profile and its corresponding metamodel by means of a merge mechanism, explained later on in the chapter. Finally, graphical examples using the extended MARTE profile are given for a better comprehension.

6.1 Control semantics for Gaspard2

We first present some basic conditions that must be respected for the specification of a control semantics in the context of an MDE based SoC Co-Design framework.

6.1.1 Basic requirements

Behavior in systems can be specified in different forms, and at different abstraction levels, etc. In this thesis, we focus on *control-related behavior*, which will only be called behavior afterwards for the sake of simplicity. According to the context of Gaspard2, the behavior modeling and the generic control semantics should take the following aspects into account:

- ***Conformance with MARTE profile.*** The proposed control mechanism should be compatible with MARTE, i.e., the control concepts can be modeled using the MARTE profile or MARTE compatible concepts in a graphical modeling tool supporting MARTE.
- ***Complete design flow.*** Only abstract modeling concepts are not sufficient. Concrete modeling concepts at either the MARTE profile or its metamodel must be developed. Similarly, model transformation chains must be developed that interpret these high level control concepts and generate the required code.
- ***Precise semantics.*** The control needs to consider *conciseness* and *clarity*, which will benefit, on one hand the documentation and communication, and on the other hand, the model transformations that enable the implementation level code generation.
- ***State-based control.*** A state-based control mechanism, e.g., state machines, is preferable in Gaspard2, not only because of its wide adoption in academia and industry, but its verifiability supported by large quantity of formal verification tools. Additional extensions have been proposed based on the mode automata in [152].
- ***Control flow representation.*** As all the data in Gaspard2 are modeled in the form of arrays, control events in Gaspard2 should also follow the same semantics, i.e., they should be modeled as arrays. For example, an infinite flow of control events can be modeled as an infinite array having a special shape of $\{*\}$. The value of $\{*\}$ is used to denote an

infinite dimension in the RSM package in MARTE. However, Gaspard2 specifications do not take environment or platform aspects into account. Hence, control event arrays are also modeled independently from environment constraints in the same way as data, e.g., what they are and when and how they are presented. Control event arrays, similar to data arrays, are supposed to construct and exhibit *data dependencies* between tasks.

- ***Data/control separation.*** In [139], a separation between control/data flow has been presented. Extensions of this control separation have been presented in [95, 96, 104, 116].

While Gaspard2 respects Array-OL semantics, the proposed control extension does not aim at full compliance, due to Array-OL's nature as a pure data parallelism specification language. The generic control semantics endows Gaspard2 with dynamic features. This permits Gaspard2 to become far more effective, as SoC applications and hardware functionalities are expected to be increasingly flexible and adaptive. Similarly, for reasons of clarity, this control mechanism provides precise semantics, so that the control can be easily understood.

6.1.2 Related works

We now provide some related works related to control and its specification. In several tools and development environments dedicated to mixed dataflow/control-flow applications, a *multi-paradigm* approach has been proposed to integrate languages in different styles, i.e., dataflow and some imperative features:

- SIMULINK and STATEFLOW[1]: the former is used for the specification of block diagrams where some operators of the latter are used to specify controllability.
- SCADE[2]: in the SCADE environment, state machines are embedded and used to activate dataflow processing specified in Lustre [105].
- PTOLEMY II: state machines can also be mixed with dataflow equations [40].

In [141, 221], the authors concentrate on control based modeling and 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. A similar approach has been presented in [87]. However, control methodologies vary in nature as they can be expressed via different forms such as petri nets [18], or other formalisms such as mode automata [151, 152].

Mode automata extend synchronous dataflow languages with an imperative style, but without many modifications of language style and structure. They are a simplified version of Statecharts [107] in syntax, which have been adopted as a specification language for control oriented reactive systems. Mode automata have a clear and precise semantics, making inference of system behavior possible, and are supported by formal verification tools.

These constructs are mainly composed of *modes* and *transitions*. In an automaton, each mode has the same interface, and equations are specified in the modes. Transitions are associated with conditions, which serve to act as triggers. Mode automata can be composed together in parallel; and enable formal validation by using the synchronous technology. Among existing UML based approaches permitting design verification, we find examples in literature such the Omega project [100] and Diplodocus [14]. These approaches essentially utilize model checking and theorem proving.

In mode automata, computations in each state share the same basic clock with events (or condition expressions) that fire the transitions. As a result, there is the coherence between the computations and events with regard to their clocks. Unlike mode automata, in execution platforms, the independence between control and computation makes it possible to have incompatible correspondence. The latter leads to an unsafe design. Mechanisms must be developed to address this difficult challenge. Similarly in [139, 172], a multigranularity approach has been proposed for synchronizing control/data flow. However, this introduces additional deficiencies

[1] http://www.mathworks.com/products/simulink
[2] http://www.esterel-technologies.com

as the arrival of control events is non-deterministic in general, and thus not easily visible to the SoC designer.

An initial hypothesis of control for expressing dynamic behavior in Gaspard2 at the application level has been proposed in [139, 172]. The control is state-based, as it is inspired from the mode automata. However unlike mode automata, the control and data computations are specified in a separate manner, allowing a clear distinction between the two. As a result, data computations can be specified independently from control. However this approach has several drawbacks. Parallel and hierarchical compositions of automata have not been defined, these compositions are considered as basic mechanisms that enable specification of complex reactive control systems in Statecharts [107]. Similarly, synchronization between control and data is not well defined to guarantee a safe design free from faults or errors.

Extensions to these works have been proposed in [95, 96, 104, 116] and introduce an improved control at the application modeling level in Gaspard2, with hierarchical and parallel compositions; and introduce formal semantics based on mode automata and the Array-OL language. These works allow to express the control events as arrays not dissimilar to the data arrays (An infinite flow of control events is modeled as an infinite array) which also have data dependencies. The main drawback of this approach is that it is only an hypothesis and non-generic in nature, and lacks actual model transformations. Similarly, this approach uses pure UML semantics and does not provide a bridge between UML and MARTE specifications.

6.1.3 Abstract generic control model concepts

This section first describes the basic concepts related to our generic control model[3]. Several basic control concepts, such as `Mode Switch Component` and `State Graphs` are presented first, which are abstract in nature. Afterwards, a basic composition of these concepts is illustrated, that helps in the construction of a mode automata, which is the actual intention of this generic control model. Thus the abstract modeling semantics derive from the concepts of modes 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. Finally, the control model follows a component based approach, in order to respect the semantics of mode automata, and a SoC framework based on MDE methodology.

This control semantics can be integrated into different levels (application, architecture and allocation) in a SoC Co-Design framework [66], which will be detailed later on in the chapter. Similarly, concrete details and usage about these abstract concepts are discussed afterwards in sections 6.4 and 6.5 respectively.

6.1.3.1 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 [172]. The mode switch component in Figure 6.1 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 i_d (data input) and o_d (data output) input and output ports. 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 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). All the input and outputs share the same time dimension, ensuring correct one-on-one correspondence between the inputs/outputs. 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 illustrated in Figure 6.1. Other type of ports, such as input/output ports are not necessary and are thus represented with dashed lines.

[3] Here control model refers to control semantics and not an UML model

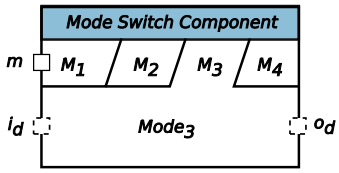


Figure 6.1: Abstract representation of a mode switch component in Gaspard2

6.1.3.2 State graphs

A state graph in Gaspard2 is similar to state charts [107], which are used to model the system behavior using a state-based approach. We term these state graphs as *Gaspard state graphs*. A state graph can be expressed as a graphical representation of transition functions as discussed in [95]. 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 can be represented in different ways, for example, with the help of state charts or via state tables. The right-hand side of Figure 6.2 illustrates a Gaspard state graph which is similar to state charts [104]. Similarly, the left-hand side of the same figure represents a state graph by an equivalent state-transition table. Formal definitions of Gaspard state graphs have been presented in [104].

As compared to mode automata and state charts, state graphs do not require initial states. In state charts or mode automata, transitions are carried out in an automatic manner, i.e., the source state of one transition is identical (or same) to the target state of previous transition. If there is no previous transition, then the initial state is taken by default. However, in state graphs, only transitions are defined between the states, and no automatic sequential order is specified. Thus, set of source states and events in the form of arrays can be provided to the state graph, in order to get respective array sets of target states and mode values. The consequence of this mechanism is that a state graph itself cannot be treated as an automaton. State graphs can also be parallely composed as viewed in Figure 6.3, or composed in a hierarchy [104].

target / source	S1	S2	S3
S1	all	e1 & e2	e3
S2	e1	all	e1 & e3
S3	e1	e2	all

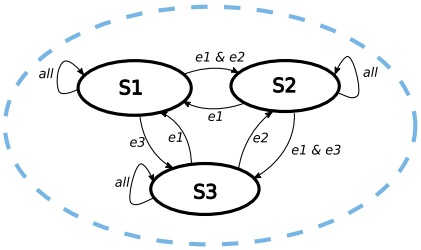


Figure 6.2: Different representations of a Gaspard State Graph

6.1.3.3 Gaspard State Graph Component

A state graph in Gaspard2 is associated with a `Gaspard State Graph Component` as shown in Figure 6.3. Thus a state graph determines the internal behavior of a Gaspard state graph component. A Gaspard state graph component or GSGC determines the mode value definition by means of its associated state graph(s). The mode values allow activation of different exclusive computations or modes in the related mode switch components. Thus, GSGCs are ideal complements of mode switch components, with mode values being the relation between the two concepts. A Gaspard state graph component can be viewed as a controller component while the mode switch component switches between the modes according to the controller.

Similar to the mode switch component, a Gaspard state graph component has its interfaces. These interfaces include event inputs from the environment, source state inputs, target state

outputs and mode outputs. Event inputs are used to trigger transitions present in the associated Gaspard state graph. The source state inputs determine the states from which the transitions take place, while target state outputs determine the destination states of the fired transitions. The mode outputs are associated with a mode switch component in order to select the correct mode for execution.

In this dissertation, we focus mainly on simple non hierarchical Gaspard state graphs associated with a GSGC. However, the control model makes it is possible for a Gaspard state graph component to have parallel or hierarchical Gaspard state graphs. These concepts have not been treated during the course of this dissertation; and can be considered as a future evolution of our design methodology.

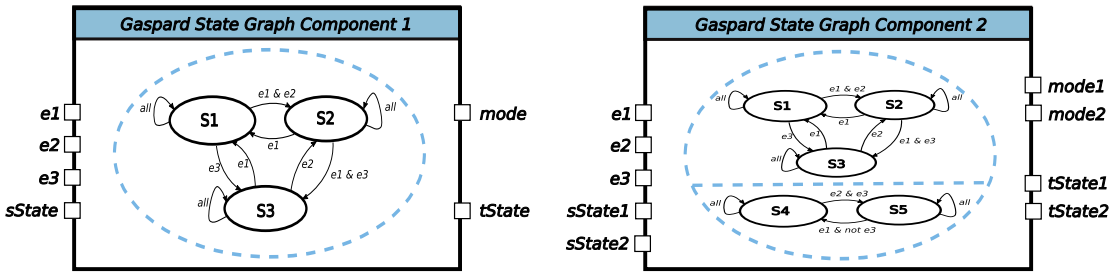


Figure 6.3: Examples of Gaspard State Graph Components

6.1.3.4 Combining modes and state graphs

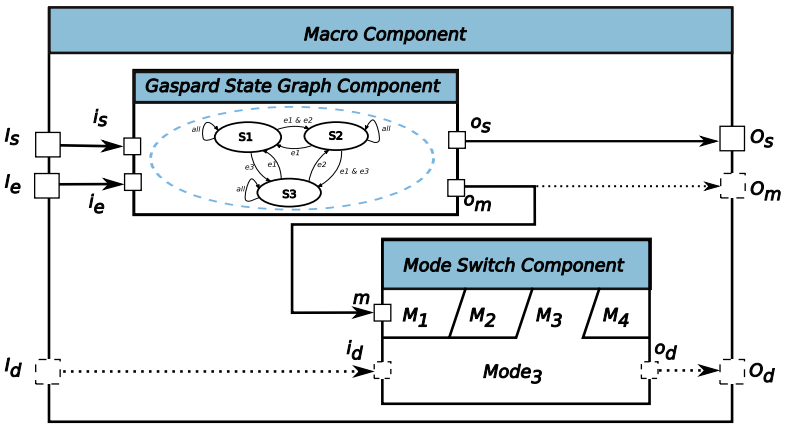


Figure 6.4: An example of a macro structure

Once mode switch components and Gaspard state graph components are introduced, a `Macro Component` can be used to compose them together. An abstract representation of the macro component in Figure 6.4 illustrates one possible composition; and represents a complete Gaspard2 control structure. In the macro, the Gaspard state graph component 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, data dependency between I_d and i_d. They are drawn with dashed lines in Figure 6.4. The illustrated figure is used as a basic composition, however, other compositions are also possible, for instance, one Gaspard state graph component can control several mode switch components [204]. In order to simplify the illustration, events `e1`, `e2` and `e3` are only shown as a single event I_e.

In [104], the authors have illustrated that hierarchical composition of state graphs using the concepts of mode switch components (MSCs) and Gaspard state graph components (GSGCs) is possible. A mode switch component can contain a nested lower level Gaspard state graph as

a mode. This hierarchical composition can be of two types: composition in a simple repetition context; or composition in an hierarchical repetition context. The first type insures that all the GSGCs at different hierarchical levels have the same transition rate, in comparison to the second type, where low level GSGCs have a faster transition rate then the high level GSGCs. As we do not treat hierarchical composition of Gaspard state graphs, this context is currently non applicable to this dissertation.

In order to be compatible with underlying specifications in Gaspard2, the control modeled as an array (section 6.2.5.2), can be mapped onto the same time dimension as data arrays. Thus this time dimension is common to all control and data arrays in the specification. However, this is not always the reality. Issues arise when Gaspard2 specifications are mapped onto an execution model, e.g., a dataflow model. These problems involve the relation and synchronization of control event and dataflow, which are always ambiguously specified. These issues arise from the specification and implementation gap of Gaspard2 dynamic behavior.

When the specifications are mapped onto a dataflow model, the control event-data relation implies that the two flows (event and data) have the same clock and thus can be synchronized. However, Gaspard2 is not associated with any specific execution model. In an execution model, a control event may be uncorrelated with the repetitions of data computation with regards to their clocks. For example, a person watching a program on television can change the channels with the help of a remote control. The user input for channel change does not depend upon or correspond with the TV data flow. This problem is caused by different specification styles of control and data computation, i.e., state-based control adapts event-driven style [107, 108] and data computation adapts dataflow style [36]. In [104], the authors choose to use classical synchronous dataflow control mechanisms to resolve these problems. However, when targeting execution platforms and technology levels such as RTL, a more elaborate mechanism is required, which will be discussed later on in the dissertation.

6.2 Control at different system design levels

The previous section described an abstract control model for integrating dynamic aspects in a system. Similarly, these control mechanisms can be integrated in different levels in a SoC Co-Design framework, with the advantage of introducing dynamic semantics in these SoCs. We first analyze the control integration at the application, architecture and allocation level in the particular case of the Gaspard2 framework, followed by a comparison of the three approaches.

6.2.1 MARTE concepts for constructing mode automata

We first present some additional MARTE concepts which aid in the modeling of mode automata. The basic concepts of Gaspard2 control have been presented in section 6.1.3, but complete 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: `interrepetition` dependency (IRD), `tiler` and `defaultLink` connectors, are used to build mode automata.

Hence, it is possible to establish mode automata from Gaspard2 control model, which requires two subsequent steps. First, the internal structure of a generic `Gaspard Mode Automata` is presented by the `Macro component` illustrated in Figure 6.4. The Gaspard state graph component 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 component and it should be placed in a repetition context.

The reasons are as follows: a macro component represents only one single transition function (one map) from a source state to a target state, where as an automata has continuous transitions which form an execution trace. In order to execute continuous transitions as present in a typical automata, the macro should be repeated to have multiple transitions. This functionality is determined by the interrepetition dependency (IRD).

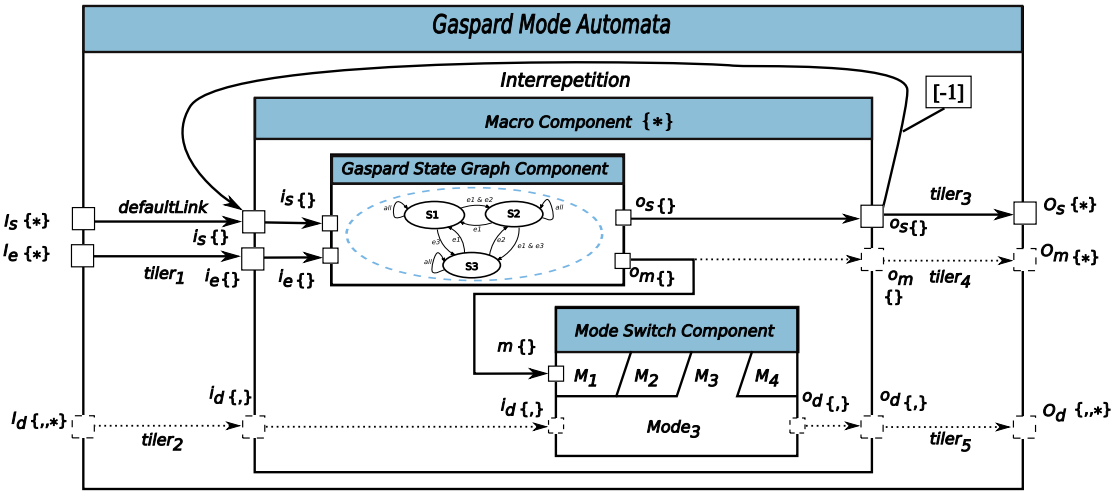


Figure 6.5: Abstract representation of a generic Gaspard2 mode automata

A vector associated to an interrepetition dependency expresses the dependencies between the repetitions inside the repetition context, i.e., the `Gaspard Mode Automata` component. Thus an interrepetition dependency serializes the repetitions and data can be conveyed between these repetitions. An IRD sends the target state of one repetition as the source state to the next repetition. This permits the construction of mode automata which can be then executed. Figure 6.5 illustrates an example of this construction.

If a dependent repetition is not defined in the repetition space, a default value is selected. The *defaultLink* provides default value for repetitions whose dependencies for the input are absent. Additionally, this concept helps to give the initial state value for the first repetition of the macro component. While in a graphical modeling approach, the initial state of a state machine can be determined by an initial pseudostate, a Gaspard state graph does not contains an initial state.

Thus this mechanism bridges the gap between a graphical representation as showed in section 6.5 and the actual semantics. It thus creates an equivalency between a state graph (having no initial state) and an automaton (having a initial defined state). Afterwards, the `tiler` connectors help in interconnecting a repetition context (Gaspard mode automata) task to the multiple repetitions of its interior repeated task.

Also, an infinite dimension is present on the input and output state, events ports of the Gaspard mode automata component to account for continuous control/data flows. Similarly the non obligatory mode output port(s) and input and output data ports also have an infinite dimension in addition to other possible dimensions. Since the macro component represents one single transition, its respective ports have shape values equal to {}, (except in the case of non-obligatory data input/output ports) accounting for one value in the dataflow at an instant of time *t*. Similarly the internal sub components of the macro also share the same shape values.

Finally, the shape value of {*} on the macro component represents its multiple (possible infinite) dimensions. The macro component is repeated in a sequential temporal dimension by means of the interrepetition dependency. It should be made clear that the sequential execution of the control model must be synchronized with the execution of the hardware accelerator which will be presented in the subsequent chapter. This synchronization procedure is explained in detail in chapter 9.

We now present the integration of the generic mode automata at different SoC Co-Design levels, starting with the application level.

6.2.2 Application level

Integration of control model and the construction of mode automata at application level is very similar to the generic Gaspard mode automata shown in Figure 6.5. Figure 6.6

represents a mode automata at the application level by illustrating an example of color effect processing module (`ColorEffectTask`) used in typical smart phones. This module is used to manage the color effects of a video clip and provides two possible options: color or monochrome/black&white modes, which are implemented by `ColorFilter` and `MonochromeFilter` respectively. These two filters are elementary tasks at the application modeling level, which should be deployed to respective IPs. The changes between these two filters are achieved by `ColorEffectSwitch` upon receiving mode values through its mode port `colorMode`. The mode values are determined by `EffectController`, whose behavior is demonstrated by its associated state graph.

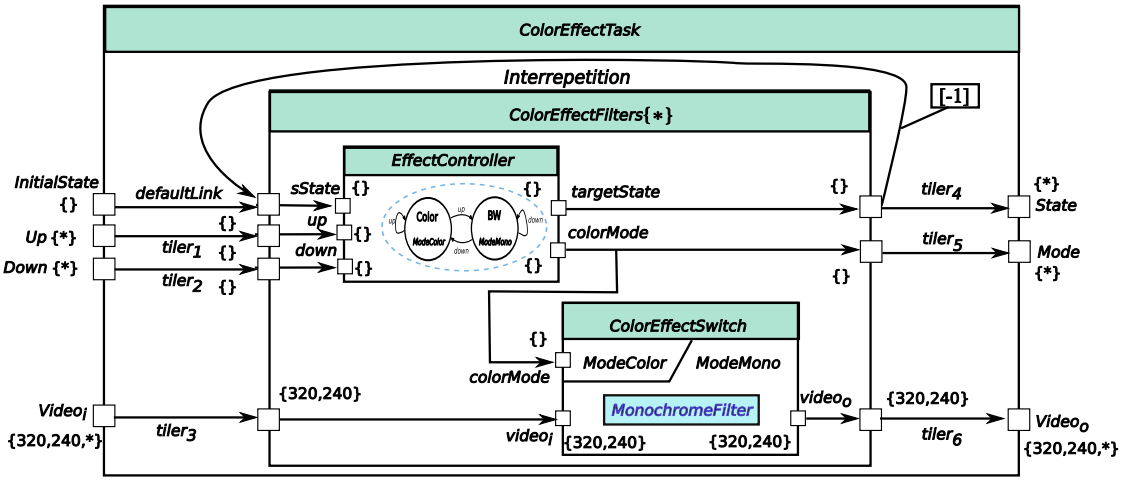


Figure 6.6: An example of color style filter in a smart phone modeled with the Gaspard2 mode automata

The `ColorEffectFilters` can be treated as a macro component; and is composed of `EffectController` and `ColorEffectSwitch` components. `ColorEffectFilters` executes the processing of one frame of the video clip, which should be repeated. In the example, `ColorEffectTask` provides the repetition context for `ColorEffectFilters`. An `interrepetition` dependency is also defined, which connects the different repetitions of the `ColorEffectFilters` component. It has an associated vector with a value of -1. Simply put, the source state of one *ColorEffectFilters* repetition relies on the target state of the previous `ColorEffectFilters` repetition. The data computations inside a mode are set in the mode switch component `ColorEffectSwitch`. The detailed formal semantics related to Gaspard mode automata can be found in [95].

The control model enables the specification of system adaptivity at the application level [104]. Each mode in the switch can have different effects with regards to environmental or platform requirements. 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.

6.2.3 Architecture level

As stated in section 4.2.4, Gaspard2 uses the Hardware Resource Modeling (or HRM) package of 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 [199].

Control semantics can also be applied on to the architectural level in Gaspard2, utilizing a similar approach as described previously for the application level. 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 either be 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 that is present inside a processing unit in the architecture for managing *Dynamic frequency scaling* [266] or *Dynamic voltage scaling* [118]. These techniques help in power conservation by reducing the frequency or the voltage of an executing processor.

6.2.4 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 [181]. Currently Gaspard2 only uses spatial allocation.

Integration of control at the allocation level can be used to decrease the number of active executing computing units in order 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 at a cost of increased power consumption levels. Modifying 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 reduces the power consumption levels along with decrease in the processing frequency. Thus allocation level allows incorporation of DSE aspects which in turn can be manipulated by the designers depending upon their chosen QoS criteria.

6.2.5 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 6.7. With respect to control integration, we are mainly concerned with several aspects such as the range of impact on other SoC design 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. Control integration at the application level has a local impact and is independent of the architecture or the allocation. 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.

Control at the allocation is local only when both the application and architecture models have been predefined 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 causes 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 results 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 also be possible to create a global controller that is re-

sponsible 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. Figure 6.7 shows an global comparison of control integration at the three mentioned levels in SoC Co-Design.

	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 6.7: Overview of control on the first three levels of a SoC framework

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. Thus, a control model at the Gaspard2 deployment level seems ideal as it is local and independent from the three SoC Co-Design levels presented previously.

Additionally, the control presented at the different levels (such as application and architecture level) is only considered with switching an application task, a hardware module or structure, and is not able to correspond to their implementations. In short, it corresponds to an existing proposal of control aspects in Array-OL [104]. Thus new concepts related to implementations must be added in Array-OL, for integrating control at the deployment level.

6.2.5.1 Existing dynamic behavior in Array-OL

The proposed control extension introduced during the course of this dissertation can be treated as a subset of Array-OL semantics. Subsequently, this subset allows introduction of dynamic features in Gaspard2, at high modeling levels, from which eventual model transformations can be carried out. We first recall certain Array-OL behavioral aspects crucial for understanding our contribution. In Array-OL, behavior of a system can be presented at different levels: namely *intra-task*, *inter task* and *application* levels, as specified in [104]. While Array-OL was initially intended for application specification, it can be used to express parallelism for hardware architectures as well, as seen in chapter 4. Thus the different levels presented subsequently also hold true for architecture modeling.

- An *intra-task* level is considered to be the finest level that makes a task dynamic. For instance, the if/then/else statements specified in a task resulting in dynamic changes during the execution of the task. However, these tasks are considered as atomic and elementary in nature. With respect to pure Array-OL semantics, the associated internal behavior of these tasks is invisible.
- An *inter-task* level, also results in an application having a dynamic behavior, via the change of tasks. Only atomic tasks are taken in consideration at this level.
- Finally at the *application* level, complete applications can be changed by switching between different applications. An application here can either refer to a global application task or a hierarchical task that can accomplish a complete functionality.

Comparing the different levels to the existing adaptivity levels in component based design as presented in section 2.3, the inter-task and application levels in Array-OL can be considered

as global system level adaptations, as they can change component assemblies or the components themselves. Where as intra-task level can be viewed as equivalent to the implementation/programming level adaptation dealing with internal component behavior.

The overall behavior of the system can also be viewed as a composition of the control at these three SoC design levels. However, in the initial control proposal for Array-OL presented in [95, 96, 104, 116]; only inter-task and application levels have been taken into account. The authors claim that the intra-task level is too fine grain in nature and does not influences the overall application specification.

However, when an Array-OL elementary task is modeled at a high abstraction level in Gaspard2 using the MARTE profile, it can be associated with respective available implementations/IPs at the deployment level. These implementations are also represented in a graphical manner. While internal behavior of these implementations may not be expressed graphically in UML due to their complex nature, properties associated with these implementations are mostly related to QoS criteria and can be expressed easily. Each of these implementations has different distinct hardware or software attributes, such as latency, consumed energy, performance throughput, etc. Replacement of an implementation associated to an elementary task by another, may result in significant changes.

For example, change in implementation of a task can occur easily, provided that the interfaces of the different implementations are compatible to each other and to the elementary task. For instance, we consider a generic *MultiplicationAddition* task having two input ports *in1* and *in2*, and an output port *out*. The elementary task may have different available implementations at the RTL level such as written in a *DSP* like fashion or an *If-then-else* construct. Both implementations express the same *functionality* related to the elementary task, albeit having different properties such as consumed FPGA resources. Changing the implementation of the task in question, may effect the overall application in terms of different QoS factors, such as performance, the total reconfigurable area consumed on a target FPGA, etc. In some cases, a change of the functionality of the task or of the application itself may be possible. This results in creation of a novel *implementation* level in Array-OL, which is the special focus of this dissertation.

6.2.5.2 Implementation level in Array-OL

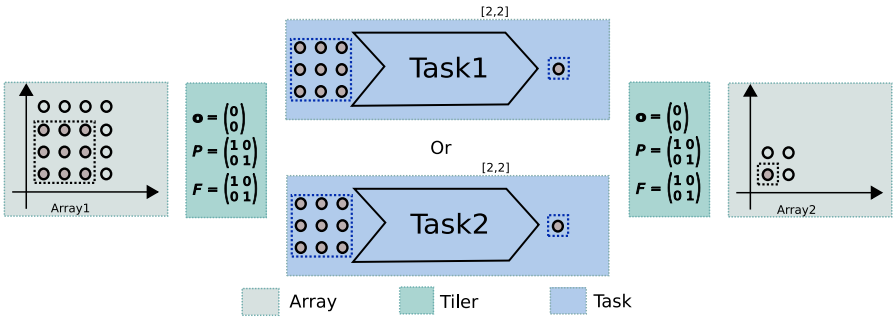


Figure 6.8: An example of task change in current Array-OL specification. A task, which is represented by one of the blue boxes, is connected to its input array (extreme left-hand box) and its output array (extreme right-hand box) through tilers. The repetition space of the task is illustrated above the task box. The patterns taken by the task as input and output are also placed aside of the task in the task box. Only the tiles used by the task repetition at [0,0] in the repetition space are surrounded by a box with dashed lines

This subsection first presents the study of the existing control notion, which is compatible with Array-OL specification model, without involvement of any execution model. The presented notion is taken in the context of Gaspard2 framework, but lacks aspects related to implementations. This control can be associated with four main types of elements: array, tiler, task and repetition space. A study regarding these four elements has been carried out in [104]. Intuitively the behavior of a repetition context task (RCT) can be achieved by changing any of these main elements or any combination of these elements.

Figure 6.8 shows task switching (inter-task level): a task can be swapped in response to some changes: such as requirements associated with the platform, application or the environment. In Figure 6.8, either *Task1* or *Task2* can be exclusively selected to be executed according to some control, which is not shown here. A prerequisite of this change is that the two tasks must have the same interface so that they are compatible with other elements. Control presented earlier in section 6.2 for different high level SoC Co-Design levels (mainly the application and architecture aspects) is mainly concerned with inter-task and application (or respectively architecture) levels.

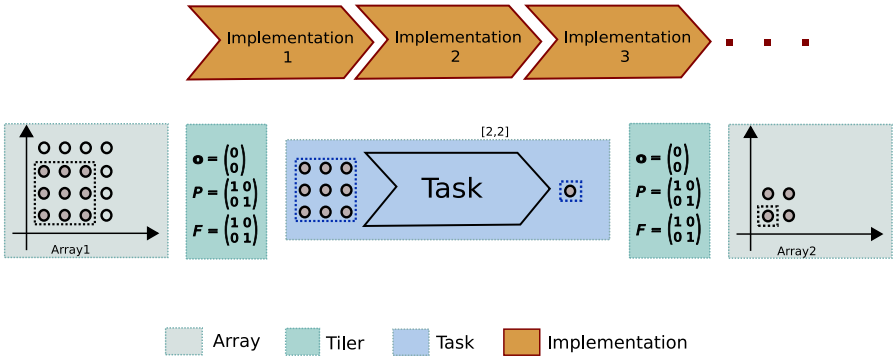


Figure 6.9: An example of the implementation level in Array-OL. The elementary task has different available implementations; and one can be replaced by another. Strict constraints enforce change of implementations only

However, the focus of this thesis is the novel *implementation* level which allows to manipulate the associated behavior of a task, in combination with the introduction of a control model at a SoC specification level dealing with component implementations. In order to be compatible with our introduced semantics, we enforce certain restrictions which allow us to respect a subset of Array-OL semantics.

In our introduced control compatible Array-OL model, we are mainly concerned with the implementations associated with an elementary component. We first introduce the notion of an implementation in Array-OL model. This notion may not be visible in pure Array-OL specifications, however when the same specifications are modeled in Gaspard2 at high modeling levels, the implementations related to an elementary task are rendered visible in terms of their interfaces and QoS properties.

Figure 6.9 shows an illustration of the newly introduced *implementation level*. A task can have an arbitrary number of available implementations as indicated by *Implementation1*, *Implementation2* and *Implementation3*. As compared to the previously mentioned control mechanism, the input/output arrays, tilers, repetition space and the concerned tasks are considered fixed in our control model. Adaptations are thus only possible with respect to the implementations.

6.2.5.3 Conditions for our control model

A Hybrid control scope level. As a dependency specification, data and control arrays need to be matched to each other, i.e., a good correspondence is required. The correspondence between control and data array indicates the control array can be regularly mapped onto the data array, which also implies the control scope of the data. As the control array can be mapped onto the data array, they can be also mapped onto the same concrete time. This mapping can be considered as an *affine-transformation* that leads to different possible control scope levels: *task level*, *repetition level* and *same task-multiple implementations* level. The first one indicates that all the repetitions of a task are changed into those of other tasks. Hence this is a complete change of the task in question. The second one, repetition-level control, enables to change the task at a finer level, enabling the switch between the repetitions of different tasks. However, the end result is an escalation in design complexity.

Finally the last level draws characteristics from the two control scope levels. Here different repetitions of the *same* task can be associated with different implementations. However, this introduces another level of complexity in the system design. We thus create a new control

scope which derives characteristics from the hitherto mentioned control levels. This control scope level is termed as the *hybrid* level: in which different repetitions of the same task are always associated with *one exclusive* implementation during execution. If an implementation associated with an elementary task is changed by the designer, then *all* the repetitions of the elementary task reference the new implementation. This is expressed mathematically as:

$$\forall R^{[n]}, \exists\, i \,\epsilon\, I^{[m]}, \forall\, k \,\epsilon\, [1 \ldots n],\ \mathit{Implement}\,(R_k) = i \tag{6.1}$$

In the above state equation, R_k represents a repetition identifier for each repetition of a repeated elementary task, having n repetitions in a RCT: $R^{[n]}$. Whereas, i represents an implementation identifier for one unique implementation related to the elementary task among a set of available implementations m, i.e. $I^{[m]}$. The $\mathit{Implement}\,(R_k)$ is a function that determines the actual implementation i for each R_k.

Control/Data flow synchronization. Also, with regards to control and data flows, only data dependency is taken into account in Gaspard2, which is a high-level specification language, i.e., no execution model or non-functional constraints are involved, hence the synchronization between control and data computation or data computation granularity with regards to control is not considered. In a low-level implementation, according to the target execution platform or technology, for example at the electronic RTL; different flows may have different clocks, particularly dataflow and control flows, which leads to the synchronization problem in the composition of these flows. Similarly for hardware circuits, different clocks having different rates may de-synchronize the circuit.

In [139], a concept of *Degree of Granularity* has been proposed to introduce synchronization between the control and data flows in Array-OL. This concept is based on the *synchronous hypothesis* and assumes a basic common clock shared by the controlling and controlled tasks. Thus they assume that control values produced by the controlling component along with data values, arrive at a controlled component at the same instant of the time period specified by the common clock. However this hypothesis is difficult to translate when these specifications are mapped onto an execution platform such as RTL.

Not only that, but in Gaspard2, the timing aspects of different dataflows is invisible at the high modeling level. Similarly, the appearances of control events are non-deterministic in general, which are not visible to the designer. Another additional drawback of the aformentioned approach is that it introduces additional hierarchies in the high level models which complicates the design specifications; and only deals with task level adaptation.

Keeping in mind all these above mentioned problems, during the course of this dissertation, the synchronization between control and data flows is considered less stringent, as its specification at the high modeling levels is currently not evident; and difficult to map onto eventual target execution platforms. Similarly Gaspard2 does not offer any middleware or RTOS to support and manage dynamic and especially partial dynamic reconfiguration of the specified models. Dataflow present before and afterwards a reconfiguration needs a mechanism which is currently not possible in Gaspard2. We thus impose certain restrictions on our implementation based control model. Firstly, control/data flow synchronization is not specified at the high modeling levels, as its introduction at these abstract levels force the designers to have detailed information about the target low level platforms. This in turn causes a high level model to become dependent on underlying platform details. This synchronization is thus treated at a targeted execution platform level. In chapter 9, we provide a RTL level synchronization approach for the control/data flow.

Secondly, the loss/disruption of dataflow during a reconfiguration is another important issue. Currently Gaspard2 does not offer any mechanisms to avoid this pitfall. During this dissertation, in chapter 9, we provide a mechanism at the RTL level that assures that the PDR system does not enter into an unsafe state during/after the reconfiguration, at the compromise of the dataflow. The mechanism while a bit limited, assures a generic approach for modeling all types of applications possible in Gaspard2. Future evolutions in Gaspard2 may make it possible to propose a high level semantics for preserving dataflow and thus will serve as extension of our design methodology.

6.3 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 implementations, and corresponds to the control notion specified in section 6.2.5.2. From now on, the term control refers to this localized form of control.

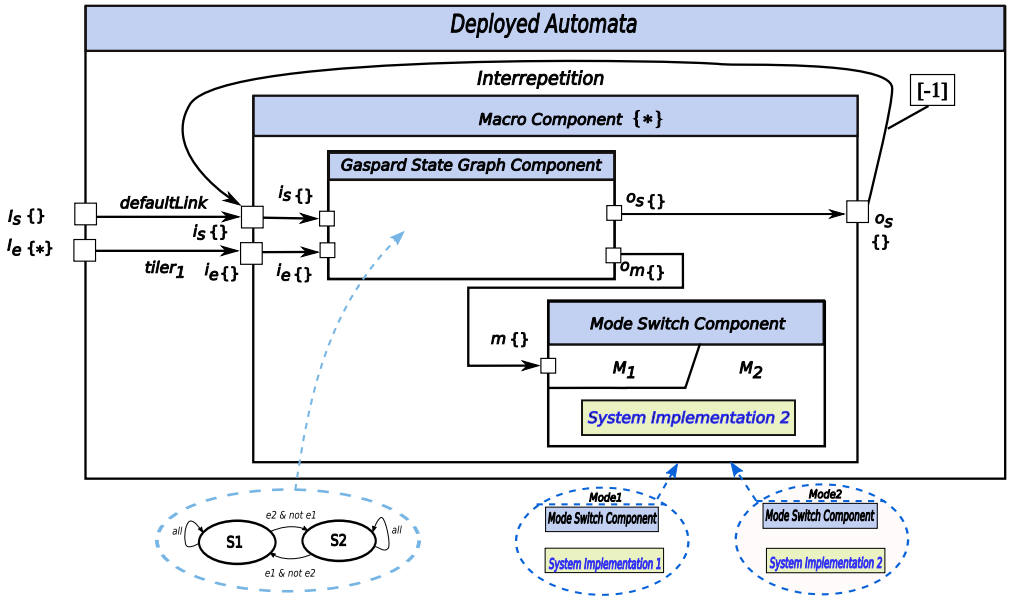


Figure 6.10: Integrating control at deployment level

Figure 6.10 shows the integration of control at the deployment level in a SoC Co-Design framework. As compared to control models at other levels which only incorporate structural design aspects, this control model deals with comportmental aspects. The deployment level automata, termed as `Deployed Mode Automata` deals with atomic elementary components and their implementations which are present at the lowest hierarchical level in the modeling; in order to address global system level implementations. As compared to other control models, a mode in a mode switch component represents a global system implementation which is a collection of different local implementations associated with their respective elementary components. Thus dataflow associated to the generic Gaspard mode automata is not explicitly expressed and input/output data ports are suppressed at all hierarchical levels in this control model. As a Gaspard state graph associated in this control model is related only to the global system implementations, this is the reason that we concern ourselves only with flat state graphs.

Also we need to address the issue related to the incoming events arriving in a deployed mode automata. In a control model at application or architecture, the events arrive either from the external environment (for example user generated stimuli) or the events are produced randomly in the application or architecture itself (due to the actions of some elementary components). However in the deployment level, the incoming events are not related to the high level modeling but are basically used to represent low level user inputs depending upon the chosen execution platform. For example at the RTL level, these user events can arrive in the form of user or environment input from a camera attached to an FPGA, or inputs received via an UART terminal. A designer modeling the system at a high level is not concerned with these low level implementation details. However, in order to make this control model as flexible as possible, and to respect the semantics of the abstract control model, event ports have been added to this proposal. During the model transformations and eventual code generation, these event ports are replaced and translated into actual event values which are used during FPGA implementation phase.

Similarly, for mode automata at application (or architecture) level, its initial state is given by an application (or architecture) component that has input event ports and an output state port.

Initially some events are generated and taken as input by that component in order to produce the initiate state. After that, this component remains inactive due to the absence of the events arriving on its input ports. This initial state is then sent to the mode automata and serves to determine the initial state of the Gaspard state graph. However, for a deployed mode automata, structural aspects are absent and only information related to elementary components is present. Thus the initial state related to the deployed Gaspard state graph cannot be determined explicitly. This limitation has been removed by introduction of new concepts in the deployment level, which help to determine the initial state of the deployed mode automata and are explained later on in the chapter. However, the proposal retains the usage of initial state port and the `defaultLink` concepts, as they help to conform to the abstract control model; and are used in subsequent model transformations for eventual code generation.

Finally, the current control at deployment is only related to creating a state machine for a reconfigurable controller. In cases of FPGAs supporting several embedded hardcore/softcore processors; it is possible to select any one for acting as a controller. However, this requires high level modeling and allocation of the reconfigurable system, currently absent in our design flow. This information can thus be passed onto the deployment phase. Currently the code generated from our design flow is explicitly linked to a generic controller, and it is up to the user to determine the nature and position of the controller. This aspect has been detailed in chapter 9.

6.3.1 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 re-used again and only the necessary IPs are modified. As we validate our methodology by implementing partial dynamically reconfigurable FPGAs, we need to clarify about the option of choosing mode automata.

Although many different approaches exist for expressing control semantics, mode automata were selected as they permit separation of 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 (modes/configurations) 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.

This concludes the portion relating to the hypothesis of our proposed control model. We now turn to actual concepts which help in the initial modeling and eventual model transformations of our control concepts.

6.4 Extending MARTE profile and metamodel

As stated before, the MARTE profile permits to model UML *Behavioral* semantics, however, in absence of an underlying framework that interprets these high level models, the modeled diagrams can only be used for specification purposes. Additionally, while the MARTE metamodel introduces the concept of *Behavior*, associated concepts that permit to concretize the exact behavior of a system are lacking. Hence the metamodel lacks the actual concepts of these specific behaviors as present in pure UML specifications, and model transformations related to a modeling framework will not be capable to interpret and transform these high level modes into actual code. It is thus up to the underlying modeling framework to extend the MARTE metamodel, in order for a designer to correctly model a specific behavior. Similarly, model transformations need to be developed that take into account the new concepts, for eventual code generation.

Moreover, a question arises about specification of the state-based behavioral modeling concepts in MARTE, as presented earlier in the chapter. Thus we turn towards the basic UML

behavioral modeling concepts such as state machines and collaborations [178] and integrate them into the existing metamodel with some modifications, in order to respect semantics of the MARTE metamodel and the model transformations.

In the extended version of the MARTE metamodel for integrating our control model, behavioral state machines are chosen to demonstrate the state-based behavior of an individual component, which acts as a controlling element in the system. State machines are the first choice because they are compatible with the state-based modeling presented previously in the chapter in section 6.1.3.2. Additionally, they are considered as key kernel for a reconfiguration controller. As in Gaspard2, control and computation are separated, the structure modeling of the controlled computation is not enough, thus collaborations are used to illustrate the behavior of the controlled components in the system.

Finally, while the MARTE profile and its metamodel enable specification of the application, architecture and allocation levels of a SoC Co-Design framework, they lack the concepts for linking the elementary components to their respective implementations. Thus the deployment concepts present in Gaspard2 need to be integrated into the MARTE metamodel and its respective profile.

This section details the above mentioned concepts added in the current version of the MARTE metamodel; their integration leading to an extended version of the MARTE metamodel by means of a *merge* mechanism. We first recall some basic concepts related to this mechanism, integral for understanding and functioning of the extended concepts.

6.4.1 Merge mechanism: extending metamodels

According to UML Infrastructure specifications [179], a *merge* mechanism is defined as *how the contents of one package are extended by the contents of another package*. A metamodel is a collection of packages, and a package can be considered a metamodel in its own right; as it itself can be further composed of sub packages.

The merge mechanism is considered to be a directed relationship between two or more metamodels (or packages); and implies a set of model transformations. It indicates that the contents of the present metamodels are to be combined. Similar to the *Generalization* concept in UML, the source element conceptually adds the characteristics of the target element to its own characteristics. The result is an element that combines the characteristics of both source and target elements. In the case where certain elements in different metamodels represent the same entity (having same name and concept), their contents are merged into a single resulting element. Details about the merge mechanism can be found in [179].

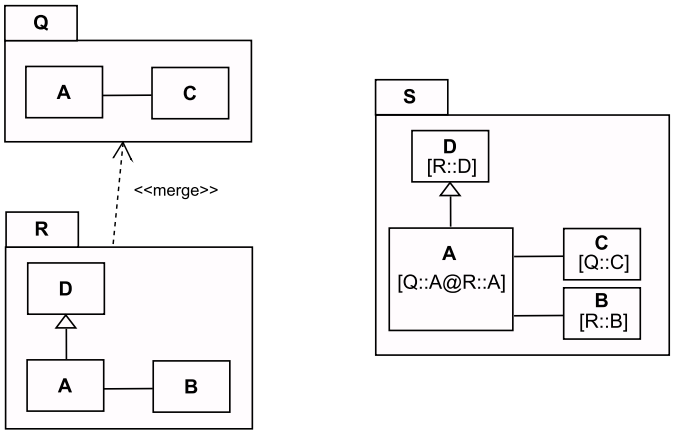


Figure 6.11: Simple example of merging (between packages) in UML

The semantics of the merge mechanism are defined by a set of constraints and transformations. The constraints specify the preconditions, while the transformations describe the semantic effects (i.e., postconditions) of the merge. If any constraints are violated, the merge is ill

formed and the resulting metamodel that contains it is invalid. This results in failure of the subsequent model transformations which take the resulting merged metamodel as input.

For a merge, the general principle is always the same: a resulting element cannot be less capable than it was prior to the merge. This means, for e.g., that the resulting multiplicity, visibility, navigability etc., of a receiving model element will not be reduced as a result of the merge. The merge mechanism is particularly useful in metamodeling and is extensively used in the definition of the UML metamodel [178].

Figure 6.11 shows the abstract representation of merging between two packages Q and R. The left-hand side of the figure shows the two packages while the right-hand side shows the result of the merge. The expressions in square brackets in the merged S package indicate which individual increments were merged to produce the final result. Here, the @ character denotes the merge operator.

In the course of this dissertation, we employ the use of a merge mechanism developed internally in our team, via the usage of model transformations. This merge mechanism is used to extend the current MARTE metamodel and enables its merging with the metamodels (or packages) detailing the concepts related to state machines and the IP deployment level as shown in Figure 6.12. Direct modifications in the MARTE metamodel itself are undesirable, as it is soon to become an industry standard and is currently being utilized by different research teams and industrial partners. Thus, direct modifications will surely make a MARTE metamodel incompatible with other researches.

In contrast, a merge mechanism increases comprehensibility between different collaborators, designers or research teams. Individual contributions are grouped together in the form of new metamodels, packages, or concepts which allow to easily distinguish between the different contributions. Similarly, conflicts arising due to merging can be easily determined and traced. Finally merging helps in the evolution of a metamodel, such as in the case of the MARTE metamodel. Modifications in either of the inputed metamodels can be carried out, provided dependencies between other input metamodels are respected and conflicts have been resolved. Now, before moving onto the different metamodels which have been introduced in the current MARTE metamodel, we briefly provide a summary of UML state machines and collaborations which partly inspire the extended MARTE meta model.

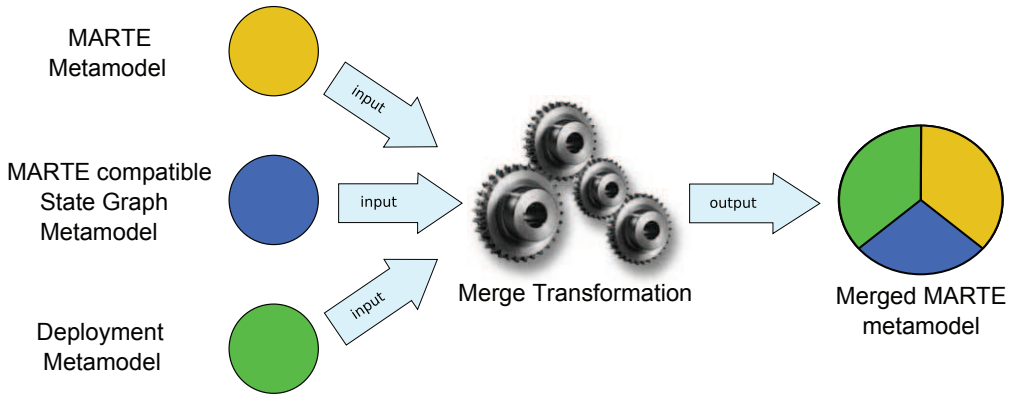


Figure 6.12: Overview of the merging mechanism in this dissertation

6.4.1.1 UML State Machines and Collaborations

UML state machines are object-based variant of *State charts* [107]. UML state machine package defines similar concepts to *State charts* that can be used for either complex discrete behavior modeling, or the expression of the usage protocol of a part of system. The former are called *behavioral state machines* and the latter *protocol state machines*, which allow the specification of a life-cycle of some objects or invocation order of its operations. *Protocol state machines* are not involved in implementations; in contrast, they enforce legal object usage scenarios.

Figure 6.13 illustrates an extract of UML state machines metamodel, that includes the main concepts of UML state machines and their relations, which are briefly presented subsequently:

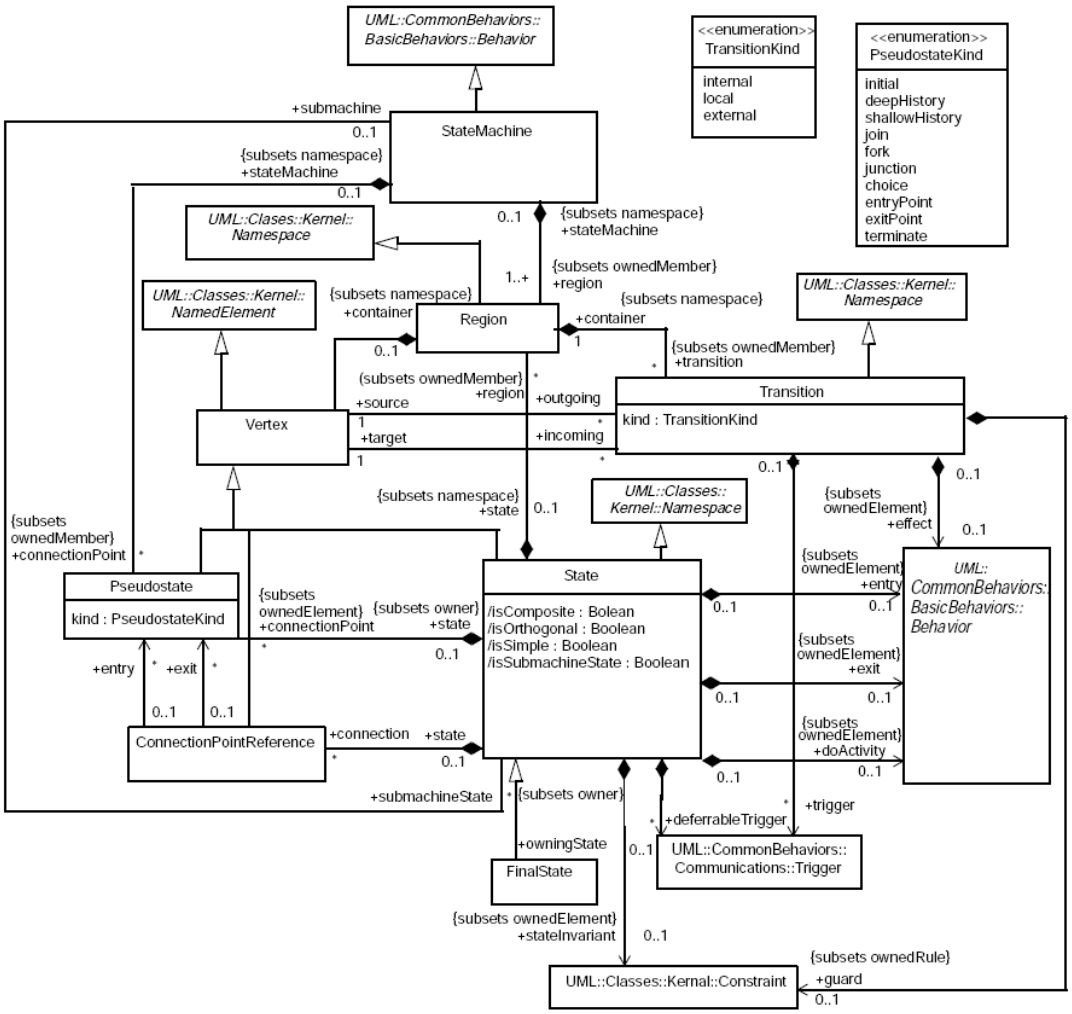


Figure 6.13: An extract of the metamodel of UML state machines [178]

StateMachine: a *StateMachine* is a concept used to exhibit the behavior of a part of system. This behavior can be expressed by execution traces of the state machine, obtained when transiting between states. The transitions are fired by events. During this execution, a series of activities associated with the elements of the state machine can be carried out. A *StateMachine* can be a sub-machine, i.e., it refines a state in another state machine. A state machine may have *Regions* and *Pseudostates*.

Region: a *Region* is an orthogonal part of either a composite state or a state machine [178]. Simply, a region is introduced as an intermediate element in order to describe the relation between state machines and other concepts used in them (e.g., states and transitions). A region may contain vertexes (states/pseudostates) and transitions.

Vertex: a *Vertex* is similar to a node if state machines are considered as node-edge graphs. But a node does not necessarily imply a state, i.e., a vertex can be also a *Pseudostate*, which is not a state but conveys some information about some states or state machines.

State: A *State* in a state machine can be any of the following kinds: *simple state*, *composite state* and *sub-machine* state. Composite states and sub-machine states make it possible to define state machines in an hierarchical way. In this dissertation, we are only concerned with simple states. This is due to the reason that we treat a state as a global system configuration, which is not composed of hierarchical sub configurations. Figure 6.14 shows an example of a state machine with simple states.

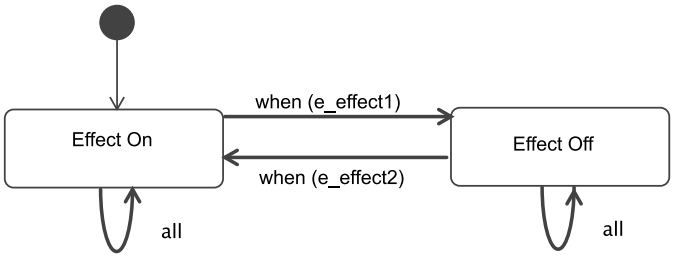


Figure 6.14: An example of a simple state machine

Pseudostate: pseudostates can be classified into several families: *initial*, *deepHistory*, *shallowHistory*, *join*, *fork*, *junction*, etc. These pseudostates convey the corresponding information of the states or the state machine they are connected to. In our dissertation, we are only interested in initial pseudo states as illustrated in Figure 6.15. The initial pseudostate is connected to one of the states in a state machine (more precisely, in a region), which is the initial state of the region.

Figure 6.15: Illustration of the initial pseudostate in UML

Transitions: a transition is a directed connection between a source vertex and a target vertex (state or pseudostate). A transition can have several triggers, any satisfaction of these triggers can fire the transition. In the previously illustrated example, the prefix *when* on a transition signifies a trigger associated with *ChangeEvents*. The prefix *all* indicates a self transition when no triggers are satisfied.

Collaboration

A collaboration specifies the relation between collaborating elements from a point of view of the functionality that they co-operate to accomplish. However, collaboration is not intended to define an overall structure of these elements. These elements in a collaboration are called *roles*, whose properties or identification can be ignored, i.e., only their useful properties and types are referenced in the collaboration.

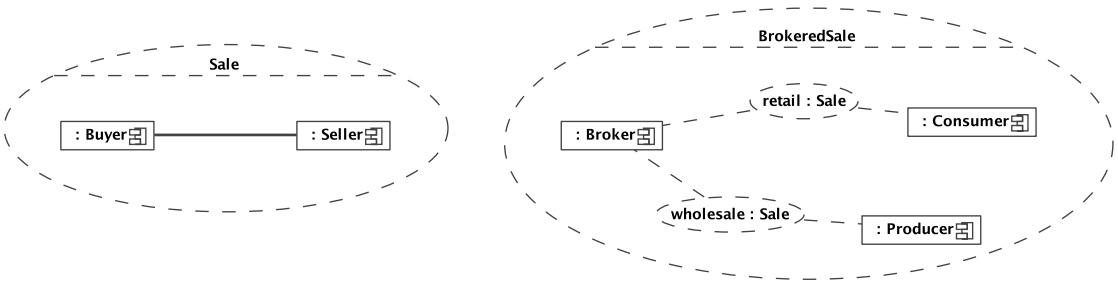


Figure 6.16: An example of collaborations taken from [178]

Figure 6.16 illustrates an example of collaborations, which are defined in a composite structure diagram with components. The two dashed ellipses express two collaborations, which are called *Sale* and *BrokeredSale* respectively. *Sale* describes the collaborating relation between *Seller* and *Buyer*. *Sale* is then used in the *BrokeredSale* to depict a more complex collaborating relation between *Broker*, *Producer* and *Consumer*.

6.4.2 MARTE metamodel with integrated state machine concepts

A view based composition in MDE as described in section 3.1.1 permits to present the different relations between the various concepts present in the extended MARTE metamodel. Hence a view is not a *total* representation, but allows to focus on a subset of a metamodel. We utilize this mechanism for the description of the extended MARTE metamodel.

Figure 6.13 shows an extract of the metamodel of UML state machines, which seems to be too complex. The state machines (the state graphs) used for the modeling of Gaspard2 control are only a subset of UML state machines. This is due to two reasons: a) UML state machines have been intended to be applied in all application domains, however many associated concepts are unnecessary in Gaspard2; b) in consideration of a concrete implementation in the form of model transformations, Gaspard2 should remain concise but expressive enough to simplify the development. The main concepts of UML state machines used in Gaspard2, and in turn MARTE, are enumerated: *StateGraph*, *Region*, *State*, *Vertex*, *Transition*, *Pseudostate*, *PseudostateKind*, etc. Relations between these concepts are illustrated in the Figure 6.20.

6.4.2.1 Overview

We first present the concept of a MARTE `StructuredComponent`, that enables associating behavior such as state machines with this modeling element. Afterwards, we introduce the notion of `OpaqueBehavior` in the MARTE metamodel that enables expressing those behaviors that cannot be specified using a graphical modeling methodology. Thereafter, we present the concepts of collaborations which help to determine behavior of certain modeled elements. Ensuingly, we present the global overview of the concepts related to state graphs which have been integrated in the extended version of the MARTE metamodel. Finally, the relations between events and state transitions are illustrated.

6.4.2.2 StructuredComponent and BehavioredClassifier

Figure 6.17 illustrates the basic concepts related to a MARTE `StructuredComponent`. This is equivalent to the concept present in section 4.2.1. A `StructuredComponent` defines a self-contained entity of a system, and can encapsulate structured data and behavior[4]. A `StructuredComponent` in MARTE specializes the abstract `BehavioredClassifier` concept. The `BehavioredClassifier` concept is the equivalent concept found in pure UML specifications, that permits to associate a `Behavior` with a UML classifier (classes, components and packages). This relation allows to model and associate behaviors such as state machines, sequence and activity diagrams with a `StructuredComponent`. As MARTE `StructuredComponent` has been previously presented in section 4.2.1, its details are not given here.

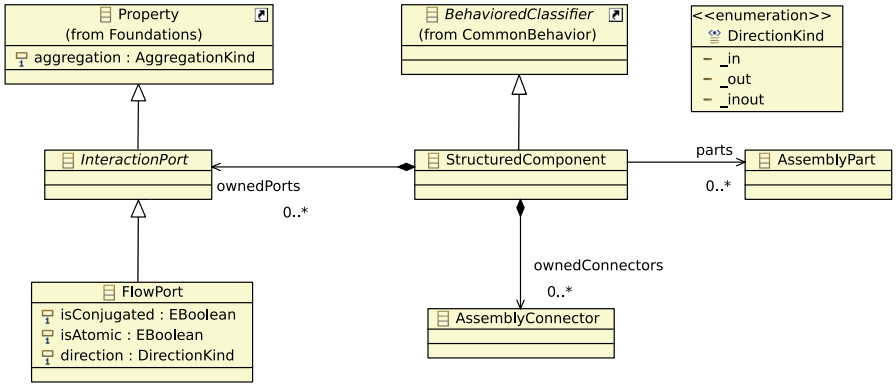


Figure 6.17: MARTE GCM package: Relation between a structured component and a behavior

[4]This behavior is different from the control behavior expressed earlier in the chapter and explains general behavior of a component

6.4.2.3 Behavior

In particular since in UML, as in MARTE, a `Behavior` is a kind of class, it is possible for a behavior to be its own structural context. A `Behavior` determines a specific comportment or action of a `StructuredComponent`. Different types of `Behavior` can be expressed in UML, such as automata (via usage of state machine diagrams, petri net like graphs by means of activity diagrams, use cases etc). It is up to the designer to determine the exact behavior and specification of a component associated with a behavior. `Behavior` can be either an atomic `Action` behavior or a `CompositeBehavior` (which may contain several behaviors itself). The model is inspired from (and hence compliant with) the *Common Behavior* model of the UML superstructure [178].

If the `Behavior` is owned by a `BehavioredClassifier` as indicated by the compositional *ownedBehavior* relationship, then the associated classifier (in turn the `StructuredComponent`) is the context for execution of the `Behavior`. Otherwise, the context is the first `BehavioredClassifier` reached by following the chain of owner relationships, as indicated by the *mainBehavior* reference.

We have also added the concept of `OpaqueBehavior` in the extended MARTE metamodel. An `OpaqueBehavior` respects the semantics of pure UML specifications, and allows to specify the behavior related to a `StructuredComponent` to be expressed in any language, such as natural language. The `OpaqueBehavior` class contains two attributes, specifically `Body` and `Language`. The `Body` attribute can be used to express behavior in any language and can be used to store user code or to express a custom action. The `Language` attribute determines the language used for expressing the behavior. The values associated with these attributes are ordered in the form of a list.

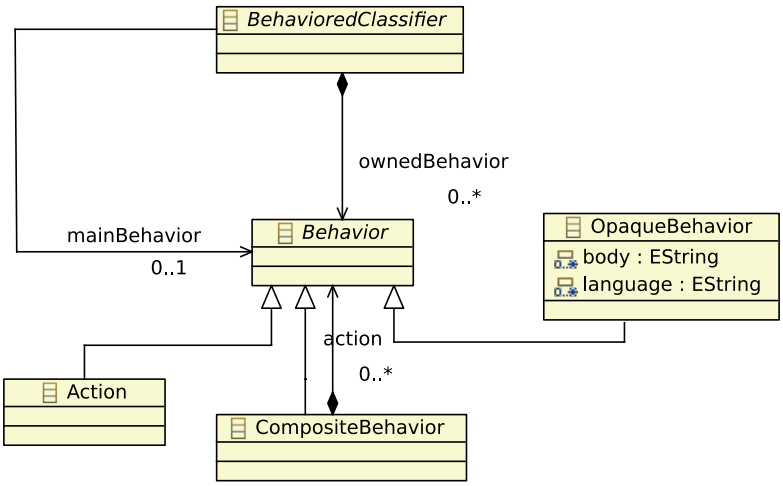


Figure 6.18: Behavior and OpaqueBehavior in MARTE

6.4.2.4 Collaboration

Figure 6.19 presents the integration of UML collaborations into the extended MARTE metamodel. A UML collaboration specializes from UML *StructuredClassifier* and *BehavioredClassifier* concepts. However, in the current MARTE profile, a `StructuredComponent` can be viewed as a UML *StructuredClassifier* as it contains different *ConnectableElements* or *Properties* (such as ports and instances). A MARTE `Collaboration` thus specializes from the `BehavioredClassifier` concept and has a reference *collaborationRole* to a MARTE `Property` (such as `AssemblyPart` and `InteractionPort`). A *collaborationRole* references elements possibly owned by other classifiers (such as connectors and ports); and represents roles that assembly parts play in this collaboration.

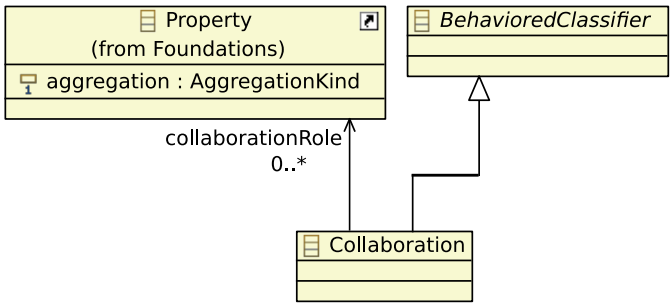


Figure 6.19: Collaboration concept in the extended MARTE metamodel

6.4.2.5 StateGraph concepts

Figure 6.20 represents an extract of the Gaspard state graph concepts which have been included in the extended MARTE metamodel. These concepts have been inspired from the metamodel of UML state machines. Since in section 6.4.1.1, we have already given a brief overview of UML state machines, and the added `StateGraph` concepts have nearly an one on one correspondence with the UML concepts, we do not again give the detailed explanation regarding the new concepts. A `StateGraph` concept coincides with the UML *StateMachine* concept, while other concepts such as `Region`, `Vertex`, `Transition` etc., have similar significations as found in UML. A `StateGraph` is a behavior which is in turn associated with a `StructuredComponent`, for example a controlling component. The concepts of `StateGraph` may differ slightly from the UML state machine concepts. This modification has been carried out explicitly in order to respect the MARTE metamodel, and for correct functioning of the model transformations which will take these concepts as inputs. A `StateGraph` may have at least one `Region`(s), which contain vertices and transitions.

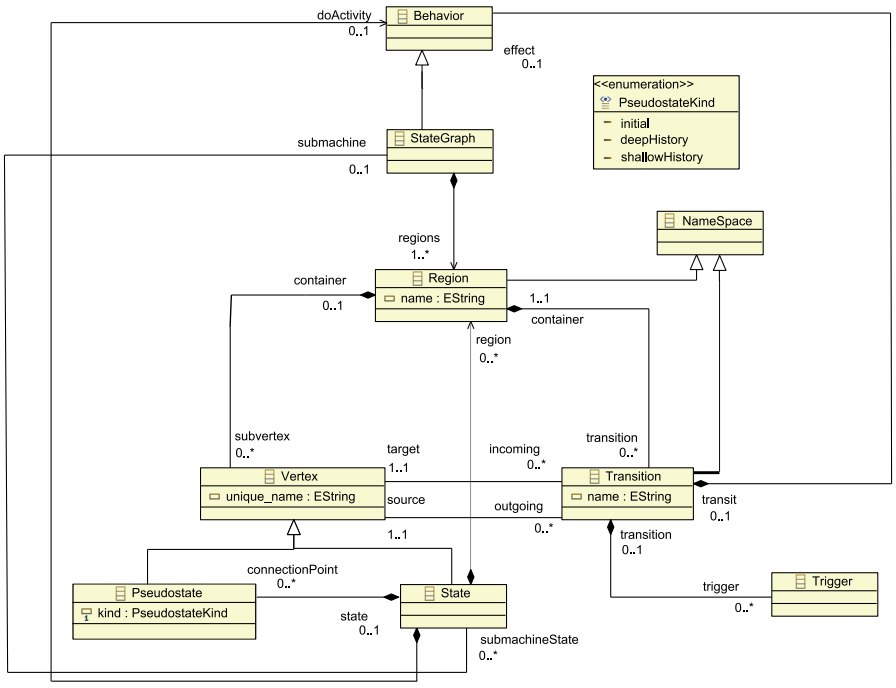


Figure 6.20: An extract of the Stategraph concept in MARTE

A `State` can be any of the following kinds: simple state, composite state and sub-machine state. The composite and sub-machine states make it possible to specify state machines in an hierarchical way. The extended state machine concepts introduced in the MARTE metamodel

allow to specify such state machines. This step has been carried out explicitly to serve other existing Gaspard2 transformation chains that may need these concepts. However, in this dissertation, we are only concerned with non hierarchical state machines. The hierarchical concepts can however be also used in a future evolution of our design methodology. A `State` has a specific `doActivity` , where we can specify a behavior carried out in this state. This behavior can be specified as an `OpaqueBehavior` for its expression, for e.g., in a natural language.

A `Vertex` can be either a `State` or a `Pseudostate`. It is used to indicate the source and target of a `Transition`. With respect to pseudostates, we are only interested in the initial pseudostate type. Note that this pseudostate does not exist in Gaspard state graphs, but is used to greatly simplify the graphical design in UML. It is kept in our proposition in order to respect UML semantics.

An initial pseudostate is usually connected to one of the states in a state graph (more precisely, a region), which is the initial state of the state graph (region). But if there is an input state port associated with a component linked to the state graph; and defined for the same region, the initial state indicated by the initial pseudostate is re-defined by the state obtained from the input state port.

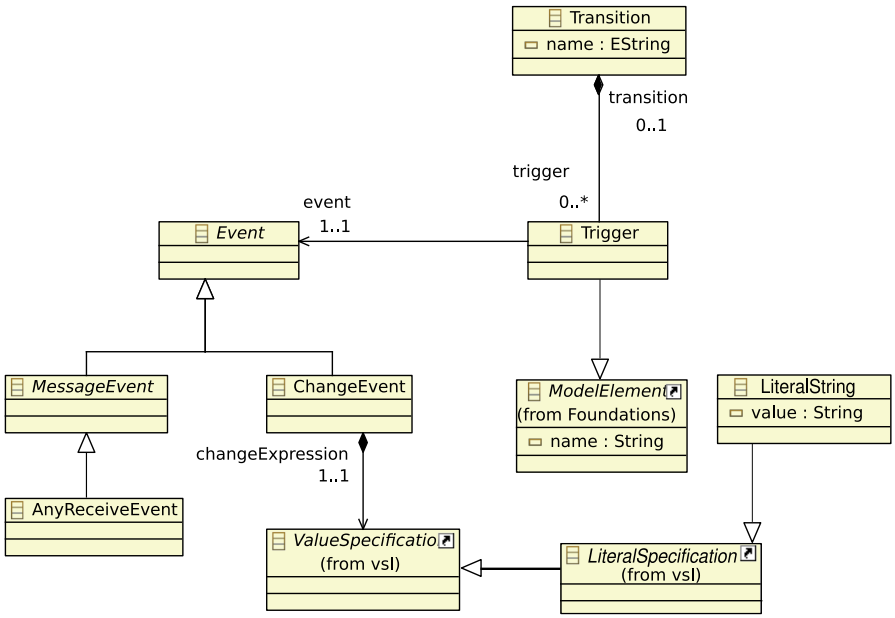


Figure 6.21: An extract of the added concepts related to Events in MARTE

Figure 6.21 shows the relationship between `Transitions`, `Triggers` and `Events`. A `Transition` between the source and target vertices (either states or pseudostates) is always associated with one or more `Triggers`. In the current MARTE specifications, the nature of a `Trigger` class is set as abstract, however in pure UML specifications, this is the inverse case. This is an ambiguity in the MARTE specifications, and has serious consequences for the eventual model transformations. In order to respect UML semantics, the nature of the `Trigger` class has been set as true. A `Trigger` is associated to one `Event`, which defines the specification of some event occurrence, potentially triggering a transition.

An Event is also an abstract class and is further defined into several subtypes. We have only integrated the subtypes `MessageEvent` and `ChangeEvent` which are crucial for the concepts of `StateGraph`.

An event which is used in the trigger of a transition is generally a `ChangeEvent`. This event has an expression called *changeExpression* that is a Boolean expression, which can result in a change of a state. By means of a compositional relation, the exact type of the *changeExpression* can be found in the *Value Specification Language* (VSL) package of the MARTE profile. Thus a `ChangeEvent` contains a `LiteralSpecification` which is further specialized into a `LiteralString`. Hence an expression or condition that helps to trigger a successful tran-

sition from a vertex is finally transformed into a literal string. An other kind of event is the `AnyReceiveEvent`, which can be considered as a default event. This event helps to trigger a transition by the receipt of an event which is not explicitly referenced in another transition from the same vertex. This `AnyReceiveEvent` specializes the `MessageEvent` class which itself is abstract in nature. A `MessageEvent` specifies the receipt of an event or a signal.

6.4.2.6 Modifying the concept of Datatypes in the MARTE metamodel

Figure 6.22 presents the integration of the attribute `nbbits` to the `DataType` metaclass present in the VSL package of the MARTE metamodel. This attribute has been added for the concepts related to the creation of hardware accelerators in our design flow. For hardware accelerators dealing with DIP applications, the incoming data from sensors/receptors is generally encoded in bits. This data can be varied between different types such as integer, float, etc. For example, [112] advocated the usage of 4-bit encoding of a digital signal for an analog-to-digital radar signal conversion. Moreover, in image processing applications, data streams of complex types are used. Similarly, enumerated types are used for defining user defined types. MARTE already provides these basic data types, however, as it does not targets a specific platform, the concepts related to bits for representing data types are absent. As we deal with reconfigurable FPGA based architectures, this is a crucial concept. Subsequently, its integration in the MARTE metamodel allows to specify the type of incoming and outgoing data in a hardware accelerator[5].

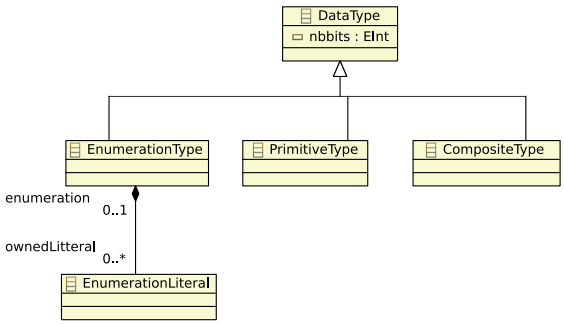


Figure 6.22: Modification of the data types concepts in MARTE metamodel

6.4.3 Deployment metamodel

Using the MARTE UML profile, Gaspard2 SoC Co-Design framework permits specification and modeling of an application, an architecture and their respective allocation using a high level component based approach. The system is completely detailed until the lowest hierarchical level is achieved. This level consists of atomic *elementary components*, which can be viewed as *black boxes*. Thus the interior functionality or behavior of such a component cannot be visualized via a modeling approach such as state machines or activity diagrams as these components are typically related to low level technological details.

Relating to the component definition specified in chapter 2, only the interface, i.e., the input and output ports of the component are visible to the environment, and the component implementation remains invisible. However, for the conception of a complex system, knowing the internal details of these components may not be extremely important as compared to their QoS criteria, as they are the basic building blocks of the system and should be utilized as such. These elementary components have corresponding available implementations in the form of source code. For hardware modules/components, this code can be in the form of an HDL (such as VHDL or Verilog) or SystemC. Similarly for software components, the implementation can be either in the form of assembler code, or written in a high level language such as C/C++, Fortran, etc. These related implementations are viewed as *Intellectual Properties* or IPs. This key term is

[5]The data types of the ports of our modeled applications for eventual conversion into hardware functionalities can be specified with a range: For example, an input port having the type INTEGER RANGE 0 TO 7 is encoded in 4 bits

mostly used in SoC hardware domain, but for reasons of symmetry, we also use this terminology equally for their software counterparts frequently referred in the form of user specified or standard libraries.

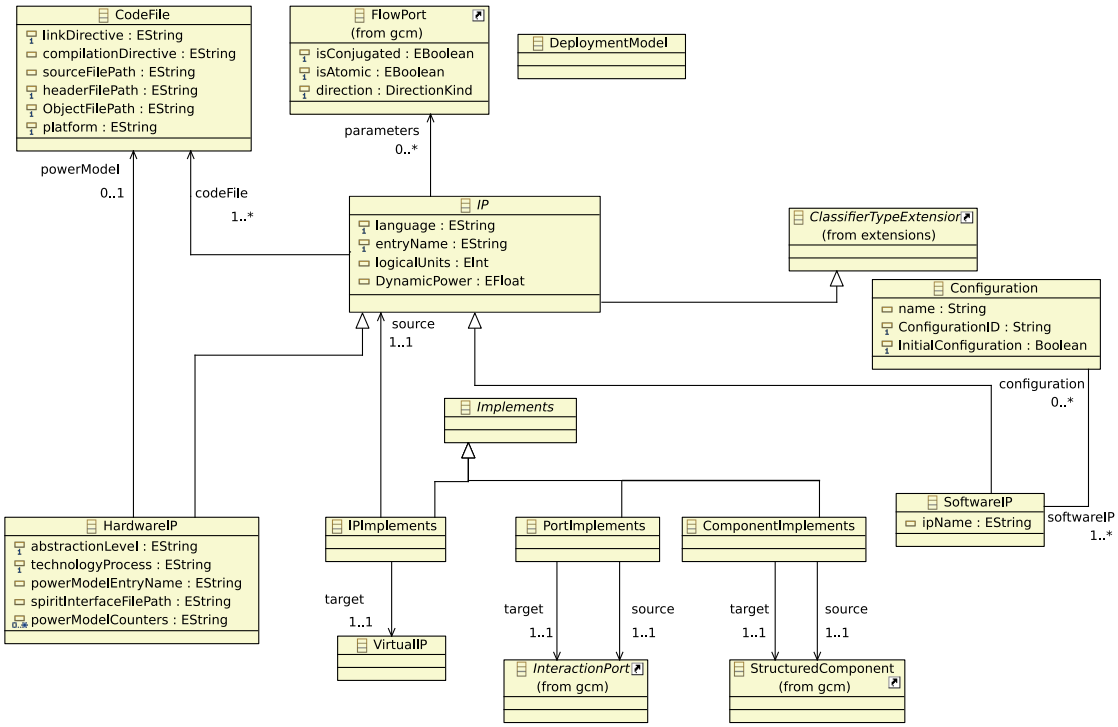


Figure 6.23: Global overview of the MARTE integrated Deployment package

In order to generate, and afterwards execute a complete SoC from high level models, the designer should be able to precisely associate an elementary component with its corresponding IP(s), while remaining at these high abstraction levels. This information is necessary in order to assure that the model of a SoC is executable, i.e., it should not only be possible to just generate the code skeleton, but the code should be correct, error free and effective enough to be directly compiled and executed for different purposes, such as simulation or synthesis. For responding to these questions, in [27, 194], some initial concepts have been presented which allowed to link the high abstraction level models to the IPs. This collection of concepts was developed in the form of a metamodel, termed as a *Deployment metamodel* (for utilization in the model transformations), and corresponding profile, i.e., the *Deployment profile* (utilized for the conception with the help of UML tools such as Papyrus).

As compared to the deployment specified in these earlier approaches, in this dissertation, we present a MARTE compliant deployment level, which also respects the semantics of traditional UML deployment mechanism. A part of this deployment is collectively developed by the Gaspard2 research team, in addition to our own personal contributions. The respective deployment metamodel and profile concepts have been integrated into the MARTE metamodel and its profile respectively.

6.4.3.1 Overview

The basic principal of the deployment level is to give designers the means to link the basic building blocks, which are the elementary components, to implementations that embody the related behavior. More precisely, sufficient information must be provided so that code integration in these IPs can be carried out automatically, at the time of code generation and execution. Thus, it should be made possible to associate *at least* an IP with each elementary component. Similarly, information related to the IP interfaces must be taken into account as well. Thus, interface compatibility between an elementary component and associated IP must be ensured in order to

determine the exact manner for invocation of an IP. The deployment metamodel has three main features, which are explained subsequently.

Associating IP with properties. Firstly, to properly correspond to the manner by which the SoC industry operate, the IP must have associated properties. For example, in case of a hardware IP, properties such as consumed platform resources must be provided. In case of FPGA based reconfigurable SoCs, these resources can be the number of logical resources such as CLBs. Similarly for software IPs, information related to compilation options must be provided.

Component re-utilization. Secondly, in order to correspond to SoC Co-Design aspects, this level allows the re-utilization of components. As seen in chapter 1, the re-use of hardware and software IPs is extremely important for design productivity. When developing a new SoC, most elementary building blocks are not developed or created from scratch, but are off-the-shelf components internally developed or purchased from third parties responsible for IP verification. This level should also allow the creation of IP libraries. The deployment level extension in the MARTE standard pays special attention that not only the IPs can be described independently from a specific SoC model, but also that the usage of IP from a library is simple and intuitive as possible. Particularly, evolution in the IP library and the SoC model must be independent from each other. Additionally, deploying an elementary component to an IP must not be a tedious task. Graphically it could be simple as drawing an arrow between the two concepts.

Abstraction of associated functionality. Finally, an elementary component linked to an IP is an abstraction of the functionality related to the IP. However, this IP is specific to a target execution platform and technology. For example, for targeting simulation at SystemC TLM level, it is preferable to use hardware components associated with SystemC TLM IPs. However, when the same model is targeted for a more detailed lower technology level such as RTL, and there exists an IP written in HDL such as VHDL; then it is preferable to select that IP. More importantly, the IP can be changed respective of the final target. For example, if a software elementary component is allocated to a programmble processor (for example, a hardcore/softcore embedded processor in a target FPGA), then its IP will be implemented in assembler, C etc. However, if the same component is to be treated as a hardware functionality and is consequently mapped to a hardware accelerator, then the IP will be implemented in SystemC or VHDL, etc.

It is thus not desirable to change the deployment level each time a different allocation in the SoC model is carried out or a different technology level is selected. This level must provide an abstraction so that the different IPs having the same functionality can be regrouped and linked with an elementary component, and thus provide flexibility related to the allocation and the selected execution platform/technology level.

Figure 6.23 illustrates the global overview of the deployment metamodel. This view gives a first impression of different metaclasses and their relationships. We now go into detail related to each class or group of classes depending upon their functionality. It should be observed that the root of this level is the `DeploymentModel`, and all the other classes are directly or indirectly related to this class by means of a composition mechanism. Certain number of classes are not specific to the deployment metamodel, but are concepts present in different packages of the MARTE metamodel, such as `StructuredComponent`, etc. These metaclasses are the different *crossing points* that permit to link deployment concepts to other existing concepts present in the MARTE metamodel.

There also exists an additional part of the deployment level which is mainly concerned with non-functional properties of hardware components. This is shown by the relation *powerModel* between the concepts `HardwareIP` and `CodeFile`. As in this dissertation, we primarily focus on the application aspects of a SoC model, we have not detailed this extension, which is written in detail in [27].

The description of the deployment metamodel can be classified into three stages, in order to increase flexibility at the high modeling level; and in order to avoid adding redundant information. The lowest stage, expressed with the class `CodeFile`, corresponds to the description of a source file. The intermediate stage, expressed with the abstract class `IP` determines the description of an IP/implementation for a particular target. Finally, the highest level is expressed with

the help of the `VirtualIP` class, corresponding to a functionality, independent from a targeted technology or execution platform. This concept permits to target different execution platforms, from the same SoC model.

Each functionality can correspond to either a `SoftwareIP` (thus expressing the functionality of the system), or a `HardwareIP`. The notion of `Implements` helps to determine the relations between the different stages of the deployment level and the SoC model. This concept is further refined into several sub types and helps in the construction of an IP library. An elementary component, its associated `VirtualIP` and available `IPs` are defined in a model. Afterwards the chosen `IP` is linked to codefile(s) in another model. The following sections describe in detail each concept of the metamodel and their corresponding attributes.

6.4.3.2 CodeFile

The goal of the deployment level is two fold. Firstly it helps in the final code generation of the high level SoC model. Secondly, it also provides the semantics to ensure correct code compilation/execution by providing all necessary source files related to a selected IP. The class `CodeFile` provides features for fulfilling this last aspect. A `CodeFile` provides a graphical representation of *only one* source file. It thus provides the mandatory sourceFilePath attribute for determining the physical location of the source file (it thus provides the name of the source file along with its extension) in the file system of a computer, which carries out the code generation. This class also contains other additional attributes such as compilationDirective and LinkDirective. Since we intend to treat the modeled application into a hardware functionality, these attributes are not mandatory for our needs. Detailed explanation about these concepts can be found in [194].

Figure 6.24 shows an example of utilization of the `CodeFile` in the MARTE profile with integrated deployment concepts. Here two implementations `VSIPL-Multiplication` and `HwAcc-Multiplication` are linked to their source files via their respective `CodeFiles` and appropriate properties.

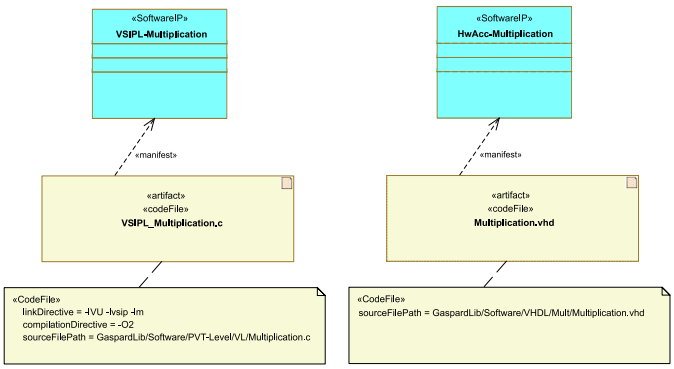


Figure 6.24: Example of the usage of Codefile in Gaspard2

6.4.3.3 IP

The abstract class `IP` directly represents the concept of an intellectual property or implementation, which can be either software or hardware based in nature. By means of the *codeFile* reference, the source files necessary for implementing an `IP` are listed. There is no direct correspondence between an `IP` and associated `CodeFile`. An `IP` may require several `CodeFiles`. For example in C++; it is usual to have code and declaration files for each object. Similarly, a hardware IP written in VHDL may require several source files in case of a compositional hierarchy. Also, different `IPs` can share the same `CodeFile` as well. For example, a collection of similar functions of a library can be regrouped in the same file.

The `IP` class contains several attributes, permitting to specify the relationship between the generated code of a SoC model and the code of an IP. This information is the core foundation of

the deployment level. The Language attribute determines the programming language used for developing the IP, such as C, Fortran, SystemC, VHDL, Software Binary etc.

The entryName attribute helps to determine how the IP is invoked, if it is software based in nature. The IP thus corresponds to a function which returns an integer value, and whose name is determined by the `entryName`. Details related to invoking IP functions related to purely software IPs have been presented in [194].

The logicalUnits attribute determines the number of logical units consumed for an implementation. This attribute can be either specified as a *Logical Element* (LE) or a *Slice* depending upon the targeted FPGA series. Finally the DynamicPower attribute determines the dynamic power consumption levels for a given implementation. These last two attributes have been specifically created for resource/energy estimations at the RTL level and are independent of the nature of the IP itself: either a hardware or software IP.

6.4.3.4 SoftwareIP

The class `SoftwareIP` inherits from the `IP` class and thus represents a particular implementation of an IP. It corresponds to an elementary component present in the application model of the complete system. No matter the final generated form of the IP, all the IPs which correspond to a Gaspard2 application are represented by this class. For example, an IP that carries out an FFT is represented by a `SoftwareIP`, even if it is written in VHDL or Verilog (thus acting as a hardware accelerator). Similarly, even if different tasks of an application are allocated on different hardware resources present in a SoC, such as allocation of task1 onto an embedded processor and task2 onto a hardware accelerator, IPs related to both tasks are still represented by a `SoftwareIP`.

6.4.3.5 HardwareIP

The class `HardwareIP` permits representation of an IP corresponding to an elementary component of the targeted hardware platform. It contains several attributes related to targeted hardware platforms. A `HardwareIP` also contains a `PowerModel`, the details of which are out of the scope of this dissertation. We refer the reader to [27] for a detailed explanation related to these IPs.

6.4.3.6 Virtual IP

An IP represents an implementation for a given target (for example, an abstraction level in a given language). If an elementary component is deployed directly onto a desired IP, then loss of an abstraction occurs. The drawback of this absence is that it will not be possible to entirely generate a SoC, and the different possible targets will not be compatible with this IP. Thus the high level models become dependent on the final implementation technology/execution platform. The principal disadvantage is making the designer modify the high level models, respective of the selected model transformations. To avoid this pitfall, the notion of a `VirtualIP` is present in the deployment level. This class regroups all the IPs having the same functionality. A `VirtualIP` can contain several IPs. Each of these IPs must be equivalent: Each IP must have the same interface, i.e., thus they must have the same number of *in*, *out* or *inout* ports. These ports must have the same semantics, type and shape. Similarly, the `VirtualIP` associated to these IPs must also share the interface characteristics which are shared by the grouped IPs.

Only non-functional properties of these IPs are allowed to be varied. Other than the programming language of the IP, we can imagine that the IPs vary based on different QoS criteria such as execution time, precision of computation, energy consumption, latency, consumed resources, etc. These properties can be specified by means of attributes present in the class `IP`, as well as its respective types: `SoftwareIP` and `HardwareIP`. During the deployment phase, a SoC designer can link an elementary component to a `VirtualIP`, based only on the functionality desired by the designer, without specifying a precise IP.

It should be made evident that for reasons regarding comprehensibility and to promote the usage of this abstract concept, an IP must be related to a `VirtualIP` component, even if there is only one IP available for the elementary component. This could provide beneficial in the

long run, if other IPs with same functionality are developed or become available for the same elementary component.

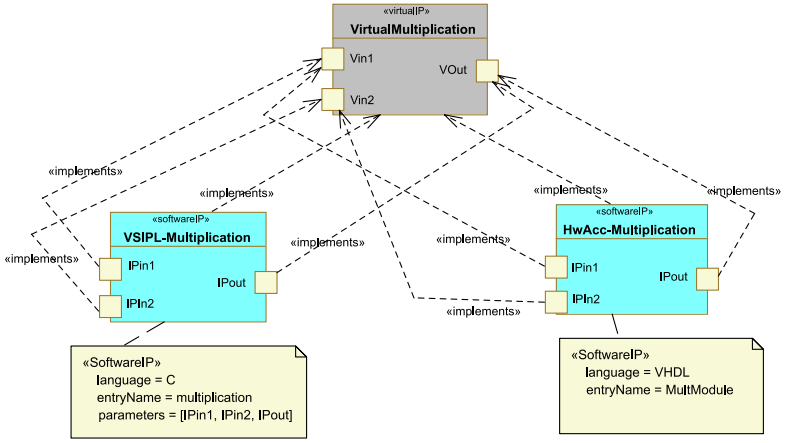


Figure 6.25: Grouping different implementations in the same functionality

Figure 6.25 shows an example of usage of a `VirtualIP`. Here two implementations `VSIPL-Multiplication` and `HwAcc-Multiplication` are grouped together for the abstract multiplication functionality `VirtualMultiplication`.

6.4.3.7 Implements

Until now, we have presented the mechanisms for the description of an IP and its interface. The notion of the abstract `Implements` class permits to create concrete links between this description and the elementary components. This class is further decomposed into three subtypes. An `IPImplements` permits to link different IPs to a `VirtualIP`, while a `PortImplements` permits to associate ports of an `IP` to a `VirtualIP`, and consequently the ports of a `VirtualIP` to the ports of an elementary component in question. Finally, the `ComponentImplements` permits to link either a hardware/software IP or `VirtualIP` to an elementary component. These classes are independent of other classes specified until now, and help in the creation of an IP library. The IPs and the source files are defined in a self sufficient manner in the model; and afterwards, the concepts related to `Implements` and its subclasses can be modified independently in order to create a link between the definitions of an IP and a SoC model. Care shold be taken to ensure that different elementary components are not deployed on the same implementation or IP.

6.4.3.8 PortImplementedBy

The class `PortImplements` creates an association between the ports of an elementary component and the ports of a selected IP. In order for this relation to take place, several conditions must be fulfilled. Firstly, a port of an IP must have the same direction as that of the respective port of an elementary component. This ensures that the model rests coherent and error free.

Similarly, as the IPs are grouped together in the form of an abstract `VirtualIP`, this condition must hold valid between the respective ports of the `VirtualIP` and the `IP`. Initially all the ports of an `IP` are linked to the respective ports of the `VirtualIP` by means of `PortImplements`. Afterwards the ports of a `VirtualIP` are linked to the respective ports of the elementary component also by `PortImplements`. This two stage approach offers abstraction advantages, thus, when a designer needs to switch between different implementations related to an elementary component, via the abstraction mechanism offered by the `VirtualIP`, he does not need to change the interface links between an elementary component and a virtual implementation. he only needs to change a single reference, termed as a `ComponentImplements` link. This concept is presented subsequently.

Even if an implementation is not selected by an elementary components, its ports must still be linked to the ports of a `VirtualIP` in case of future modifications of the model. The references *source* and *target* permit to determine the source and target ports linked by a `PortImplements`.

6.4.3.9 ComponentImplements

The class `ComponentImplements` contains two references *source* and *target* for linking two MARTE structured components. This class offers two functionalities. First it enables linking a `VirtualIP` to an elementary component. This permits an elementary component to be associated with a desired functionality represented by a `VirtualIP` and consequently the implementations. Secondly, among the different available implementations, once a designer selects an `IP` based on different QoS criteria, this implementation can be linked to the elementary component by means of a `ComponentImplements` dependency, for selecting the actual implementation of the elementary component.

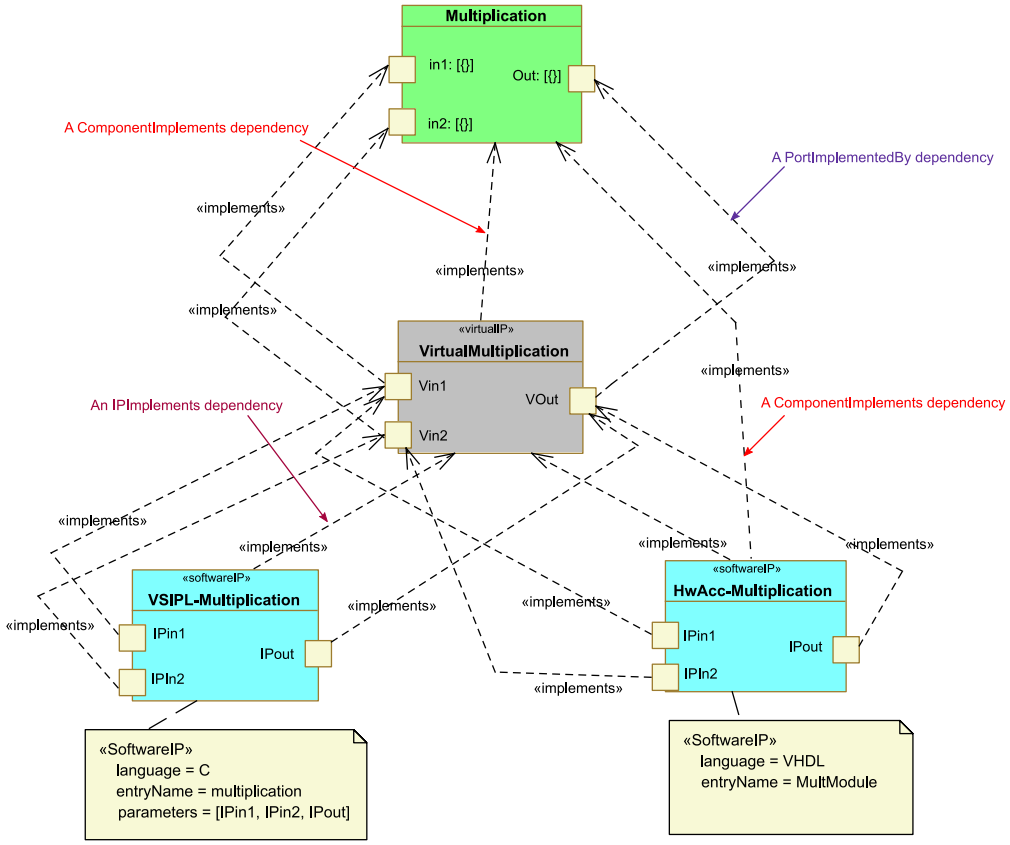


Figure 6.26: Explanation of the connectors

6.4.3.10 IPImplements

The class `IPImplements` helps in creating a link between an IP and a `VirtualIP`. Irrespective of the IP selected for an elementary component, all the IPs available for an elementary component must be linked to the `VirtualIP`. In combination with `PortImplements`, this concept associates a functionality with its different implementation variations.

Finally, Figure 6.26 represents a clear global overview of the deployment mechanism related to an elementary component, with the different *implements* dependencies highlighted for a clear distinction.

6.4.3.11 An adaptable deployment level

Limitations of current deployment level. The concepts of deployment until now, help to create a static implementation of a reconfigurable system. This is due to the reason that while the given concepts allow to associate multiple implementations to a given elementary component, ambiguities arise when a designer needs to determine, and afterwards implement a system configuration. This configuration in turn is concerned with the elementary components and their associated implementations.

For e.g., in an application, an elementary component *P* has three implementations *P1*, *P2* and *P3*. A designer may wish to select all implementations to be associated to *P* for targeting a reconfigurable SoC. He may wish to implement three systems configurations *ConfigurationQ*, *ConfigurationR* and *ConfigurationS*, associated to *P1*, *P2* and *P3* respectively. However, during the implementation phase, it would not be possible to determine which implementation belongs to which configuration via the current deployment model semantics. As the number of elementary components and their available implementations increase, the complexity increases exponentially. Thus a mechanism needs to be put into place to resolve this design ambiguity present at the high modeling deployment level.

Notion of *Configurations*. In order to respond to the above mentioned issues, we introduce the notion of a `Configuration` in the deployment metamodel[6]. A `Configuration` can be either specified for the application or architecture or can be viewed as a collective composition (mapping of the two aspects to form a global system). Since in this dissertation, we only take an application model as input for final implementation in a target FPGA, currently we associate only the software implementations to a configuration. A `SoftwareIP` can thus be part of a `Configuration`, helping in determining if the concerned IP is part of one or more configurations for a reconfigurable system; or more specifically in the case of this dissertation, partially reconfigurable FPGA based SoCs. It is however possible that a `SoftwareIP` is not included in one or any of the configurations required by the designer or environment.

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 `Configuration`, which in turn are used by the control aspects presented earlier in section 6.1.3. 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. 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 whether it is the initial configuration to be loaded on to the target FPGA. This attribute also helps to determine the initial state of the Gaspard state graph.

Thus, in combination with the control concepts, deployment level creates several configurations for the final effectuation(s) in an FPGA. Each configuration is viewed as a collection of different IPs, with each IP associated with its respective elementary component. The current model transformations for the RTL transformation chain have been modified to generate different implementations of a hardware accelerator (with each corresponding to one specified configuration) in an FPGA, as illustrated in chapter 8.

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 (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

[6]While in the latest version of MARTE, a notion of configuration has also been presented, it is not associated to IP levels, as illustrated in appendix B

eventual final implementation. This condition is determined by the *softwareIP* reference from a `Configuration` to the `SoftwareIP` class.

Figure 6.27 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. Being abstract in nature, the figure omits several concepts such as `VirtualIP` and `Implements`. However, this representation is very close to UML modeling as represented in section 6.5.2. 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 used FPGA resources.

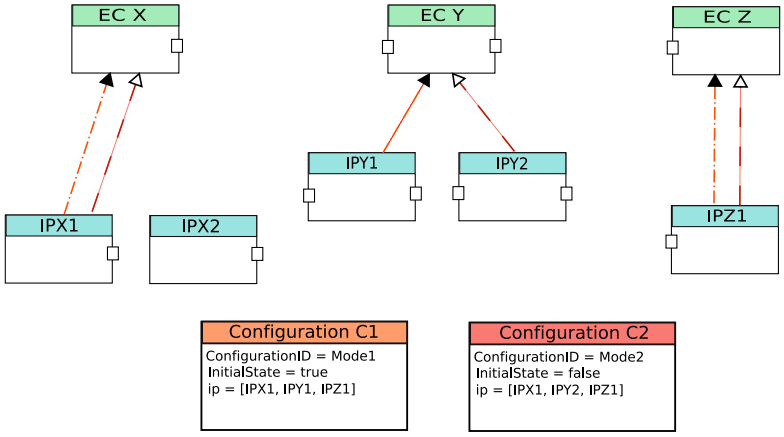


Figure 6.27: Abstract overview of configurations in deployment

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 (case of *IPX2*).

6.4.4 GaspardLIB

GaspardLIB is an added extension in the Gaspard2 Framework, which represents a significant collections of IPs in the form of a library. This library enables users to reuse components by providing the related basic elementary components for construction of complex SoC models. The library provides a clear distinction between the software and the hardware IPs for different abstraction levels (TLM-PA, CABA, RTL) and different technologies. This library is currently being adhered to the IP-XACT standard [13].

The library contains IPs from different environments such as SoCLib, as well as user defined IPs. Currently a modeling mechanism of the IPs is underway, which will enable a designer to view the different available IPs in the graphical Gaspard2 environment; facilitating in the modeling of a SoC system. The associated properties related to the IPs in GaspardLIB have already been detailed, such as the required source code files. However other attributes such as dynamic power consumption depend on the chosen target platform. Currently the IPs related to the RTL chain have been incorporated in GaspardLIB during this dissertation.

6.5 MARTE profile examples

In the previous sections, we first described abstract control semantics, followed by the control and deployment metamodel concepts that are integrated into the current MARTE metamodel. In order to provide a detailed understanding and usage of these concepts, we now provide examples of some models specified using the MARTE profile with our integrated contribution.

The MARTE profile has been integrated with the introduced deployment concepts, while the control semantics use pure UML semantics, permitting their modeling with UML tools such as Papyrus supporting the MARTE profile.

6.5.1 Example of a Multiplication-Addition application

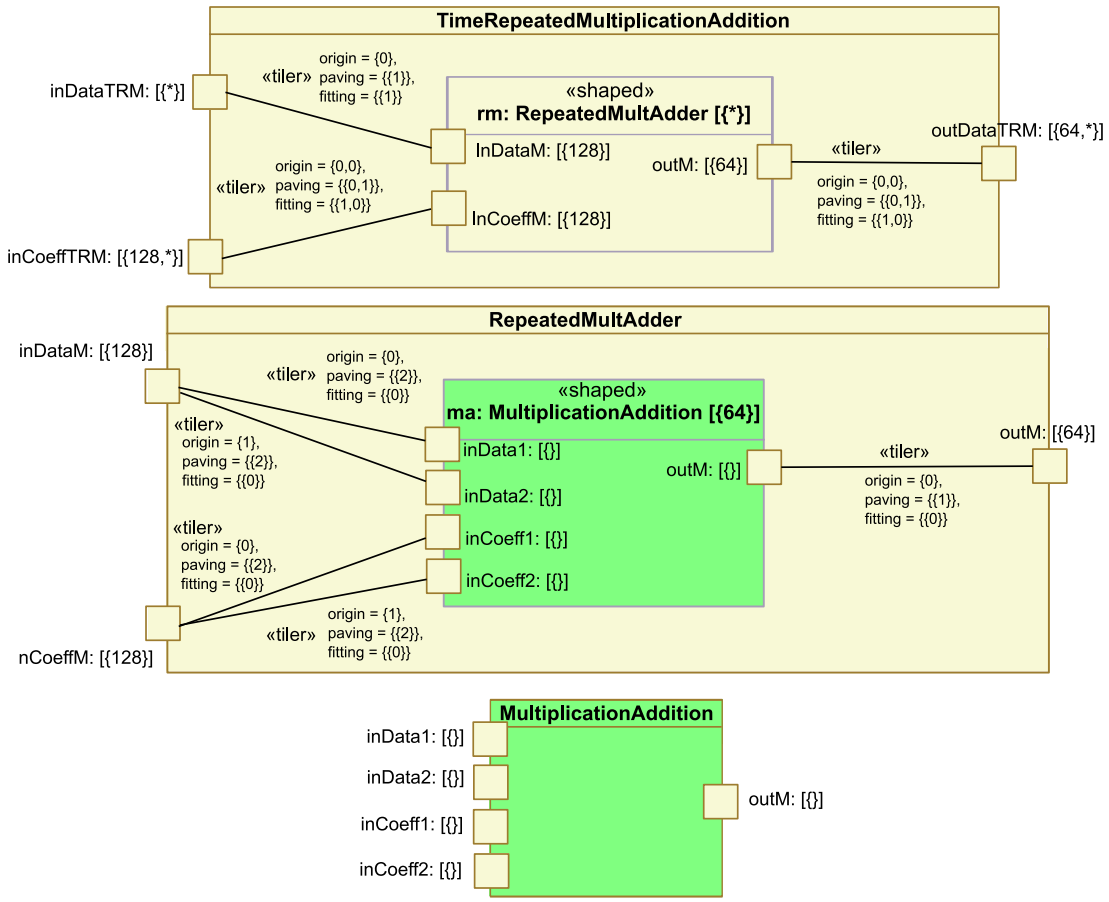


Figure 6.28: A Multiplication-Addition application

In Figure 6.28, we provide an example of a typical DIP application modeled in the Gaspard2 framework. The application in question is a multiplication-addition application[7] which takes some data and coefficients and carries out multiplication and addition. Details about the functioning of the application is presented in the case study in chapter 9.

The top level of the application `TimeRepeatedMultiplicationAddition` contains three ports: `InDataTRM` and `inCoeffTRM` of input direction; and `outDataTRM` of output direction, respectively with shapes of $\{*\}$, $\{128, *\}$ and $\{64, *\}$. The infinite dimension on the `InDataTRM` port indicates that a single data value is present at each instant of time, while 128 coefficient values are consumed according to the shape value of `inCoeffTRM`, at each instant. Finally 64 data outputs are produced similarly at `outDataTRM` at each instant of time.

If this application is targeted towards an execution model such as RTL, that means that single data and 128 coefficients are consumed and 64 data outputs are produced at each clock pulse. The `TimeRepeatedMultiplicationAddition` is viewed as an RCT and contains a repeated `RepeatedMultAdder` task having a shape of $\{*\}$, indicating that this component is repeated once at each iteration of its repetition space. In simpler terms, it means that that temporally repeated RT is executed sequentially at each clock pulse.

[7] The port types of this application have not been defined in this chapter, and are detailed subsequently in chapter 9

The `RepeatedMultAdder` component has its respective input and output ports: `InDataM` and `InCoefffM`, having respective shape values of {128} equivalently. A `tiler` connector helps to connect the input port `InDataTRM` of the `TimeRepeatedMultiplicationAddition` to the input port `inDataM` of the `RepeatedMultAdder` component. Similarly other tilers connect the input/output ports of the two entities.

The tiler connectors permit to determine how the initial data array is divided into sub-arrays/patterns. In this particular case, at each instant of time, the `RepeatedMultAdder` component consumes patterns of shape value of {128} on its data and coefficient ports, while producing {64} data outputs.

Descending to another hierarchy level, the `RepeatedMultAdder` component itself contains a repeated task. The `MultiplicationAddition` component has a shape of {64}, meaning that it is repeated 64 times in parallel at each instant of time. This elementary component itself has its respective input and output ports `inData1`, `inData2`, `inCoeff1`, `inCoeff2` and `outM` with shape of {} respectively. Two tiler connectors are used to connect the input data port of `RepeatedMultAdder` to the input data ports of `MultiplicationAddition` component. Similarly, two tiler connectors connect the respective coefficient ports of these two components. Finally the output ports of these components are also connected by means of a tiler connector. These connectors also express the data dependencies between different repetitions of the `MultiplicationAddition`.

Finally, the `MultiplicationAddition` is an atomic elementary component. Once the application modeling is finished, we move on to the deployment phase where this elementary task is deployed to the available IP(s), along with the modeling of the control features.

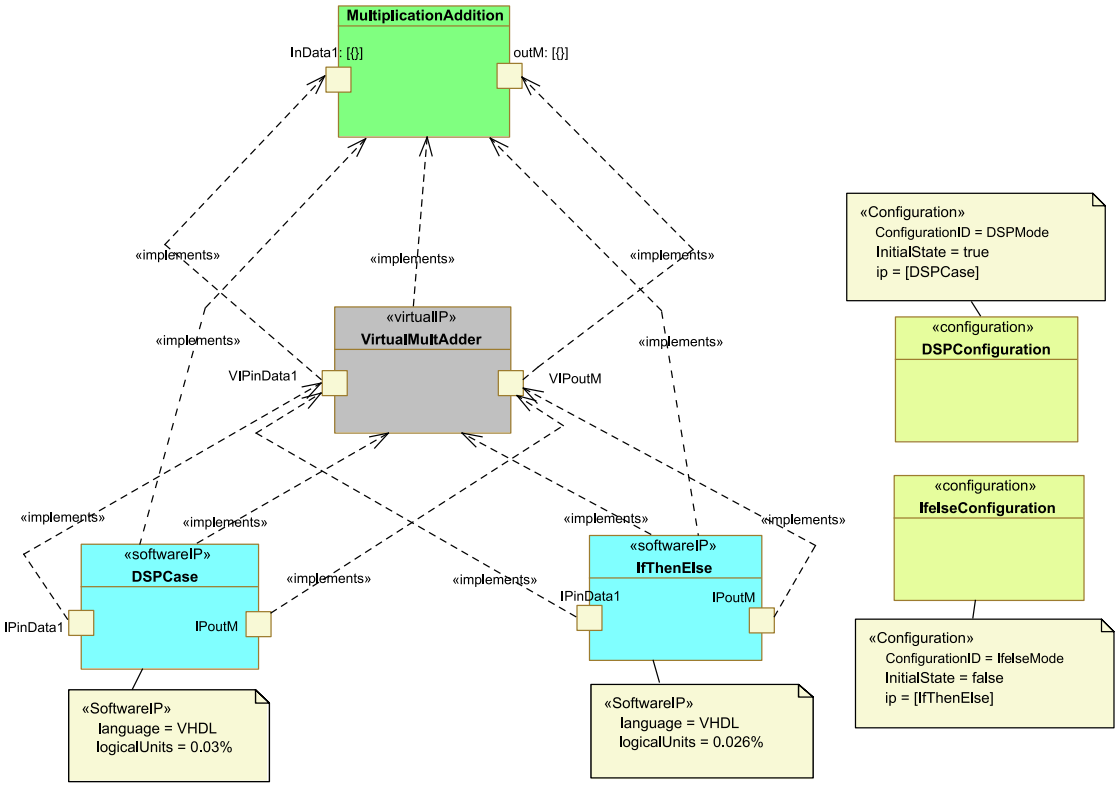


Figure 6.29: Deploying the elementary component of the multiplication addition application

6.5.2 Deploying the elementary component

The deployment of the elementary component `MultiplicationAddition` onto the available IPs is shown in Figure 6.29. The elementary task has two available IPs: `DSPCase` and

`IfThenElse` to implement the same functionality but using different implementations, one based on a DSP type implementation; while the other written as an if-then-else statement.

The two available IPs are written in VHDL, in order to convert the specified application model into a hardware functionality. At the deployment level, the designer now has different choices with respect to QoS aspects. If different QoS criteria of the available IPs are known before hand due to a DSE strategy, the designer can deploy an elementary component to different IPs for different scenarios, resulting in different overall results. In this example, we have assumed hypothetically that different measures related to the IPs are available before hand. Among several available QoS criteria, we choose to exploit the measures related to consumed reconfigurable resources in a target FPGA, as this dissertation is mainly concerned with these reconfigurable architectures. The `DSPCase` IP has the logicalElements attribute set with a value of **0.03%**, which determines the number of logical units (Slices or LEs) in the targeted FPGA. Similarly, the `IfThenElse` IP has this value set to **0.026%**. These values correspond to the consumed resources for the IPs in a specific targeted FPGA. While these resources may seem trivial related to a single IP, but when these IPs are repeated multiple times and form a part of complex application that is translated into an hardware functionality; the consumed FPGA resources are deemed significant.

Thus the designer can deploy the elementary component `MultiplicationAddition` to either `DSPCase` or the `IfThenElse` IP for a static architecture, which will result in the generation of one complete hardware functionality (a hardware accelerator) in VHDL for synthesis and subsequent implementation in the targeted FPGA. In the case of dynamic reconfiguration, and specially partial dynamic reconfiguration, the designer can deploy both IPs to the elementary component, in order to be used in different configurations of the modeled application (and the subsequent hardware accelerator). Currently in order to render the modeling simple for the reader, we have only illustrated one single input port and the output port of the elementary task. The multiple port implementations from the IPs to the Virtual IPs and to the elementary component might make the diagram complex, and our intention is to initially provide a simple explanation. However, the correct full version of the deployment contains all the ports of these components and their respective port implementations, as illustrated in chapter 9.

The two modeled components `DSPConfiguration` and `IfelseConfiguration` represent the two possible configurations related to the elementary component, and the overall application. The attribute ConfigurationID assigns the unique IDs of **DSPMode** and **IfelseMode** to `DSPConfiguration` and `IfelseConfiguration` respectively, which helps to distinguish between the configurations in the model transformations. The InitialState attribute determines the initial configuration (in the form of a bitstream) to be loaded on to the FPGA. In the case of PDR, this initial configuration is merged with the static part of the reconfigurable architecture (static bitstream) to form the default startup configuration. In this example, `DSPConfiguration` is selected as the initial configuration. This attribute also helps to determine the initial state of the deployed mode automata as explained earlier in section 6.4.2.

Finally the ip attribute permits to link a configuration to the available IPs. As stated before, a configuration must contain at least one IP. As in this example, the application only contains one elementary component having two available IPs, each configuration is associated with a unique IP. For the sake of simplicity, we have omitted the visual representation between the IPs and their respective `CodeFiles` associating an IP to its source file(s).

While the deployment modeling is sufficient for describing the different configurations related to an application functionality, it is not sufficient enough to present the actual switching mechanism between the configurations. For this, we turn to the control concepts that we have present precedently in the chapter.

6.5.3 Modeling of mode automata

Previously in this chapter, the abstract control semantics have been presented with clear syntax and semantics specifications. However, as Gaspard2 is a graphical SoC Co-Design framework with emphasis on describing a system with the help of modeling specifications, these control concepts should also take a graphical form in alignment with other concepts used in the Gaspard2 framework, while respecting the UML MARTE profile.

As previously stated, Gaspard2 adopts the component based approach in compliance with MARTE. The interfaces associated with a component indicates how this component interacts with external environment. UML behavioral state machines can be associated with components and we have introduced similar semantics in the MARTE metamodel. However, these state machines work on *attributes* and *operations* of a component in preference to its associated ports, which differentiates behavioral state machines from UML protocol state machines. Also, Gaspard2 is dedicated to the specification of DIP applications, whose nature is different from the event-driven nature of UML state machines. UML state machines are also different from mode automata for the same reason.

We now present the graphical modeling of the previously mentioned control semantics. The specific usage of UML and consequently MARTE profile will not change the semantics of state machines or collaborations. However, their semantics are changed under some conventions. The result of this change can be considered as a variant of UML state machines, termed as Gaspard state graphs as specified in section 6.1.3.2.

It should be made evident that the modeling of the mode automata is different from the modeling of a Gaspard2 application. While both provide a structure for the application and control respectively, the similarity ends there. During the design phase, the application model can be created independently and thus subsequently deployed with absence of control semantics, resulting in a static implementation. While the control model is directly dependent on the deployment specifications; and results in introduction of dynamic characteristics. In this dissertation, the visual illustrations of the modeled control and application models are represented differently, to emphasize their different natures.

6.5.3.1 Gaspard State Graph Component and State machines

Respecting the control semantics introduced previously, a MARTE `StructuredComponent` which is associated with a Gaspard2 state graph is termed as a `Gaspard State Graph Component` (GSGC) and reacts to external events. It is always considered as a controlling component (as it is required to produce output mode values for other components). A GSGC as seen in Figure 6.30 is an implementation of its associated state graphs, which in turn give a precise external view with regards to the ports of the GSGC.

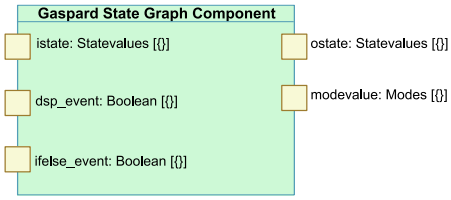


Figure 6.30: UML representation of a Gaspard State Graph Component

The interfaces of the GSGC are represented by *ports* and stereotyped accordingly as MARTE *FlowPorts*. The shape of a port indicates the number of values arriving at a port at one instant of time. As this proposal takes control/data flows into consideration, the shape corresponds to one reaction of the state machine and is always mono-dimensional: having a value of {}. The input ports of a GSGC can be either *event* or *state* ports. Event ports serve in triggering a transition in the associated state graph and are normally of type Boolean. An event used in Gaspard2 for triggering a transition is generally a *ChangeEvent* [185]. This event has a *changeExpression* which is a Boolean expression that can result in an event change. The second type of event is the *AnyReceiveEvent* which is considered as a default event when all the triggers of the transition of a state are not satisfied. They are expressed as the *all* statement in the modeling of the Gaspard state graph.

Values associated to state ports are termed as *state values* and identify the different states in the state graphs. The input state port for a GSGC indicate the initial state upon entering the GSGC. For a hierarchical state graph, multiple initial states may be required. This issue is discussed in detail in [104], and is out of the scope of this dissertation, as we deal only with non hierarchical state graphs. A GSGC also supports two kinds of output ports: *mode ports* and

state ports. The state graph associated to the GSGC carries out transition functions on the states and each state is associated to one mode. The output mode port thus carries mode values to the `Mode Switch Component`; which are determined by the transitions of the state graphs. The output state ports are similar to the input state ports and provide next state of the GSGC (the next state after a transition). The mode values are defined in the deployment modeling phase and are equivalent to the ConfiguratonID attribute related to a `Configuration`. These mode values are independent of GSGC and its associated mode switch component for re-use purposes. As a result, a Gaspard state graph component can be replaced by another GSGC, provided that the replacing component follows the control and deployment level semantics and is still compatible with the predefined mode values.

The two enumeration `Statevalues` and `Modes` related to the state and mode ports of a GSGC are not illustrated here. The first enumeration contains as enumeration values, the states defined in the associated Gaspard state graph. While the second enumeration contains the collection of mode values possible, i.e., it contains the different `ConfigurationIDs` related to the different configurations.

Figure 6.30 shows the Gaspard State Graph Component for the previously modeled application. This component shows only the external interface of the controlling component. It also contains two input event ports `dsp_event` and `ifelse_event` which are used to trigger transitions. The `iState` is used to indicate which state of the state graph is the source state. The target state and mode are produced through `ostate` and `modevalue` ports respectively. Each of the ports is mono-dimensional in nature as indicated by the shape value of {}.

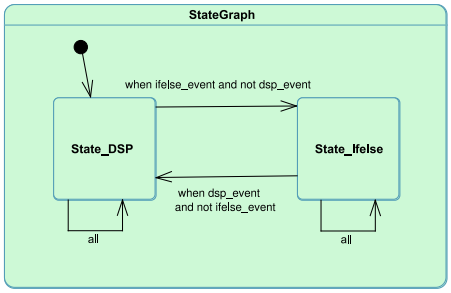


Figure 6.31: A Gaspard state graph associated with a Gaspard State Graph Component

Figure 6.31 illustrates an example of a Gaspard State graph associated with the GSGC for the modeled application. This state graph has two states: `State_DSP` and `State_Ifelse`. One of these states is connected to the initial pseudostate, indicating that this state is the initial state. Some transitions connect these states, which can be fired by triggers on the transitions. Triggers are defined on events, which are Boolean expressions of the event port variables (`dsp_event`, `ifelse_event`) of the Gaspard state graph component. The condition *when ifelse_event and not dsp_event* represents a trigger associated with a transition. The events associated to the event ports are of the type `ChangeEvent` which can result in an event change. However, an event may arrive on an input event port that does not fulfills the condition for triggering a successful transition. These kinds of events are called `AnyReceiveEvents`, and are considered as default events. They thus help to model a self transition in a state. They are modeled as the `all` statement in Figure 6.31. An example of such a case can be when both events (`dsp_event` and `ifelse_event`) arrive simultaneously on the input ports of a GSGC. Depending upon the actual state at the moment, a self transition takes place.

A state has a specific `doActivity`, where we can specify a behavior carried out in this state. This behavior can be specified as an `OpaqueBehavior`, which can be described in a natural language. Due to the current limitation of the modeling tools, it is not possible to visualize this behavior. In Gaspard2, we use this `OpaqueBehavior` in natural language to specify values that are sent to output ports when the state is active. For instance, in Figure 6.30, the GSGC has two output ports: state port `ostate` and mode port `modevalue`. The first one indicates the current state of the state machine and second one indicates the mode to execute at this moment. The `doActivity` related to a state can be specified as:

region.ostate=self.name and app.modevalue=Configuration.ConfigurationID

where the left part of the statement concerns the port and the right part concerns their values. For instance, *region.ostate* and *app.modevalue* are port names of the component. Here *region* is the name of a region, which owns the current state. Also, *ostate* is a string that denotes a state port. The *app* represents the name of a component that requires this mode value. Finally *modevalue* is also a string that denotes a mode port. On the right-hand side of the expression, *self* indicates the state itself. This value can be replaced by either *super* or *sub* values to distinguish a containing state or a sub-state, in case of state hierarchy. As we focus on flat state graphs, we always use the self value that indicates the name of the state itself. *ConfigurationID* is the unique ID assigned to a `Configuration` which is defined in the deployment model. For example, the *doActivity* of the *State_DSP* state in Figure 6.31 is set as:

region.ostate=State_DSP and Gaspard_State_Graph.modevalue=DSPMode

6.5.3.2 Mode Switch Component and Collaborations

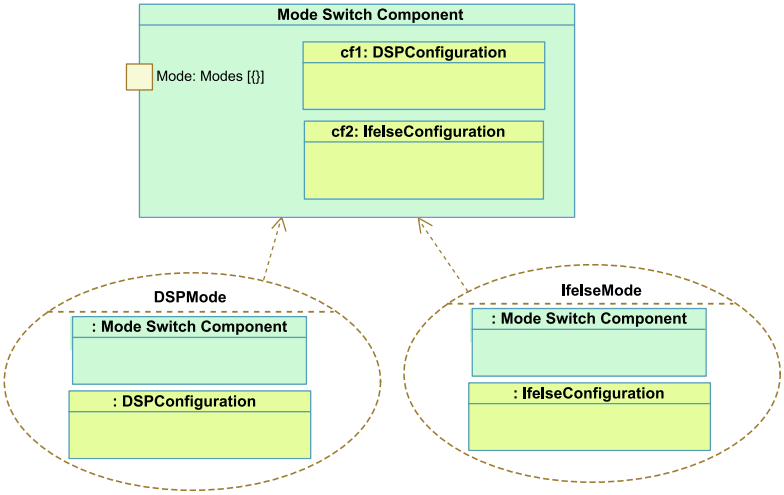


Figure 6.32: A mode switch component and its associated collaborations

A `Mode Switch Component` (MSC) is associated with *collaborations* and serves to switch between the different exclusive present modes. The MSC has an input mode port which obtains the mode values from the GSGC. The MSC acts on these mode values and executes the corresponding mode. It has at least one mode, and only one exclusive mode can be selected at time *t* depending upon the mode value present in the input mode port at that instant.

While the structure of the MSC can be defined using the MARTE general component concepts as defined in the *Generic Component Modeling* (GCM) package, the behavioral semantics of an MSC and the internal collaborations of its internal parts are not evident. For this reason, `Collaborations` are associated with a MSC. These `Collaborations` specify roles of components (instance level collaboration) via usage of connectors and parts in composite structures. A collaboration specifies the relation between some collaborating components (or roles). Each of these roles provides a specific function, and executes some required functionality in a collective way. Only the concerned aspects of a role are included in a collaboration while others are omitted. Figure 6.32 shows an example of a MSC for the modeled application described in section 6.5.1. The two collaborations depict the behavior of the MSC `Mode Switch Component`. The name of the collaborations correspond to the mode values and thus these collaborations define the activity of the MSC upon receiving a particular mode value. For example, the collaboration `DSPMode` shows the relationship between the MSC and the mode/configuration *DSPConfiguration* (indicating that mode value *DSPMode* switches the current executing mode to *DSPConfiguration*). As in this mode only *DSPConfiguration* is to be executed, the second mode *IfelseMode* is omitted along with the mode port of the MSC, due to the semantics of UML collaborations. The collaboration is finally linked to the *Mode Switch Component*.

6.5.3.3 Creation of a Gaspard Mode Automata

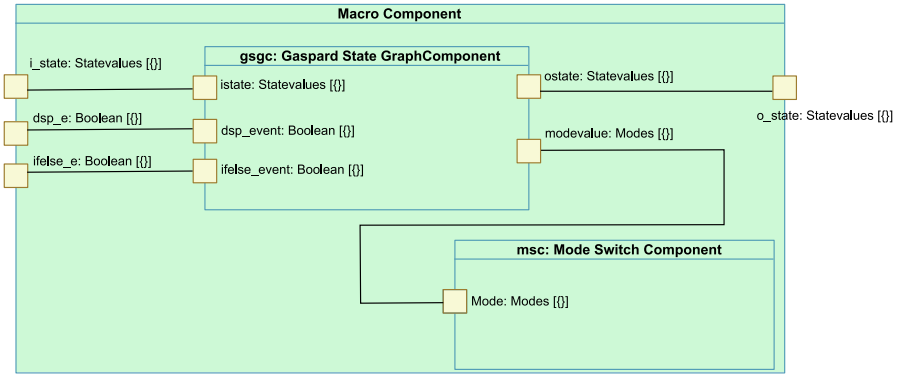


Figure 6.33: Modeling of the macro component

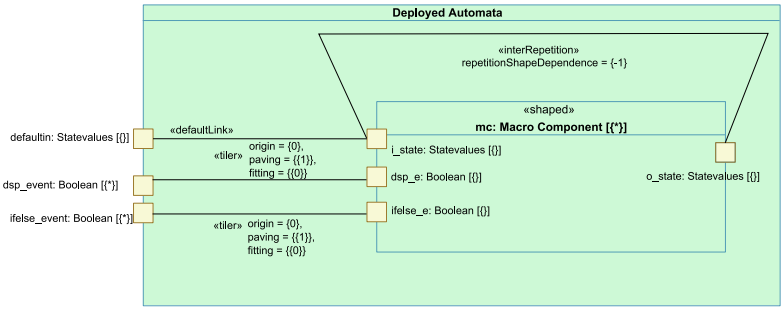


Figure 6.34: Modeling of the deployed mode automata

To create a mode automata at the deployment level, first its internal structure: a composition of a GSGC and a MSC, is constructed. The GSGC produces mode values which are taken by the MSC, which executes a switch function between the modes present in the MSC. Compared to purely synchronous mode automata, the computations are not set in the states of a state machine (or a state graph), but are placed in the MSC. As specified in section 6.1.3.4, this composition is termed as a `Macro Component`. Finally, Figure 6.33 shows the graphical representation of the macro component while omitting the collaborations for the sake of simplicity.

Afterwards, the macro component is then placed in a repetition context termed as a `Deployed Automata` with a shape value corresponding to {*}. In a `Deployed Automata` component, a `Macro Component` can be executed in parallel (each repetition is independent of other GSGCs; and a GSGC has no memory of the previous state as the inputs required by each successive repetition of a GSGC are only present at one time). A macro component can also be executed in a sequential manner (a dependency exists between the repetitions of a GSGCs allowing to introduce the concept of memory of previous states). In this dissertation, we only address the second approach. A dependency in the sequential execution is represented by the `InterRepetition` dependency (IRD). The `repetitionShapeDependence` vector associated to the IRD expresses the dependencies between the repetitions inside the `Deployed Automata`. If the depended repetition is not defined in the repetition space, a default value is selected by means of the `defaultLink` connector. The `Macro Component` should be placed in a `Deployed Automata` with at least one IRD specification. Figure 6.34 shows the modeling of the deployed mode automata concept.

6.6 Conclusion

In this chapter, we first presented the conditions related to our control semantics. Afterwards, the generic concepts related to control were explored at different levels in SoC Co-Design, namely application, architecture and allocation level. Control at these levels were compared with their advantages and disadvantages. Finally we propose the utilization of a local control at the deployment level of a SoC framework. This control respects the conditions that we have introduced earlier in the chapter and allows re-utilization of high level models. More over, in order to implement partial dynamically reconfigurable SoCs, the control concepts along with the deployment level semantics were integrated in the MARTE metamodel and profile.

This integration allows expression of dynamic aspects related to an application functionality via current UML modeling tools supporting the MARTE profile. The intended goal is to interpret the MARTE compliant UML model and its corresponding metamodel in order to transform it into an intermediate model (along with its corresponding metamodel) at a lower abstraction level, closer to an execution platform. The metamodel corresponding to this abstraction level, termed as the RTL metamodel is explained subsequently in the following chapter.

Chapter 7

A metamodel for targeting Register Transfer Level

In the previous chapter, we have detailed the initial concepts for integrating control in Gaspard2, for integrating dynamic aspects in SoC design specifications. The control concepts correspond to semantics related to a reconfiguration controller, responsible for managing the context switching. As specified earlier in the dissertation, another crucial part of a dynamically reconfigurable system is the region selected for an effective swap during the reconfiguration. In our design methodology, this dynamically reconfigurable part corresponds to a hardware functionality; which is equivalent to a modeled high level *deployed* Gaspard2 application. The conversion of this modeled application into a hardware functionality is due to the presence of an intermediate metamodel (and corresponding namesake model) that enriches the specified application for eventual code generation: namely the *Register Transfer Level* (RTL) metamodel.

This chapter introduces the RTL metamodel in the Gaspard2 framework and is a focal key point of the design methodology presented in this dissertation. The RTL metamodel corresponds to an intermediate level between the description of a modeled Gaspard2 application (specified using the MARTE profile) along with its consequent deployment and control specifications at the high abstraction levels; and the automatic code generation, permitting final execution in a targeted FPGA for executing partial dynamic reconfiguration. Due to the presence of control semantics introduced in the deployment level, the transformed hardware functionality has multiple associated implementations, equivalent to the high level modeled configurations, enabling the creation of a dynamically reconfigurable hardware accelerator. In that effect, the RTL metamodel takes into account the deployment level details related for this translation.

The RTL metamodel can be viewed as a collection of different concepts, as shown in Figure 7.1. The metamodel contains some common features or metaclasses, utilized for the specification of concepts related to the creation of a hardware functionality, while enriching control aspects introduced in the last chapter. In parallel, there exist specific metaclasses and metarelations, each corresponding uniquely to one of the above mentioned aspects.

For the description of the hardware functionality, the metamodel must be detailed enough in order for the model transformations to generate efficient synthesizable code for a target FPGA based SoC. For this, the chapter first provides a brief overview of hardware accelerators and

the different works realized for conception of these components. As stated earlier, since a Gaspard2 application is transformed into a dynamically reconfigurable hardware accelerator; in section 7.2, we present a detailed description of a hardware execution model for the applications modeled in the Gaspard2 environment. This execution model endows the related model transformations with clear guidelines, for converting the presented application into an equivalent hardware functionality.

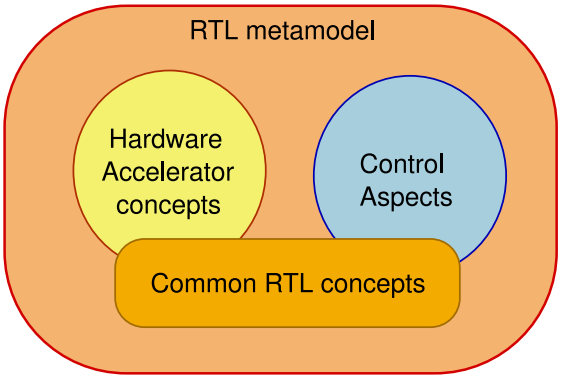


Figure 7.1: An abstract overview of concepts related to the RTL metamodel

However, as specified before in chapter 4, data parallelism in Gaspard2 is based on the semantics present in the MARTE RSM package, and its underlying model of computation: Array-OL. Hence several execution models are possible: such as *Single Program Multiple Data*[1] (SPMD), pipeline or sequential execution models. Nevertheless, normally, these execution models are mainly related to MPSoC architectures and not intended for development of hardware accelerators. While in [256], the authors present an Array-OL based hardware execution model, the proposal lacks aspects related to task parallelism. Additionally, in [143], the execution model based on Array-OL enables expression of both task and data parallelism, unfortunately, the works do not take aspects of dynamic reconfiguration into account and the intended generation of hardware accelerator is intended to be static in nature.

Hence, the hardware execution model presented in this chapter safeguards the potential parallelism of modeled applications while integrating reconfigurability features. From this execution model, we extract the concepts that serve as the basis of the RTL metamodel presented in section 7.3. This execution model is only concerned with the modeled application; and permits its transformation into a hardware functionality with dynamic characteristics. In parallel, the control concepts mentioned in chapter 6 are also integrated into the metamodel, for describing the semantics and syntax of a reconfiguration controller. This controller is eventually responsible for carrying out partial reconfiguration of the translated hardware functionality and its various implementations. Thus, the RTL control concepts are eventually utilized in eventual code generation for the reconfiguration controller.

7.1 Hardware accelerators

In traditional computing, hardware acceleration can be viewed as the utilization of hardware to execute the desired functions faster in a parallel manner, as compared to a purely software functionality which is executed sequentially, as illustrated in Figure 7.2. A hardware accelerator is a dedicated integrated circuit specialized for performing data intensive processing. It allows maximum parallelism of computations required for the execution of an application; and provides an optimal execution support for processing *regular* and *repetitive* tasks. Example of hardware accelerators can be found in state of the art *Graphical Processing Units* (GPUs), co-processors or modern FPGAs present in large complex SoCs.

In SoCs, these hardware accelerators are quite common for accelerating key kernel parts of the targeted applications. Normally, these SoCs have integrated FPGAs or are FPGA based

[1] http://en.wikipedia.org/wiki/SPMD

themselves, as detailed in chapter 1; resulting in two clear advantages. These architectures allow to express the potential parallelism required for high performance and data intensive SoC applications. While on the other hand, the reconfigurable nature of these architectures permit to introduce notions such as partial dynamic reconfiguration. This in turn, permits increased flexibility while keeping intact the aspects related to performance and parallelism, that are critical for these complex systems. We now provide a brief overview of several significant works developed for the creation of hardware accelerators.

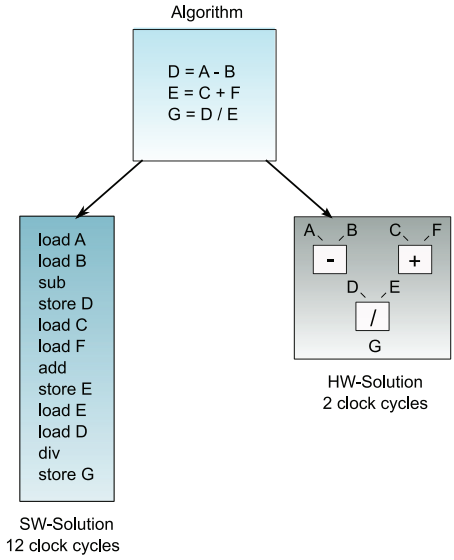


Figure 7.2: Software versus Hardware execution

7.1.1 Related works related to hardware accelerators

The usual method of describing hardware accelerators at the RTL level is carried out via *Hardware Description Languages* (HDL)s such as VHDL or Verilog. Numerous works exist based on manual hand tuned HDL based accelerators and are typically involved in improving performance of these accelerators. For example, in [247], a detailed summary about different conception methodologies related to hardware accelerators has been given. Similarly in [30], a FPGA based hardware accelerator implementation is proposed for a correlation module in an anti-collision radar detection system.

However, hand-tuned designs of hardware accelerators at RTL level are usually plagued with errors as they are directly dependent on designer expertise. Similarly, conception time for large complex designs increases exponentially, due to necessary validation of correct functionality of each sub component in an accelerator. However, in spite of all the difficulties related to their conception, hardware accelerators (such as FPGA or DSP based accelerators in SoCs) are frequently utilized, specially in the domain of intensive signal processing specified in section 4.1. Similarly, tool based approaches have been proposed which aid in conception of hardware accelerators as compared to direct manual RTL level implementations.

Creation of hardware accelerators by tools and methodologies. JHDL or Just Another HDL [26] is a design tool for specification of dynamic circuit layouts, using abstraction levels normally found in object oriented languages. The tool supports both external partial and full reconfiguration, however the configurations are normally identified manually. The design tool introduces a notion of an interface class, a *PRSocket* that illustrates the external interface of a dynamically reconfigurable module, similar to the concepts related to a partial reconfigurable region (PRR) in Xilinx methodologies. However, JHDL is a low level implementation tool.

For example, in [70, 162], generation of hardware accelerators at RTL level is proposed via ALPHA0 and ALPHARD languages respectively, which are offshoots of the ALPHA language

[155]. However, they do not take into consideration the FPGA resources necessary for final implementation, requiring commercial tools to determine these measures using floor planning related to final FPGA implementation. In [59], the authors illustrated the generation of VHDL code from Synchronous Data Flow (SDF) [144]. However, SDF only allows to express data dependencies on a single dimension; and the limitation is also evident in [59]; handling multi-dimensions impossible. SIMULINK provides an HDL CODER[2], for VHDL code generation for applications. While in [20], SYNPLICITY's SYNPLIFY DSP tool has been used in combination with SIMULINK for VHDL code generation. Computational expressions of multidimensional data along with the possibility of eventual code generation extends the usability of SIMULINK for intensive signal processing. However, the data dependencies are always expressed with the help of indexes, which is determined by the developer. This increases the chances of present errors in the design, when expressing data dependencies on multidimensional arrays.

In [256], the authors used the Array-OL language (the basis of the RSM package of MARTE) for writing applications for eventual hardware accelerator generation. Array-OL uses a factorized form for expressing data dependencies on multidimensional arrays and avoids the pitfalls present in the earlier approaches. The authors expressed data dependencies with mapping of VHDL ports. This method permitted to express data dependencies in a factorized form, but required a flat unrolled expression of the potential multidimensional ports on repeated tasks. Apart from this inconvenience, an application contained only a single component and notions of hierarchy and task parallelism are absent. Similarly data dependencies on time are not handled, as only simple connections are realized for managing data dependencies on space. [143] address these problems and propose a methodology for hardware based execution of applications where data and task parallelism are expressed via Array-OL. Data dependencies on time and space are both managed and applications can be hierarchic in nature. However, the VHDL code generated is not completely error free and the modeled applications are usually intended to be in form of VHDL black boxes; and cannot be directly synthesized using commercial FPGA tools. In addition, they do not take dynamic reconfiguration into account.

Dynamically reconfigurable hardware accelerators. In [64], the authors present a FPGA based reconfigurable hardware accelerator for solving Boolean satisfiability problems (SAT). They achieved a speed up of 3.7 to 38.6x compared to a modern CPU executing the same computations. While in [4], a dynamically reconfigurable hardware accelerator is used to accelerate key kernels (mainly the color conversion and 2D-DCT steps) of a JPEG encoder. Similarly in [196], neural network simulation has been carried out with the aid of multiple runtime reconfigurable accelerators. In [264], a dynamically adaptive reconfigurable loop accelerator is employed for unrolling loops to increase parallelism and subsequent performance of the system. However, greater reconfigurable area is required for higher degree of loop unrolling. Using partial reconfiguration, the authors illustrated a save of 93.6% of the resources at a compromise of 1.6% in performance. While in [238], the authors illustrated FPGA implementation of dynamic run-time reconfiguration related to behavior of robots. They made use of UML sequence diagrams for expressing fault tolerance and reconfiguration algorithms.

Augmenting the abstraction levels. Similarly, there has been a tendency to increase the abstraction levels for the generation of hardware accelerators using languages such as C/C++ or similar languages such as HandleC. The choice of a language is dependent on the user expertise. Numerous academic tools and commercial products have also been developed. On the commercial side, we found products such as CATAPULT C[3] and CODEVELOPER[4], while on the academic front, tools such as SPARK [103] and GAUT [57] are available.

A recent trend has been to use high abstraction levels based on MDE and UML for the generation of these circuits. For example, in [62], a VHDL metamodel has been proposed for eventual code generation purposes. However, this metamodel is strongly dependent on the syntax of VHDL and cannot be used for specification of other HDLs. In [58], XML based parsing methodology is used for the generation of VHDL code such as state machines from equivalent

[2] http://www.mathworks.com/products/slhdlcoder/
[3] http://www.mentor.com/products/esl/high_level_synthesis/catapult_synthesis/
[4] http://www.impulseaccelerated.com/products_universal.htm

UML modeling. A similar approach has been proposed in [213], using XLST and XMI parsers and the generated code was implemented on a Xilinx Spartan series FPGA. Several other works have been developed to generate VHDL code from UML state machines. In [10], the authors propose an exploratory study for using model driven methodologies in order to develop a compiler that transforms UML state machines to VHDL code. For this, they present state machines and VHDL metamodels, along with MDE model transformations. Works such as [254] use a mapping approach to create relationships between the concepts present in VHDL and UML. Finally in [61], the authors used an intermediate language for describing automata. The automata described are generated from UML statecharts and allow eventual VHDL code generation.

There exist a large number of tools, but our intention is not to give a detailed description of each, but to give a general overview. Comparison of these tools is difficult as they evolve rapidly. Moreover, utilization of C/C++ for application description, for eventual translation into hardware functionality is viewed as inconvenient as it raises different issues. As described in section 3.1.1.2, the description is mostly text based in nature and system hierarchy and parallelism is not evident. Hierarchy is usually in the form of functions where as data parallelism is in the form of nested loops. A graphical representation does not cause these inconveniences. Similarly, potential parallelism of an application is usually expressed in C/C++ in the form of sequential loops, which is not an easy task for the designer. A brief summary regarding disadvantages of these approaches has been presented in [143]. As evident from the above mentioned approaches, the domain related to hardware acceleration is extremely vast; and it is not possible to give a detailed description in this document. Here, we have only covered some design methodologies for the development of these integrated circuits. Dynamically reconfigurable hardware accelerators are being increasingly utilized in the SoC industry due to their aforementioned advantages, and are addressed in this dissertation. We now look onto the hardware execution model for a Gaspard2 application, permitting creation of a dynamically reconfigurable hardware accelerator from our design flow.

7.2 Hardware execution model for Gaspard2 applications

In section 4.2.3, we have already illustrated that Gaspard2 applications are modeled independently from low level implementation details, until the deployment phase. For generation of a dynamically reconfigurable hardware accelerator, our design methodology transforms an UML model (of a control integrated high level deployed application, respecting the MARTE profile) into a RTL model conforming to the RTL metamodel. Correct specification of the RTL metamodel is a critical aspect, in order to identify the key concepts related to the hardware execution of Gaspard2 applications.

An example of a Gaspard2 application has been earlier presented in section 6.5.1. A partial extract of this application is illustrated here to illustrate the concepts related to hardware execution in our design flow. Details related to the actual operations of this functionality will be presented in chapter 9.

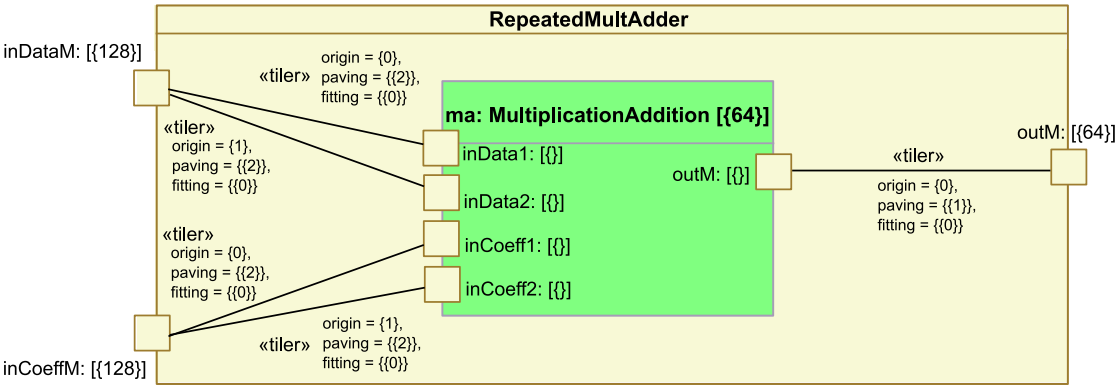


Figure 7.3: Partial extract of the Multiplication-Addition application

The partial extract of the multiplication-addition application as shown in Figure 7.3 can be itself treated as an application functionality. The Figure 7.4 represents the same application task in an flat unrolled manner while expressing data parallelism in a factorized form.

Here in the figure, the source data and coefficient arrays are present in the left part of the illustration, while the output array is present at the right side. The `MultiplicationAddition` repeated task is repeated on the basis of the value of its repetition space: i.e. 64 times. Each of the input data/coefficient ports of an instance consumes a data (or coefficient) of the respective input arrays, depending upon the associated index value; determined by tiler connectors containing information related to origin, paving and fitting. For example, the 0_{th} instance of the `MultiplicationAddition` component consumes the first two data and coefficients present in the respective input arrays. Similarly, for all the repetitions of the RT, the output port produces a single output element which is then taken by a tiler connector to construct the final output array.

In order to facilitate the visualization of the data dependencies between repetitions of the elementary task (RT) and the patterns they consume, all repetitions of the RT are colored in the same manner. The colors and index values on the ports of a repetition determine their positions in the input/output arrays. The data dependencies expressed by MARTE tiler connectors, form an interconnection topology for linking the respective arrays to associated patterns.

In [143], the authors identified two main types of effective hardware executions, possible for the Gaspard2 applications: namely a *Parallel execution* or a *Sequential execution*. Both executions allow to transform an application into a hardware functionality for eventual integration/synthesis in a target architecture. With respect to dynamically reconfigurable accelerators, a parallel execution reduces design complexity as compared to a sequential one; as described later on in the chapter (section 7.2.1.4). During the course of this dissertation, the parallel execution model has been selected and is now described in detail along with its advantages over the sequential approach. It should be made evident that this parallel execution model is only concerned with the application part of the RTL metamodel; as compared to the control which carries out execution in a sequential manner. This point is explained later on in section 7.2.1.4.

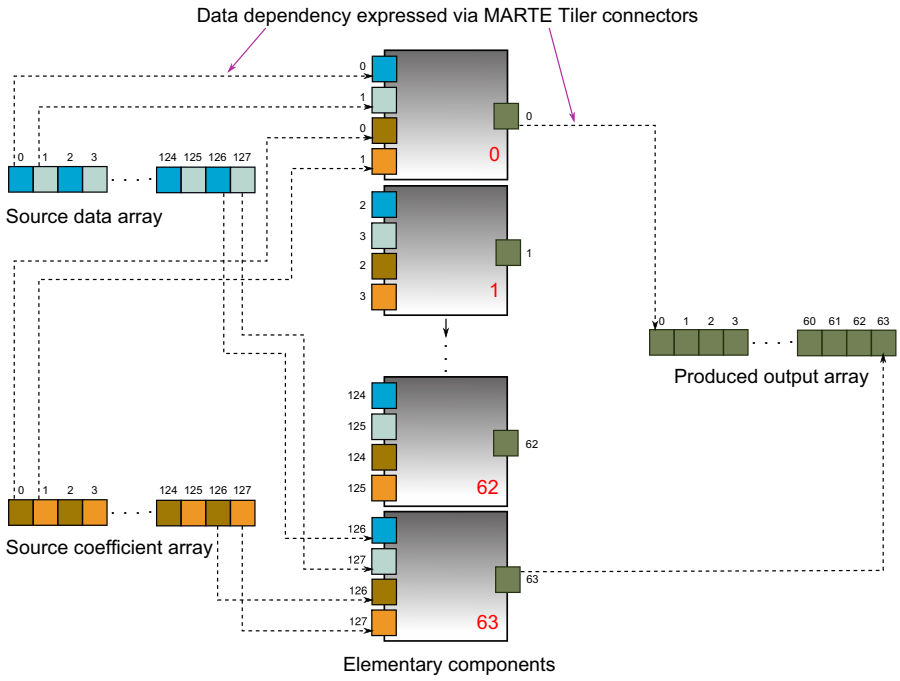


Figure 7.4: An abstract flat unrolled representation of the modeled application: the MARTE tiler connectors express the data dependency between the input/output arrays and patterns; consumed and produced by different repetitions of a RT, in a RCT

7.2.1 Parallel execution of hardware accelerators

7.2.1.1 Task Parallelism in hardware accelerators

As shown in Figure 4.9, Gaspard2 exhibits task parallelism in the form of an acyclic dependency graph. Every task in a Gaspard2 application can be executed in hardware by means of a hardware computation unit (HCU), implemented in the form of a component[5]. Several tasks can thus be executed in parallel in a hardware accelerator by equivalent number of HCUs. The input/outputs of the tasks are the data arrays; and the connectors connecting the different tasks (by connecting their inputs/outputs) represent the data dependencies. This same structure can be presented in a hardware functionality for expressing task parallelism: the HCUs are connected by means of connectors[6], and thus a task pipeline is created when a connector exists between the outputs of one task and the inputs of the subsequent task.

At the RTL level, the HCUs can be synchronized by means of a clock, and their data arrays are generated/computed at each clock cycle and directly read by other HCUs by means of connectors. Thus a data stream is established that does not requires intermediate memories (hence no data linearization is necessary for memory access). The quantity of resources necessary for the implementation of an accelerator vary with the complexity of an application. A complex application requiring a large number of resources may not be implemented if the resources present in the target integrated circuit are inadequate. In that case, it is preferable to either change the global structure of the application [31], or partition the application into several sub tasks for implementation in different accelerators or standard architectures such as MPSoCs.

In terms of a hardware accelerator, pipelined execution of tasks permit to increase the operational frequency while decomposing the critical path. For that this execution is really pipelined, registers having the same clock rate should be introduced in different data paths. Elementary components containing registers in their data paths have been introduced in [143], permitting generation of a data stream in the data paths of task parallelism. This stream implies that several tasks are executed at the same time, but each task iterates on a different instant on the temporal dimension of their repetition space. However, this pipeline introduces a latency in the production of output arrays of a task as it is necessary to fill the pipeline.

However, the limitations related to the utilization of the pipelined approach in task parallelism are evident. Different data dependencies are possible in a pipelined approach as evident in Figure 7.5. Thus synchronization barriers may be required for a pipelined execution. It is thus up to the designer of the application to guarantee that for a dependency graph, its corresponding implementation not does de-synchronize the computations.

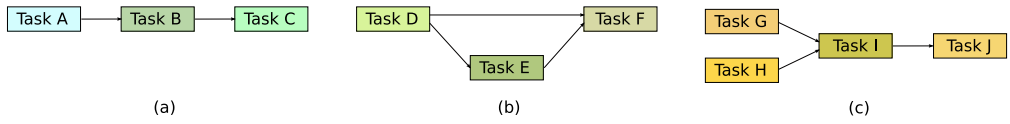


Figure 7.5: Different types of data dependencies in task pipeline

7.2.1.2 Data Parallelism in hardware accelerators

Gaspard2 applications are also able to express data parallelism as illustrated in Figure 4.10. The data parallelism in a repetition context task (RCT) can be present in different forms: the repetition space related to the interior repeated task (RT), the shape of the input/output patterns related to the RT, the compact data dependency expressions determined by the tiler connectors; and the shape of the input/output data arrays of the RCT. The hardware execution model takes all these information into account. The tiler connectors express complex data dependencies between the different iterations of the executed RT and the data arrays of a RCT. In the hardware accelerators, the tilers do not exist in their compact factorized form, but are also compiled to hardware components that illustrate the data dependencies exhibited by the tiler connectors,

[5] Thus a modeled application component is translated in the form of an HDL component with respective equivalent interfaces

[6] Signals or Nets depending upon the syntax related to a particular HDL

in the high level UML model. This step has been termed as *ADO pre-computation*, because the data dependencies are computed at design time; before the synthesis and implementation of the hardware accelerator on a target FPGA, for the eventual execution of the modeled application.

The repetition space of a repeated task does not force any execution order on the number of its associated repetitions. The execution model permits to execute the data parallelism in two manners as described before: either in a parallel manner or a sequential one. The parallel execution enables generation of an accelerator having increased performance, while consuming an increased amount of reconfigurable resources. While comparatively, a sequential accelerator takes less amount of reconfigurable resources at a compromise of reduced performance. Both types are able to handle data dependencies on time and provide adequate semantics for their implementation in the generated accelerator. This process related to data dependencies is further detailed in chapter 8. Finally, as described before, this dissertation takes only parallel execution into account, in the context of partial dynamic reconfiguration.

Spatial-Temporal mapping The parallel execution of data parallelism requires a flat design of the hardware accelerator[7]. This enables execution of different repetitions (RT) of a RCT at the same instant of time (in the same clock cycle). Thus N HCUs are needed for the parallel execution of a repeated task having a repetition space value of N. While Gaspard2 is able to represent spatial as well as temporal dimensions using the RSM package, there is an exception related to execution of an infinite temporal dimension, on a repetition space in the corresponding hardware execution model. An infinite repetition does not makes sense in the context of a hardware execution, and is in turn adequately translated. This repetition is executed on the basis of the clock related to the hardware accelerator (or a global architecture such as a SoC containing this accelerator). Each new clock cycle causes a new iteration on the temporal repetition space of the given application.

In simpler words, the clock permits sequential execution of temporal repeated tasks. A similar principal is applied on the ports of a modeled application when one of the associated dimensions has an infinite value. As equivalent hardware ports do not have an infinite dimension, the dimension determines a data stream. The overall result is the creation of a hardware execution model of a mixed nature. The top hierarchical level of the hardware functionality is executed sequentially at each clock pulse[8], while the subcomponents at lower hierarchical levels are executed in parallel. This is the case for the application modeled in Figure 6.28.

Related to partial dynamic reconfiguration, the generated hardware accelerator is placed in a system having a common base clock. The hardware accelerator is clocked to a unique frequency. This implies that the data present in the ports are consumed at the same rate as the execution of tasks is carried out. This statement is validated in chapter 9, when we illustrate the simulation results related to our case study.

However, the execution model has its limitations; and all the potential data dependencies that can be expressed in Gaspard2, via the RSM semantics; cannot be taken into account.

7.2.1.3 Hierarchy and ADO pre-computations in accelerators

The presented hardware execution model can handle hierarchy in accelerators, can be composed of either task or data parallelism or a combination of the two. This is possible because the chosen parallel execution model forces all computations at the same clock cycle at the cost of increased number of consumable FPGA resources. With regards to data dependencies, the ADOs which are tiler connectors, compute these dependencies by means of the underlying semantics of Array-OL [36, 37].

In a hardware accelerator, each data dependency is expressed by means of a hardware signal or a shift register, as explained in the next chapter. It is thus possible to implement these dependencies using existing resources, hence reducing the overall consumed resources. The pre-computation of these tilers can be used for resource optimization, for e.g., in the construction of a pattern related to a RT, or construction of all patterns in a RCT. In order to provide

[7] This is to say that all repetitions of a RT should be unrolled

[8] In an HDL such as VHDL, an infinite repetition is implemented using the GENERATE keyword, with lower and upper bounds set equally to 1. This point is further explained in the next chapter

a general mechanism for expressing the data dependencies, the modeled tiler connectors are transformed themselves in respective HCUs that carry out the pre-computations. The process related to the pre-computations is detailed in the next chapter.

Figure 7.6 illustrates an abstract representation of the electronic circuit of the application task represented in Figure 7.3. The left side of the figure represents the input ports of the accelerator corresponding to input data/coefficient arrays of the modeled application (part *A*). Similarly on the right, the output ports represent the results, produced by the repetitions of the interior repeated task in an accelerator (part *B*). These ports are connected to components representing the different repetitions of the RT (part *C*). Each repetition is connected to the five tiler components representing the input/output tiler connectors in the initial modeling using the MARTE profile. The input tilers (part *D*) are connected to read the input arrays and produce the patterns consumed by the different repetitions of the RT in the repetition space of a RCT. These repetitions are executed in parallel and are repeated 64 times for this example. Each repetition produces an output pattern which is then transmitted to the output tiler (part *E*) responsible for reconstructing the output array.

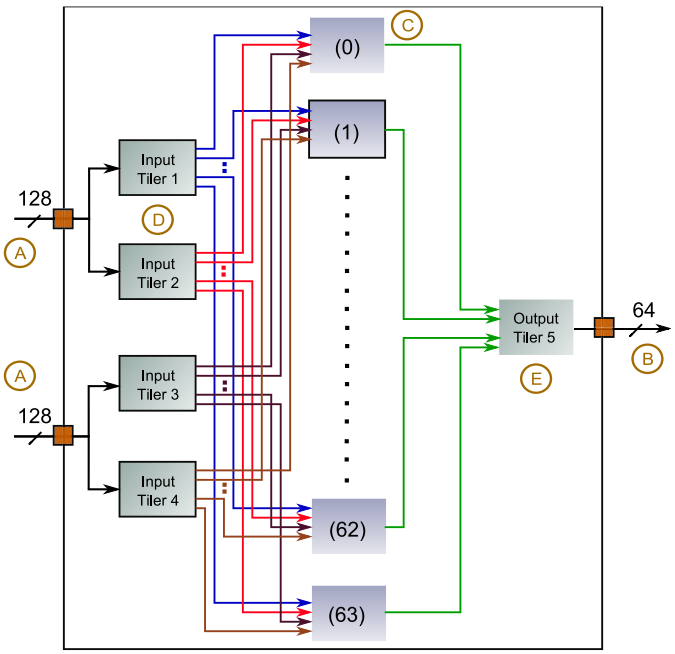


Figure 7.6: Abstract representation of parallel execution of the modeled application task in an electronic circuit

It should be observed that the modeled application task itself is a subcomponent in the application represented in Figure 6.28. Thus this parallel execution of the hardware accelerator could be treated as an intermediate hierarchical level in the final hardware accelerator produced for the global application. In the case illustrated in this chapter, we have treated this application task as a separate application, which however does not contain an continuous infinite repetition as the modeled application in Figure 6.28; that illustrates a continuous data stream of consumed and produced data arrays; as well as a sequential temporal execution.

7.2.1.4 Parallel or Sequential execution ?

The previous section introduced the hardware execution model for the Gaspard2 applications specified with the MARTE profile. The execution of data parallelism related to the applications can be either sequential or parallel in nature. Parallel execution permits to increase the performance of the hardware accelerator and reduces latency; at the cost of reconfigurable resources such as CLBs in a targeted FPGA. All the tasks are executed at the same clock cycle and data dependencies are resolved by tilers components.

The sequential execution of data parallelism introduces increased complexity in the overall design. For a sequential execution, the different iterations of a repeated task in a repetition space are not executed in parallel but in a sequential manner. Thus for the same application shown before which executed in 1 clock cycle, 64 clock cycles are needed for a complete sequential execution (one clock cycle for each iteration). Thus several clock cycles are needed to cover the repetition space of the highest hierarchical level of the application. Similarly, additional elements such as controllers, multiplexers and demultiplexers are needed for this execution [143]. While the overall cost related to a sequential execution is considerably less as compared to a parallel execution, it introduces additional latency.

For systems dealing with partial reconfiguration where time required for a reconfiguration is a significant criteria, and needs to be optimum; this can be viewed as a critical drawback. However sequential execution of hardware accelerators may be favored in some systems having limited resources; and where reconfiguration time does not have a high priority as compared to consumed resources.

Currently the hardware execution model offers a compromise. An application having a infinite dimension at the highest level of hierarchy executes the different iterations in a sequential manner; while lower levels are executed in parallel. This results in a parallel execution of temporal repeated tasks. It should be made evident that the infinite dimension can only be present at the highest level of hierarchy, as its presence in lower hierarchical levels denote the existence of multiple clocks with different clock rates. This could cause a de-synchronization in the generated hardware functionality.

The mixed sequential-parallel execution of the hardware execution model offers another advantage. As seen in section 6.2.1, the control model is sequential in nature. In the RTL execution model, this could permit a single clock to handle both the hardware accelerator computation and the transitions of the mode automata control. Thus for a clock pulse, data is produced and consumed by the different parallel parts of the presented application, while the control can carry out a self transition or transition to another state. However, additional latency has to be taken into account in an execution platform, related to the time taken for propagation of a configuration switch command by a reconfiguration controller and the actual switching. Delays related to task parallelism also should be taken in consideration. Similarly synchronization between the control/data flow values related to the mode automata have to be taken into account. Some of these problems have been addressed in chapter 9.

Finally, the last difference between the hardware accelerator and the control aspects in the RTL metamodel is that hardware execution model only takes data flow into consideration, as compared to control, that manages the control flow.

7.2.2 Interrepetition and defaultLink in RTL metamodel

As defined before, an interrepetition dependency specifies an acyclic dependency among the different repetitions of a repeated task, leading to a sequential execution. A defaultLink provides an initial link for the repeated task when the source link to the first repetition is absent. Similarly it can be used for the final link for the last repetition.

The statement defined above can be explained clearly with the help of Figure 7.16. It shows a repeated task in a RCT with a repetition space of {3} along with pattern shapes for its input and output ports set to be mono dimensional: having values of {} respectively. The value of {−1} is related to the *repetitionShapeDependence* vector for determining the *dependency* between the repetitions; and is thus interpreted differently as compared to usual shape values on ports and RTs illustrating an infinite repetition. An interrepetition and defaultLink as a tiler, are transformed into equivalent RTL components (`RTL_Interrepetition` and `RTL DefaultLink`) by means of the model-to-model transformations. However, these components are treated differently, as in our dissertation, they are related only to the control part of the RTL metamodel, and their code generation is discussed in the next chapter. However, in the subsequent paragraphs, we provide a justification of this approach, as their utilization in the creation of a hardware accelerator raises several issues.

Disadvantages for a dynamic hardware accelerator. For a RT having a finite repetition space, the shape of the source and target ports of an interrepetitive hardware component can be set to the product of the repetition space and the pattern shape of the RT, respectively. This expression is also valid for a RT having multidimensional shape values on its input/output ports or repetition space. Respecting the semantics related to interrepetition, the shape values on the input/output ports of the RT should be identical. Thus for the example in Figure 7.7, the interrepetition component has 3 source and target ports. As the last iteration of the RT is not connected to a defaultLink, the last port of the interrepetition component is not connected and is therefore not shown.

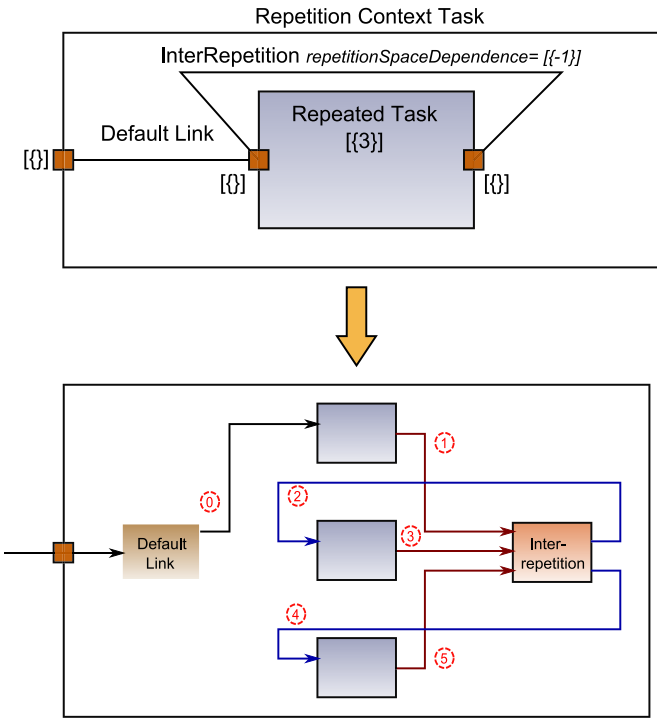


Figure 7.7: Abstract representation of mechanism related to interrepetition and defaultLink

The numbers in the bottom side of the figure explain the sequential order of the data flow related to the RT, the interrepetition and defaultLink components for a hardware execution. The interrepetition component helps to determine the data dependencies between the different iterations of the RT over time. However, this dependency forces to unroll the loop in case of a finite repetition space associated with a RT, as shown in the figure. Thus here, 3 repetitions of the repeated task are unrolled forming a chained pipeline, resulting in increased FPGA resources. This interrepetition component can be associated with a buffering mechanism to store the data between two successive repetitions of a RT.

A defaultLink connector can also be transformed into a RTL component as well. The dimensions on the source port of a default link should be equal to the shape of the input array, while for the output port, they are equal to the shape of the pattern shape of the RTs. We place a constraint that the shape of the input array and the consumed pattern should be equal for correct functioning of a defaultLink.

The semantics that we have introduced related to interrepetition and defaultLink can be used for either the hardware accelerator or the control concepts. However, the combination of interrepetition and defaultLink force a dependent sequential execution of the related RT, and in section 7.2.1.4, we have already stated the advantages of using a parallel hardware execution model for Gaspard2 applications. The only advantage of these concepts is from a modeling perspective, as they allow a designer to specify an explicit depended sequential execution of the application at the high abstraction levels; as compared to ambiguous choice (parallel, sequential or mixed sequential-parallel) related to a hardware execution in our design flow.

However, it should be mentioned that in an extended version of our design flow adapted to a sequential execution model, the sequential hardware execution of the repetitions of a RT will be not dependent on each other. For example, in a sequential execution of the application task in Figure 7.3, an i_{th} repetition does not depend upon the output of the $i{-}1_{th}$ repetition as its input, as compared to an sequential execution forced by the interrepetition dependency; with different iterations strongly dependant on each other. Additionally, an infinite repetition related to an interrepetition component is not possible to be translated into a hardware execution model.

Even if the repetition is finite in nature, the sequential execution requires a buffering mechanism to store the outgoing data produced by the sequential execution of the RTs. Similarly, multiplexers, demultiplexers and controllers may be required as well, that are not illustrated in the figure. For this reason, the concepts introduced in this section are currently only utilized for control modeling and in the construction of mode automata. However, these concepts can be enriched at a later date, for a dependent sequential execution of the hardware model with RTs having a finite repetition space. Currently, during the final model-to-text transformations, the interrepetition and defaultLink components in the RTL model are converted into variables which help to determine the initial, current and next states of an automaton.

7.3 RTL metamodel

An initial version of the RTL metamodel has been proposed in [143], however as explained before in the chapter, the metamodel does not integrate dynamic aspects, and is intended for creation of a single static hardware accelerator for final implementation as a black box. The current version of the RTL metamodel developed during this dissertation permits integration of dynamic features, for facilitating the creation of a dynamically reconfigurable hardware accelerator. The metamodel takes as input, a collection of concepts related to hardware execution model of Gaspard2 applications described earlier in section 7.2, along with control concepts introduced in chapter 6. In short, the generated hardware accelerator (and all its available implementations) exhibits characteristics: such as hierarchy, data and task parallelism specified in the high level UML application model. The part of the metamodel related to the hardware accelerators is independent from syntax related to any specific HDL, yet its low abstraction level enables code generation for a desired HDL. Similarly the enriched control concepts can either be interpreted for generation of HDL code in case of an HDL based hardware controller module; or a high level language such as C/C++ for implementation in a microprocessor based controller. As this dissertation focuses mainly on internal self dynamic partial reconfiguration, a microprocessor based solution is adapted, as discussed in chapter 5.

This section does not illustrates *how* generation of the hardware accelerator and control is carried out via the RTL model and its corresponding metamodel, but focuses mainly on *what* can be described via the RTL metamodel. Code generation is carried out by the *RTL2CODE* transformation (briefly summarized in chapter 6), for the hardware accelerator and the reconfiguration controller; and is detailed in chapter 8. Here, we first mention some initial motivations for the development of the RTL metamodel, followed by the description of the metamodel itself.

Independent nature. Firstly, the RTL metamodel possesses all the normal advantages exhibited by metamodels in general. This is to say that this metamodel is generic in nature, and enables re-utilization of clearly defined concepts (metaclasses) and their relationships (metarelations). However, in the context of the Gaspard2 framework, the RTL metamodel is an intermediate level between the UML model of a control integrated deployed application and the eventual code generation phase, as indicated in Figure 5.2; in turn permitting to separate the compilation process. Thus the RTL model is independent of a target language (specifically C/C++ for the control aspects and HDL languages such as VHDL or Verilog for hardware accelerator/control).

Influence of the MARTE metamodel. The RTL metamodel inspires from the MARTE metamodel itself. Concepts found in the RSM package related to multidimensionality, as explained in chapter 4, have been integrated into this metamodel. Similarly concepts such as components, ports and connectors found in the MARTE GCM package have also been translated into

near equivalent metamodel elements. However the MARTE metamodel (or even our extended MARTE metamodel) does not provides detailed semantics for the generation of an integrated circuit at the RTL level. Description at RTL requires enriched details related to execution platforms which do not; and should not exist in the high level metamodels.

Subsequently, we provide an overview of an extended version of the RTL metamodel developed during this dissertation, following by a detailed analytical approach.

7.3.1 An overview of the RTL metamodel

The RTL metamodel is explained by means of a view based compositional approach, similar to the one presented earlier in the precedent chapter. However, we do not express every minute detail related to the metamodel, but instead choose to focus only on significant key points. We now describe the necessary details required for expressing different concepts related to the hardware accelerator and the control aspects; such as components, ports, connectors, implementations, configurations and stategraphs.

Naming terminology. We first provide a naming terminology for the concepts present in the RTL metamodel. This terminology helps in a clear distinction between the concepts related to either control or the hardware accelerator. Metaclasses that start with the **RTL_** prefix, such as `RTL_Element` or `RTL_Model` are the common concepts used for both the hardware accelerator and the control features. While concepts initiating with the **HW_** prefix such as `HW_CodeFile` are used specifically for the construction of the dynamically reconfigurable hardware accelerator. Similarly, metaclasses starting with the **Control_** prefix are related only to the control.

7.3.2 Basic concepts

In the subsequent section, we move onto providing the basic concepts related to the construction of the RTL metamodel.

7.3.2.1 Kernel of the RTL metamodel

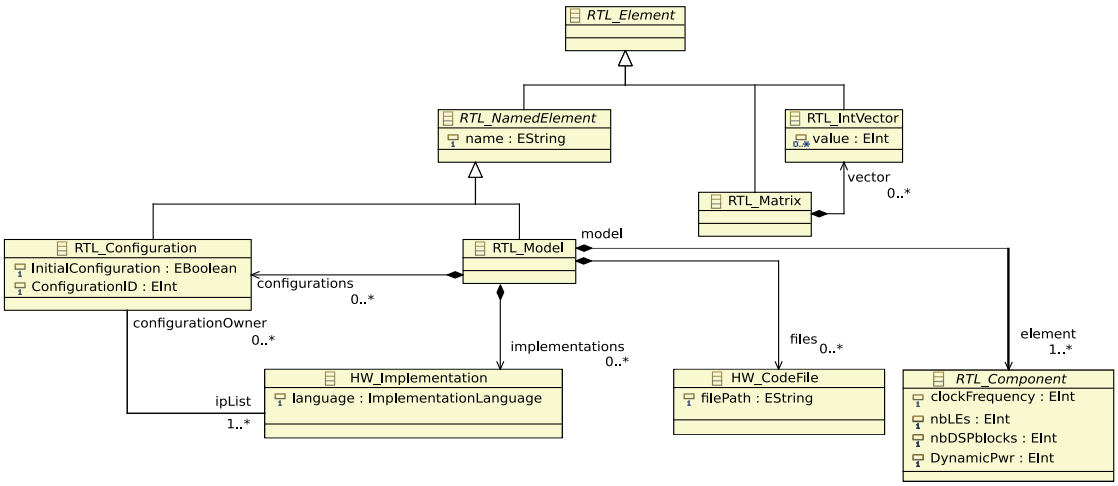


Figure 7.8: Kernel of the RTL metamodel

We first present the core concepts of the RTL metamodel. The highest concept present in the RTL metamodel is the `RTL_Element` metaclass representing a generic abstract entity. The `RTL_NamedElement` class specializes from this concept and provides a unique name by means of its associated attribute, that assigns a `name` to each entity. The `RTL_Model` is one of the output models produced by the *MARTE2RTL* transformation as discussed in chapter 6. This metaclass encompasses all the other concepts present in the RTL metamodel, except the concepts related to port types, as specified later on in section 7.3.2.6, that are related to and produced in the

RTL_PortType model. A RTL_Model mainly contains RTL_Component(s), HW_CodeFile(s), HW_Implementation(s), RTL_Configuration(s) along with integer vectors and matrices: RTL_IntVector and RTL_Matrix respectively.

A RTL_Component is a generic abstract component in the RTL metamodel: it can be used as a basic component of the hardware accelerator or as a basic module for control. This entity can be either hierarchic or elementary in nature. Details related to this concept are subsequently presented in the next section. The concepts of HW_CodeFile and HW_Implementation have equivalent meanings as their namesake concepts described in the deployed metamodel, presented in section 6.4.3. Similarly, the concept of a *Configuration* in the deployed metamodel is translated into an equivalent RTL_Configuration metaclass. A RTL_Model can contain several code files, implementations and configurations due to the presence of compositional *files*, *implementations* and *configurations* metarelations respectively. A RTL_Matrix represents a matrix specified at the high level UML model and is composed of RTL_IntVector(s) by means of the *vector* relation.

7.3.2.2 Component concepts in RTL metamodel

A RTL_Component in the RTL metamodel, as shown in Figure 7.9, serves two functionalities. It can either be considered as a component of the eventual hardware accelerator, generated from the RTL model corresponding to a RTL metamodel; or as a component of the control semantics introduced earlier in section 6.4. For example, a *Mode Switch Component* can be viewed as a RTL_Component. A RTL_Component can be either hierarchical or elementary in nature as evident by the specialized RTL_Hierarchical and RTL_Elementary metaclasses. A RTL_Component itself specializes from the RTL_NamedElement class, allowing a unique name (in the form of a string of characters) for each RTL_Component.

Similar to the nature of the RTL_Component, RTL_Hierarchical and RTL_Elementary metaclasses are abstract in nature. They help to differentiate the nature of components related to hardware accelerators and control features. A RTL_Hierarchical metaclass is used for components having some internal hierarchies, as compared to a RTL_Elementary class that is atomic in nature. A RTL_Hierarchical metaclass contains RTL_Connector(s), in order to define a relationship between two internal subcomponents.

As defined in section 4.2.2.2, a modeled Gaspard2 application can have subcomponents, each either *composite*, *repetitive* or *elementary* in nature. Similarly in the RTL metamodel, these concepts are also present for the accelerator: in the form of HW_RepetitiveComponent, HW_CompoundComponent and the HW_TE metaclasses. The HW_RepetitiveComponent and HW_CompoundComponent both specialize from the RTL_Hierarchical metaclass. A HW_TE represents an atomic element in the hardware execution model of Gaspard2 applications; and does not have any internal structure. Its complex behavior is determined by one *or* several IPs via the *implementation* reference. The relationship between a HW_TE and its respective IPs is further defined in section 7.3.2.10. Finally, the HW_TE metaclass specializes from the RTL_Elementary concept.

A HW_CompoundComponent expresses task parallelism in the hardware execution model of the Gaspard2 applications. It can contain connectors as well as component instances, i.e., HW_ComponentInstance(s). A HW_ComponentInstance references a RTL_Component by means of the *hwcomponent* reference. For a compound component, the internal component instances represent the tasks exhibiting task parallelism and the connectors determine the data dependencies between the tasks. The RTL metamodel does not impose any restriction on the nature of a component referenced by its component instances. A composite component thus can contain composite, repetitive or elementary components, permitting creation of an accelerator with different hierarchical levels.

A HW_RepetitiveComponent concept in the RTL metamodel, contains a single component instance. This metaclass is not illustrated here; and presented in the next section. The metaclass expresses the data parallelism related to the hardware execution model and is explained further in section 7.3.2.3.

Similar to the application aspects present in the RTL metamodel, the control semantics also contains components having composite, repetitive and atomic natures. For example, a

Macro Component can be observed having a composite nature; as compared to the repetitive *Deployed Automata* component; as evident in section 6.5.3. Similarly, a *Gaspard State Graph Component* can be viewed as an atomic component; but having its behavior specified by a state graph as compared to an IP. In order to take into account all these aspects, the metaclasses `Control_CompoundComponent`, `Control_RepetitiveComponent` and `Control_Node` have been created; for representing composite, repetitive and elementary concepts for control respectively. These separate classes have been created to help facilitate the development of the corresponding model transformations.

The difference between the atomic `HW_TE` and `Control_Node` components is evident. The first represents an atomic component for a hardware accelerator with its behavior determined by an IP. This behavior may not be expressed graphically. In contrast, a `Control_Node` represents an atomic concept for the control, but having its internal behavior expressed via a state graph. Similarly, a control compound component can have several related `Collaborations`, as compared to a compound component for the hardware accelerator. We now look at the repetitive components related to control and hardware accelerator in the RTL metamodel.

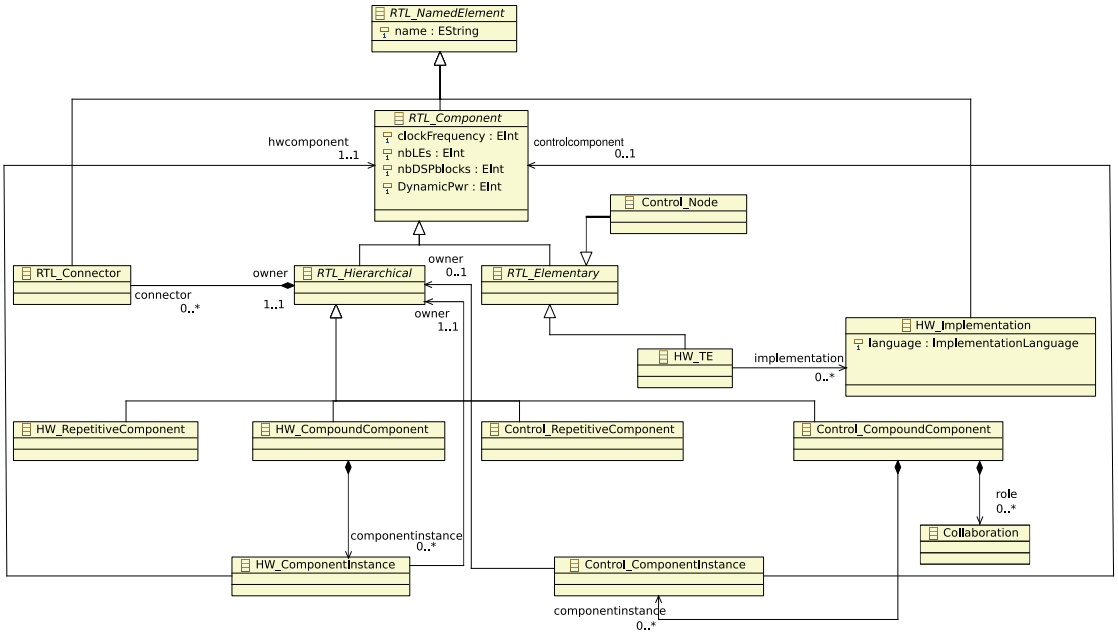


Figure 7.9: Concepts related to composite, repetitive and elementary components of hardware accelerator and control

7.3.2.3 Repetitive components

A `HW_RepetitiveComponent` metaclass as illustrated in Figure 7.10 defines a repetitive component for a hardware accelerator; and permits to specify hardware execution of data parallelism present in Gaspard2 applications. Parallel execution of data parallelism is enabled by instantiating multiple instances of a repeated task, i.e, an RT. An `HW_ComponentInstance` has a specific `HW_Shape` as indicated by the *dim* reference, for expressing the repetition space of a repeated task, in a RCT. This RCT corresponds to a component containing the `HW_ComponentInstance` metaclass by means of the *refComponentInstance* relation. The types of the hardware or control repeated instances can be determined by respective references to the `RCT_Component` as illustrated in the previous section. Similar to a repetitive hardware component, for a repetitive component in the control model: a `Control_RepetitiveComponent` metaclass is created, its multiple instances are determined by the `Control_ComponentInstance` metaclass, having a respective associated `RTL_Shape`.

A repetitive component (either related to hardware accelerator or control) contains special component instances which correspond to the tiler components: `RTL_Tiler`, defined later

in section 7.3.2.8. In fact, these tiler components carry out the ADO pre-computations as defined earlier in section 7.2.1.3. The concept of this component instance corresponds to either the `HW_ComponentInstance` or `Control_ComponentInstance` metaclasses; for hardware accelerator or control respectively.

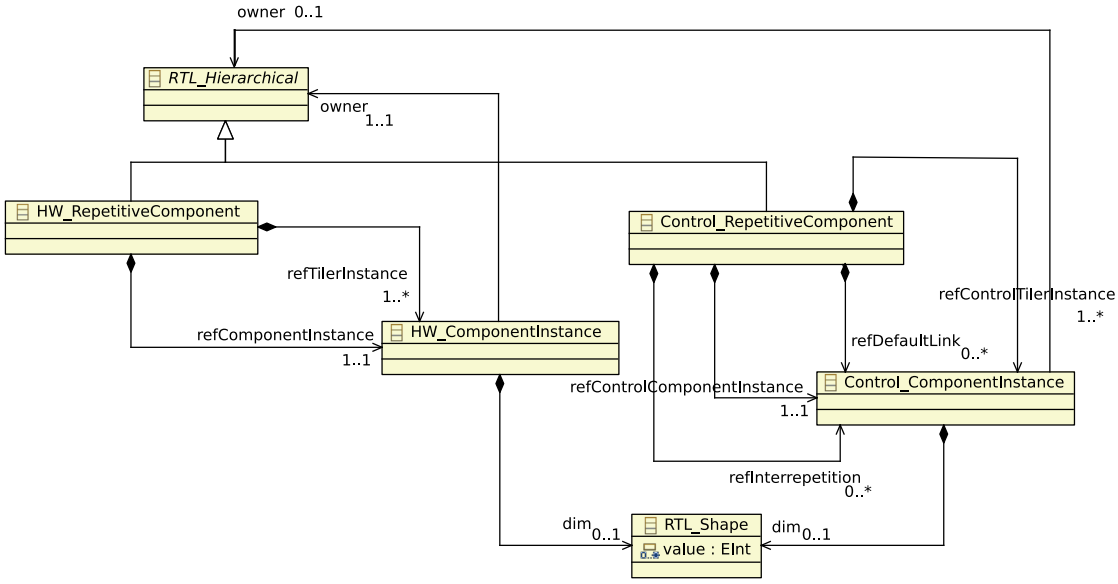


Figure 7.10: Concepts related to repetitive components in the RTL metamodel

To distinguish between the instantiations of a repeated task and that of a tiler, a `HW_ComponentInstance` is referenced by *refComponentInstance* or *refTilerInstance* appropriately. A similar mechanism is applied for the `Control_ComponentInstance`: the *refControlComponentInstance* and *refControlTilerInstance* references by the `Control_RepetitiveComponent` metaclass. In order to respect the RSM semantics, a repetitive component (or a RCT) must contain *at least* one component instance of a repeated task and one tiler instance. In fact, a RCT must contain at least one input or output tiler.

A `Control_RepetitiveComponent` also contains two other component instances: that relate to an *interrepetition dependency* and a *defaultLink*. Details related to these concepts have been presented in section 7.3.2.9. For their instantiation, metarelations *refDefaultLink* and *refInterrepetition* are present between the `Control_ComponentInstance` and `Control_RepetitiveComponent` metaclasses.

7.3.2.4 Ports

Communication with a component is possible through its associated ports as shown in Figure 7.11. A `RTL_Component` can contain an arbitrary number of `RTL_Ports` by means of the compositional *ports* relation. A `RTL_Port` specializes the `RTL_ConnectableElement` concept and contains information on the organization of the data/control arrays by means of a `RTL_Shape`; and their respective types by means of the `RTL_PortType` metaclass. A `RTL_Shape` defines the dimensions of a port by means of the *dim* relation; while `RTL_PortType` determines its type. The port types in the RTL metamodel are defined later on in section 7.3.2.6. Moreover, a `RTL_Port` is specialized by its sub types: `RTL_InputPort` and `RTL_OutputPort` representing input and output ports respectively.

Besides the communication aspects related to a component, no other functionality has been attributed to the `RTL_Port` concept. Nevertheless, the advantage of a metamodel is that new relations can be created quite easily. It is thus possible to create different new ports from new unique relations between a `RTL_Component` and `RTL_Port`. For the context related to hardware accelerators, each associated component of the RTL model contains clock and reset input ports by means of the *rst* and *clk* relations with an `RTL_InputPort`. We impose a condition

that a hardware accelerator component can only have single clock/reset ports in order to respect general RTL semantics.

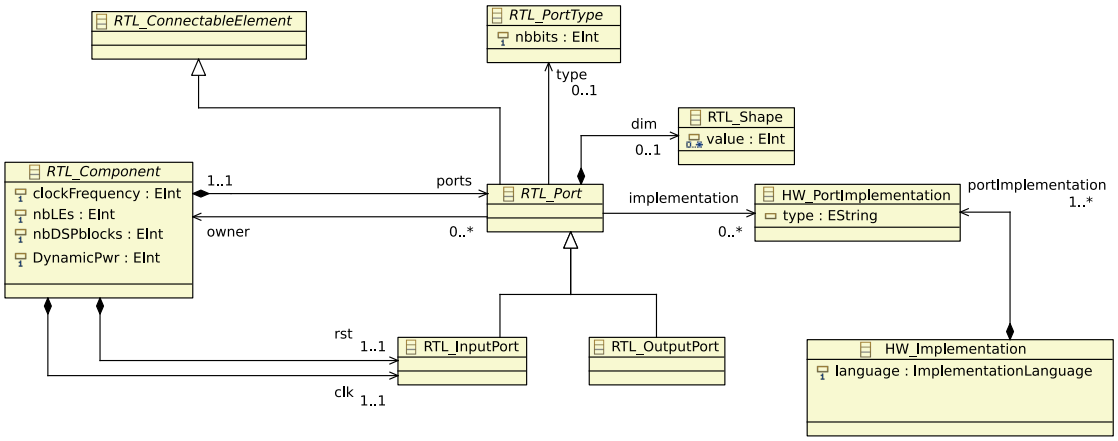


Figure 7.11: Each component in the RTL model contains several ports. Components related to the hardware accelerator also possess input clock and reset ports

7.3.2.5 Instances

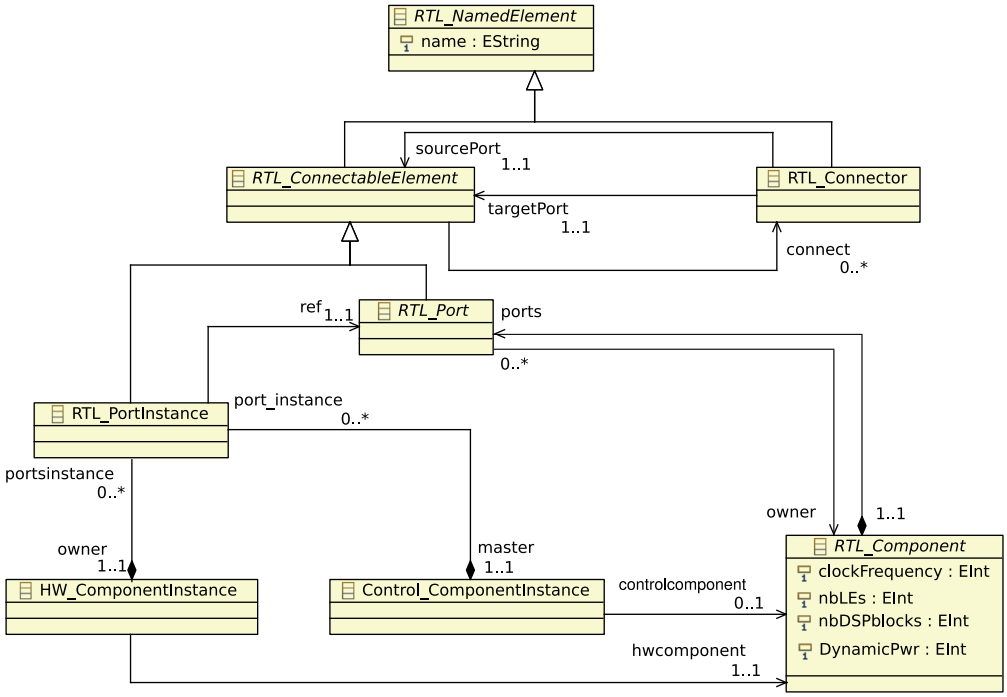


Figure 7.12: Concepts related to component instances in the RTL metamodel

From the point of view of the RTL metamodel, a port is directly accessible by its owner component by means of the *ports* relation. When a component related to the control aspects or an accelerator is instantiated by means of the `HW_ComponentInstance` or `Control_ComponentInstance` metaclass respectively; it contains its respective port instances, by means of the compositional *portsinstance* or *port_instance* metarelations with the `RTL_PortInstance` concept. The link between an instance of a port and the port itself is defined by the *ref* relation, between the `RTL_PortInstance` and `RTL_Port` metaclasses. Sim-

ilarly, instance of a module of a hardware accelerator or control references a component by means of *hwcomponent* or *controlcomponent* respectively.

The `RTL_Port` and `RTL_PortInstance` specialize from the `RTL_ConnectableElement` which itself references a `RTL_Connector`. Each `RTL_Connector` possesses one source and target port as specified by the *sourcePort* and *targetport* metarelations to the `RTL_ConnectableElement`. A `RTL_Connector` is detailed later on in section 7.3.2.7.

7.3.2.6 Port Type

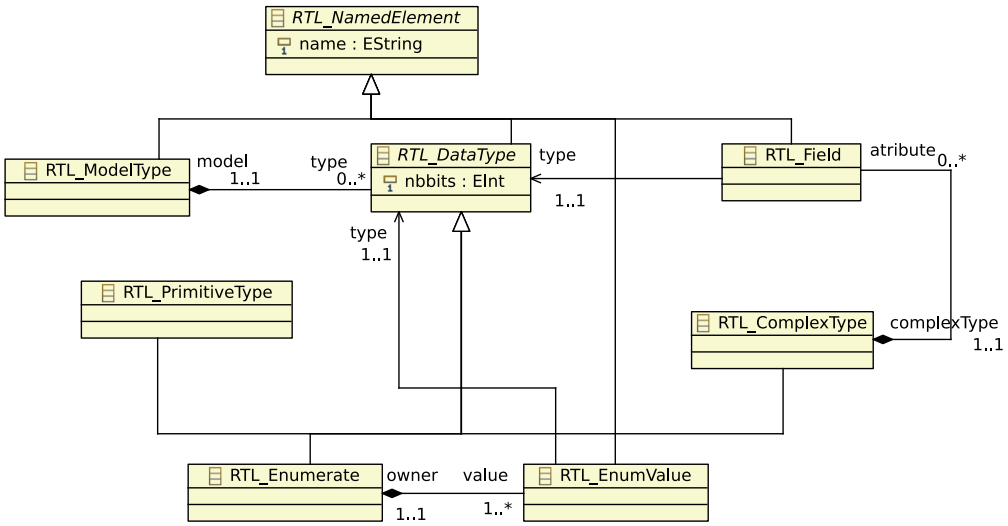


Figure 7.13: Different port types related to data and control flow in the RTL metamodel

The port types used in the RTL metamodel serve to determine the type of control or data related to ports of a `RTL_Component`. In the previous chapter, we have already described the integration of the concept related to bits for a MARTE data type, for a hardware functionality. The RTL metamodel also inspires from these semantics, while enriching the concepts with added information for RTL requirements.

In the RTL metamodel, the abstract `RTL_PortType` concept (for representing either control or data) is specialized by several types: primitive `RTL_PrimitiveType`, complex `RTL_ComplexType` and enumerated `RTL_Enumerate` as seen in Figure 7.13. We now briefly describe each of these types:

- `RTL_PrimitiveType`: an elementary type in the RTL metamodel. It permits to represent basic types such as Integer, Boolean, Float etc. The control semantics usually make use of Boolean types for determining the arrival of events.
- `RTL_ComplexType`: A complex type can be composed of several fields: `RTL_Field`(s), each field referencing a specific port type. Hence, all possible combinations are possible for the construction of a complex type, as no compositional restrictions are imposed by the metamodel.
- `RTL_Enumerate`: An enumerated type that is composed of several enumerations or `RTL_EnumValue`(s). Each of these values also references a port type. They permit defining of user customized types, and are mainly used in the control semantics.

Apart from the nature of a type, no semantics are attached to their behavior. However, the metamodelization is not generic enough for the specification and generation of complex and primitive types present in different languages. For this reason, an effective solution has been adapted which allows to generate the code relating to primitive or complex types directly on the basis of their names as specified at the MARTE modeling level.

For the generation of hardware accelerator in the RTL metamodel, while the utilization of names of different types is sufficient for the generation of a synthesizable HDL code; it is not sufficient enough to give a concrete evaluation of the generated hardware functionality (for example, the number of the reconfigurable FPGA resources such as slices or CLBs). In fact, this estimation is directly dependent on the nature of a type: if a type requires an increased number of bits for itself; the elements based on this type consume more resources. As we have already introduced the notion of bits to a MARTE data type, the same semantics are applied in the RTL metamodel. This notion permits to determine the quantity of registers required for storing the different types.

7.3.2.7 Connectors

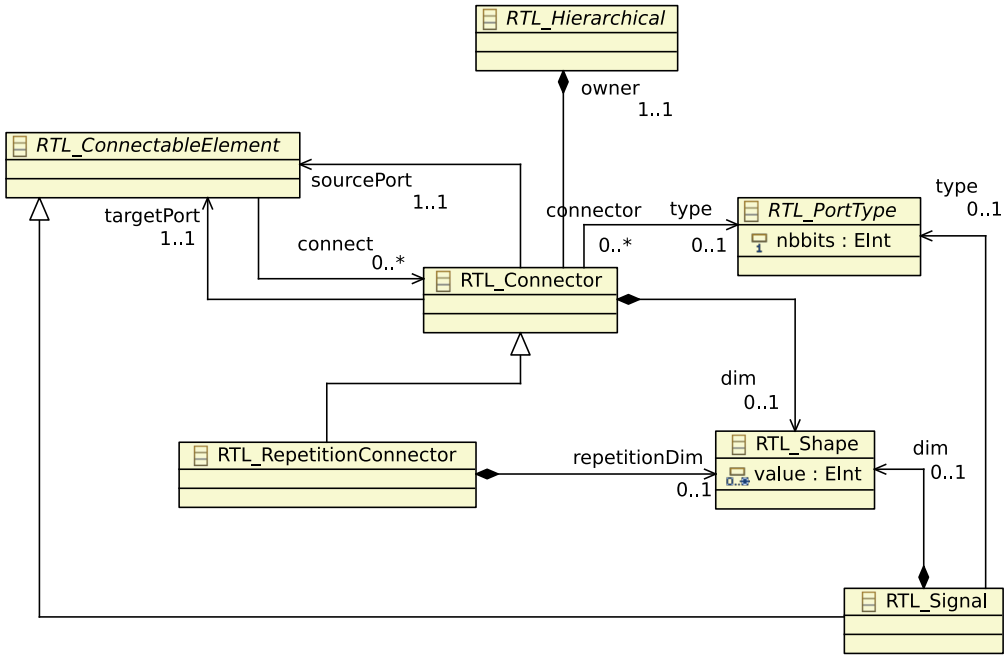


Figure 7.14: Connectors in the RTL metamodel

The concepts related to a `RTL_Connector` in the RTL metamodel are given in Figure 7.14. A `RTL_Connector` can be translated into a hardware signal in the case of a hardware accelerator, or is used as a link between different concepts for the control, such as connecting a *Gaspard State Graph component* to a *Mode Switch Component*. A `RTL_Connector` has its respective type and shape as indicated by the *type* and *dim* references to the `RTL_PortType` and `RTL_Shape` metaclasses respectively. The connector metaclass specifies the link between two RTL components, by means of a dependency between their ports, as shown by the *sourcePort* and *targetPort* references to the `RTL_ConnectableElement` metaclass.

A `RTL_ConnectableElement` is an abstract concept in the RTL metamodel. It is used to represent certain concepts that can be connected or attached to a `RTL_Component`, such as ports and connectors. It is inspired from the UML *ConnectableElement* concept [178].

A `RTL_Signal` specializes a connectable element and can be viewed as a connection point to a component. For tiler components, having several source and target indexes, a `RTL_Signal` specifies a connection point for each source/target index, and helps to connect either a `HW_SubConnector` or a `HW_DelayedConnector` from a source index to a target index. These concepts are explained in the next section.

A `RTL_RepetitionConnector` is a special type of connector used to connect the output and input ports of input and output tiler components respectively, to the various repeated component tasks (component instances) in a repetition context task. For example in Figure 7.6, the output connectors from a tiler component can be classified as `RTL_RepetitionConnector`(s).

These connectors have two shapes: one defined by the *repetitionDim* reference to a `RTL_Shape`; and the other (via the *dim* reference) due to the specialization of the `RTL_Connector` metaclass. The first shape corresponds to the repetition space of a repeated task, while the second corresponds to its pattern shape. The product of these two shape values help to create the correct number of links between the tiler component and the multiple repetitions of a repeated task.

In Figure 7.6, 64 `RTL_RepetitionConnector`(s) are created between an input tiler and each input port of the 64 repetitions of the *MultiplicationAddition* component; due to the repetition space value of {64} and a pattern shape value of {}. In a similar fashion, 64 `RTL_RepetitionConnector`(s) are created between the repeated task and the output tiler.

7.3.2.8 Tiler

The `RTL_Tiler` as represented by the Figure 7.15, extends the `RTL_Elementary` concept, as it itself is an atomic element in the RTL metamodel. A `RTL_Tiler` is specialized by `RTL_InputTiler` and `RTL_OutputTiler` metaclasses, which determine the direction of a tiler connector. An input tiler helps to create the patterns from an input array; while an output tiler constructs an output array from the produced patterns. Each tiler contains two `RTL_Shape` concepts: one corresponding to the *repetitionSpace* and *patternShape* as illustrated by the equivalent references. The first shape corresponds to the number of patterns consumed/produced by a tiler; while the second shape determines the form of these patterns. An `RTL_Tiler` also contains several `RTL_Signals`.

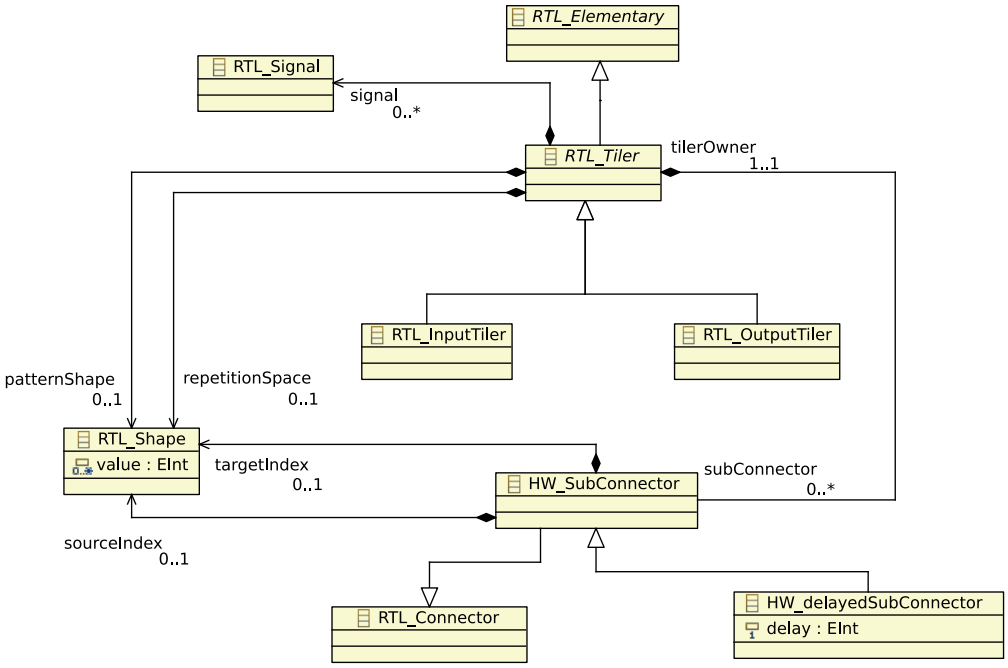


Figure 7.15: Concepts related to Tilers in RTL metamodel

As seen in section 7.2.1.3, hardware execution of Gaspard2 applications require ADO precomputations. This implies that the tilers do not exist anymore in their actual form as present in the application model (MARTE connectors with origin, fitting and paving attributes), but mainly in the form of an interconnection topology linking the data in arrays to data in patterns. The concept of `HW_SubConnector` and `HW_delayedSubConnector` express these hardware connections.

A `HW_SubConnector` extends a `Connector` and permits an interconnection between a source and target index of a tiler component by means of the *sourceIndex* and *targetIndex*: a data in a data array can be thus directly connected to data in a pattern. No constraints exist on the dimensions of the arrays, permitting to create the connectors between the arrays independent of their dimension.

A `HW_delayedSubConnector` is a type of subconnector for the hardware accelerator part of the RTL metamodel. It consists of a delay attribute that determines for a tiler component, the delay existing between an instant, when the data is present in an input array, and the instant it is transfered to the output pattern. This concept helps to create a shift register in a hardware accelerator, for managing sliding windows data dependencies. The delay is presented in the form of an integer value; and represents clock cycles: a delay value of **5** indicates that the data is delayed by 5 clock cycles.

7.3.2.9 Interrepetition and defaultLink

The mechanisms related to the interrepetition dependency and defaultLink have been presented in section 7.2.2. Similarly to tilers, these MARTE concepts do not retain their actual forms, specified either in the application or control modeling levels. Currently the RTL metamodel only treats these dependencies for the control semantics. These metaclasses specialize from the abstract `RTL_DefaultRepetition` concept that permits formation of respective ports for these concepts; due to the references *repetitionSize* and *patternSize* to the `RTL_Shape` metaclass.

While both of these concepts can be utilized either for the hardware accelerator or control, as explained previously, currently we have only implemented them for the control methodology as they are integral for the conception of a mode automata.

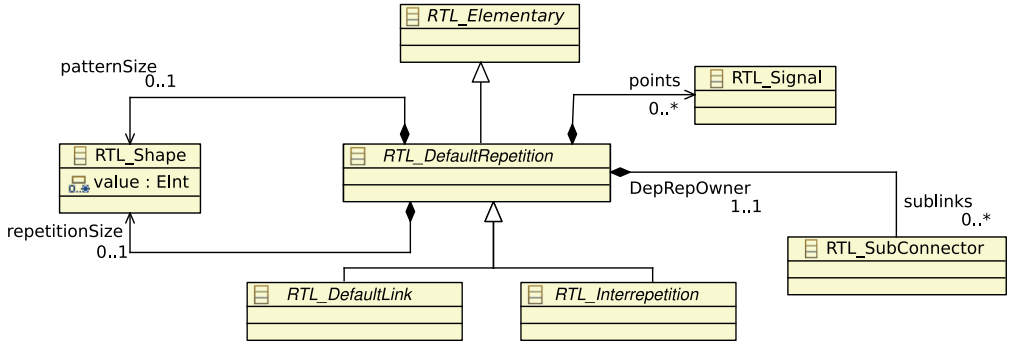


Figure 7.16: The Interrepetition and defaultLink metaclasses in the RTL metamodel

7.3.2.10 Implementation Concepts

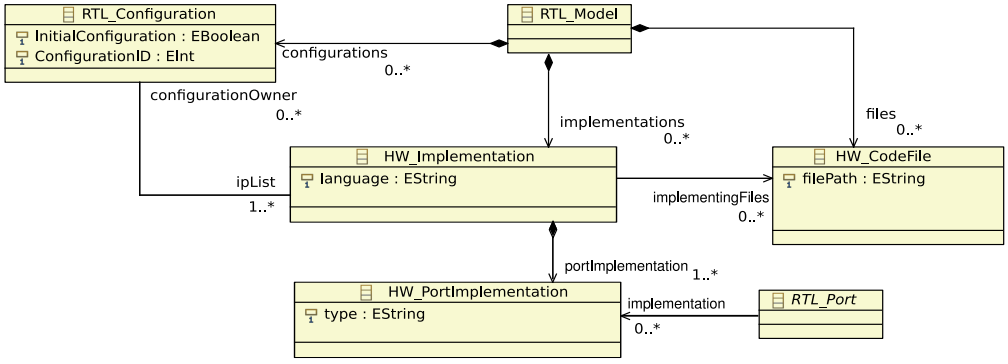


Figure 7.17: Concepts related to implementations in RTL metamodel

The concepts related to IPs, such as implementations, code files and configurations as specified in section 6.4.3, have a direct one to one relationship with similar concepts present in the extended MARTE metamodel.

The `HW_Implementation` metaclass as shown in Figure 7.17 is an IP related to a `HW_TE`, and can have several `HW_CodeFile`(s) as defined by the *implementingFiles* relationship between

the two metaclasses. Respecting the conditions as specified in the previous chapter, an implementation has an interface equivalent to its associated elementary component. This condition is satisfied by the compositional relationship between a `HW_Implementation` and a `HW_PortImplementation`. A `HW_PortImplementation` has a type similar to an equivalent port of the elementary component. The reference *implementation* between a `RTL_Port` and a port implementation helps to resolve the associated port of the elementary component.

A `HW_Implementation` can be associated with a `RTL_Configuration` which contributes in introducing dynamic features in the RTL metamodel. As before, a `RTL_Configuration` must have at least one implementation (of each elementary component) for a successful configuration. A `RTL_Model` can have several configurations, with each representing one global implementation of the hardware functionality modeled via the MARTE profile.

7.3.2.11 Collaboration

A `Collaboration` in the RTL metamodel has equivalent semantics to its counterpart as defined earlier in section 6.4.2, with some slight changes to accommodate for the RTL details. A `Collaboration` specializes from the `RTL_Component` and refers two metaclasses: the `Control_CompoundComponent` and the `RTL_Configuration`, by means of the *owncomponent* and *ownconfiguration* references. The specializations offer a unique name to a collaboration; which can be viewed as a mode value.

We apply the constraints that one collaboration can only be associated to a single configuration and a control compound component. This is due to the proposed control semantics, which permit to determine the internal behavior of a control compound component, i.e., the mode switch component, by means of the collaborations. A single collaboration is responsible for the mode switch related to one configuration in a mode switch component, by means of an associated mode value.

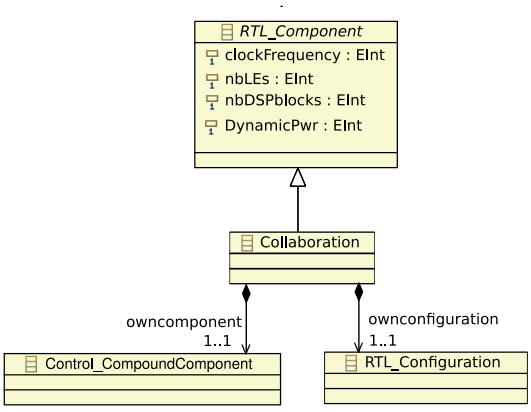


Figure 7.18: Collaboration metaclass in the RTL metamodel

7.3.2.12 Automaton

This part of the RTL metamodel is mainly concerned with the concepts related to automata in the RTL metamodel. These concepts are similar to the Gaspard State graph concepts introduced earlier in section 6.4.2.

A `Control_Behavior` corresponds to the *Behavior* concept in the extended MARTE metamodel; and is associated with an atomic `Control_Node`. This `Control_Behavior` is composed of an `Automaton` that contains `State`(s) and `Transition`(s).

As compared to the different types of states present in the *StateGraph*, the RTL metamodel currently only takes simple states into account. Additionally, a *PseudoState* in the *StateGraph* package in the extended MARTE metamodel is converted into a *State* metaclass in the RTL metamodel. The isState attribute related to this metaclass determines the nature of the state: whether

it is an actual state or an initial pseudo state. A `State` itself has a control behavior as evident by the *function* reference: the `ControlOpaqueBehavior`. This behavior corresponds to the *doActivity* of a state in a *StateGraph*. The opaque behavior of a state has two attributes, statevalue and modevalue, that determine the current state and its related mode value. These values take as input the expression provided by the *doActivity* feature, and separate it into proper relevant parts for the eventual code generation.

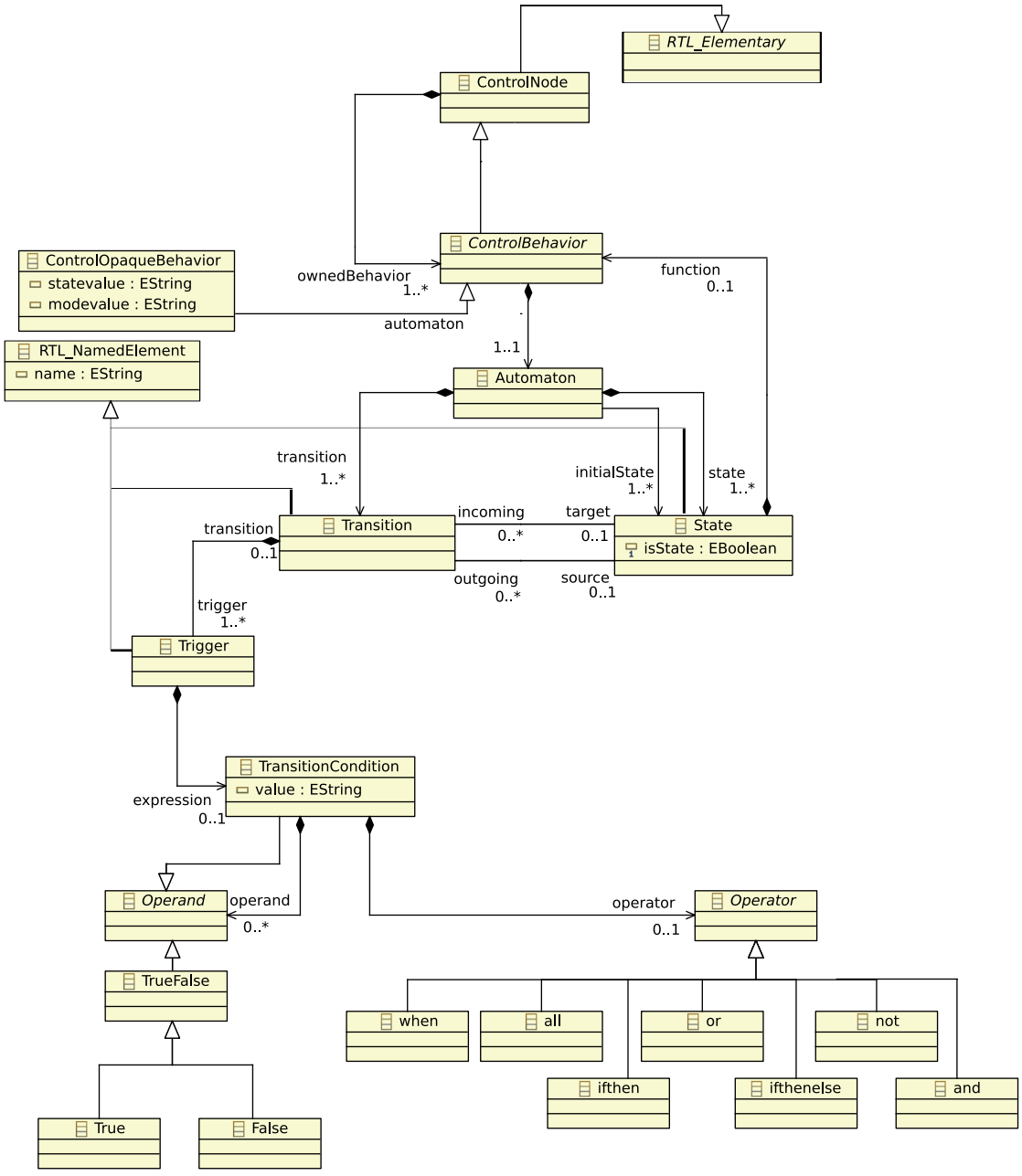


Figure 7.19: Concepts related to control in the RTL metamodel: specification of mode automata

A `State` also has incoming and outgoing `Transitions`. A transition is always associated with one or more `Trigger`(s). In turn, a `Trigger` is related to a `TransitionCondition` which can be equivalent to an *Event*. A condition related to either a *ChangeEvent* or *AnyReceiveEvent* is taken as input for the value attribute associated with a transition condition. This value is processed iteratively in order to separate it into `Operand`(s) and `Operator`(s). An operand can be either `TrueFalse` in nature or its specializations: either `True` or `False`. Similarly, an operator can be of different types: such as `when`, `all`, `and`, `or`, etc.

For example, an expression: *when ifelse_event and not dsp_event* as illustrated in Figure 6.31 is processed iteratively using the model transformations as: 1) First, for the whole expression, **when** is the operation, while **ifelse_event and dsp_event** is the operand. 2) Then the operand is analysed and is also considered as an expression. Now the operator is **and** while operands are **ifelse_event** and **not dsp_event**. Then finally, the last expression is evaluated with **not** being the operator and **dsp_event** being an operand. This iterative mechanism is used to ease the final code generation related to the mode automata.

7.4 Conclusions

This chapter provides the basic concepts that comprise the RTL metamodel in our design flow. The metamodel mainly consists of two significant areas, one related to the hardware execution model of Gaspard2 applications, while the other is related to the conversion of a mode automata from Gaspard state graphs. Basic principles related to the hardware execution model are highlighted: such as conserving the parallelism specified in the application at the UML model. Afterwards, the various concepts present in the metamodel are analyzed. While some of these concepts have a one-to-one relationship with the extended MARTE metamodel introduced in the previous chapter, the RTL metamodel enriches these concepts in order to bring the abstraction level closer to the RTL for the eventual code generation.

This chapter concludes the theoretical contributions present in our dissertation. We now move onto the implementation portion, which first details the various model transformations present in our design flow, that help to convert the various concepts present in the different metamodels. Afterwards, a case study is presented, that validates our design methodology.

Part III

Implementation details and case study

Chapter 8

Model transformations and code generation

This chapter provides the details related to the model-to-model and model-to-text transformations present in our design flow for implementing partial dynamically reconfigurable SoCs. With regards to model-to-model transformations, we present: *UML2MARTE* and *MARTE2RTL*, the two principle transformations; and provide the basic semantics related to these transformations. The first transformation converts an UML model of a deployed application with integrated control aspects into a MARTE model, that is then taken as input by the second transformation, for subsequent conversion into a model corresponding to the details related to the RTL. Afterwards, we detail the semantics related to the *RTL2CODE* model-to-text transformation that permits to generate the source code, for eventual utilization in commercial synthesis tools for implementing dynamic reconfiguration.

8.1 Model-to-Model transformations

As described in section 3.1.3, model transformations enabling conversion of higher abstraction model(s) into lower enriched model(s), each corresponding to their respective metamodels. In this section, initially, we first describe the general details of the two main transformations in our design methodology. Here, we do not describe *how* these transformations are carried out, but instead focus on *what* is to be transformed. The first part is covered later on in the chapter. We initially provide the basic goals of these transformations, before moving on to certain examples in our transformation chain, specified in section 8.1.4. These examples illustrate the technical details related to the above mentioned transformations.

8.1.1 UML 2 MARTE transformation

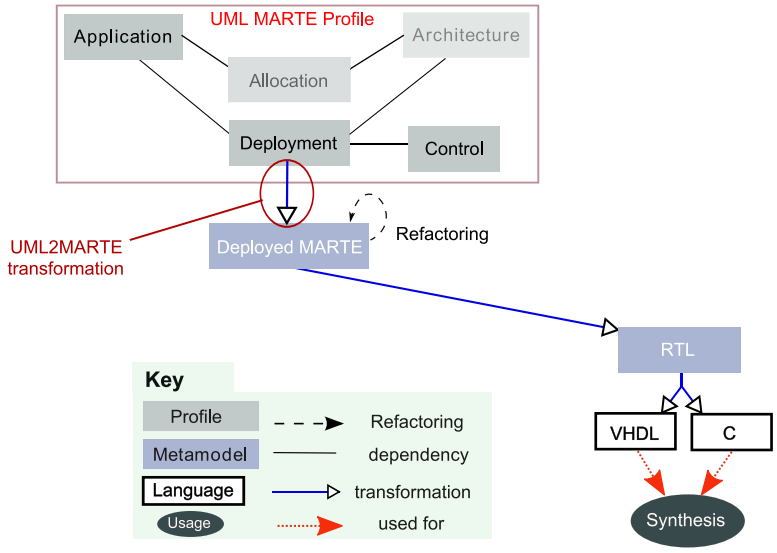


Figure 8.1: The *UML2MARTE* model transformation in our design flow

Figure 8.1 shows the *UML2MARTE* transformation in our design flow. The transformation has been developed internally in the DaRT team and allows to transform an UML model consisting of a modeled deployed application, architecture or an allocation of the two into an intermediate MARTE model, as indicated in Figure 5.2. The UML model conforms to the UML metamodel and the MARTE profile, while the MARTE model respects the extended MARTE metamodel illustrated in section 6.4. While we have also contributed to the development of the global *UML2MARTE* transformation, with respect to this dissertation and our proposed design flow, we present the two main contributions that we have integrated in this transformation.

Contributions in *UML2MARTE*. Firstly, as seen in chapter 6, for the modeling of a mode automata, we make use of UML state machine diagrams in the Gaspard2 environment. While these concepts are present in the UML model (due to the presence of the associated metaclasses in the UML metamodel), they cannot be interpreted directly by model transformations into a MARTE model, due to the absence of corresponding metaclasses in the current MARTE metamodel. Thus these necessary concepts were introduced in an extended version of the MARTE metamodel, as specified in section 6.4, for a successful interpretation. Firstly, a UML component adhering to the MARTE profile, modeled via a graphical modeling tool such as Papyrus, is transformed into a MARTE *StructuredComponent*. This entity contains additional elements, such as assembly parts, flow ports and connectors. As these concepts have been developed globally by our research team and do not count as our unique contributions, they are represented differently in the Figure 8.2 with bold outlines, as compared to our unique contributions, presented in a normal manner. The figure shows the global overview of this model transformation, with respect to the design flow presented in this thesis.

In this transformation, we first transform a UML state machine attached to a component, into a state graph related to a corresponding structured component. A state graph can have several regions, each in turn can have multiple transitions and vertices. A *Vertex* is an abstract concept and can either be a pseudostate or a state. A state in turn, can have a *doActivity* which determines the behavior of the state. A transition consists of source and target vertices; and may have multiple triggers. Each trigger is associated with an event, either a *ChangeEvent* or a *AnyReceiveEvent*. In case of the former, it contains a *ChangeExpression* which is effectively transformed into a MARTE *LiteralString* as specified in the VSL package. The UML collaborations, present in the UML model are directly converted into MARTE collaborations in the MARTE model. A collaboration in our transformation chain can have interior assembly parts which serve to determine the behavior of a structured component, to which the collaboration relates to. While

according to pure UML specifications, a collaboration can contain ports and connectors as well, as seen in section 6.5.3 related to our control model, we only require the assembly parts in a modeled collaboration.

Finally a *configuration* present in the UML model in the form of a component is also converted into a structured component and contains information related to the software implementations in the UML model.

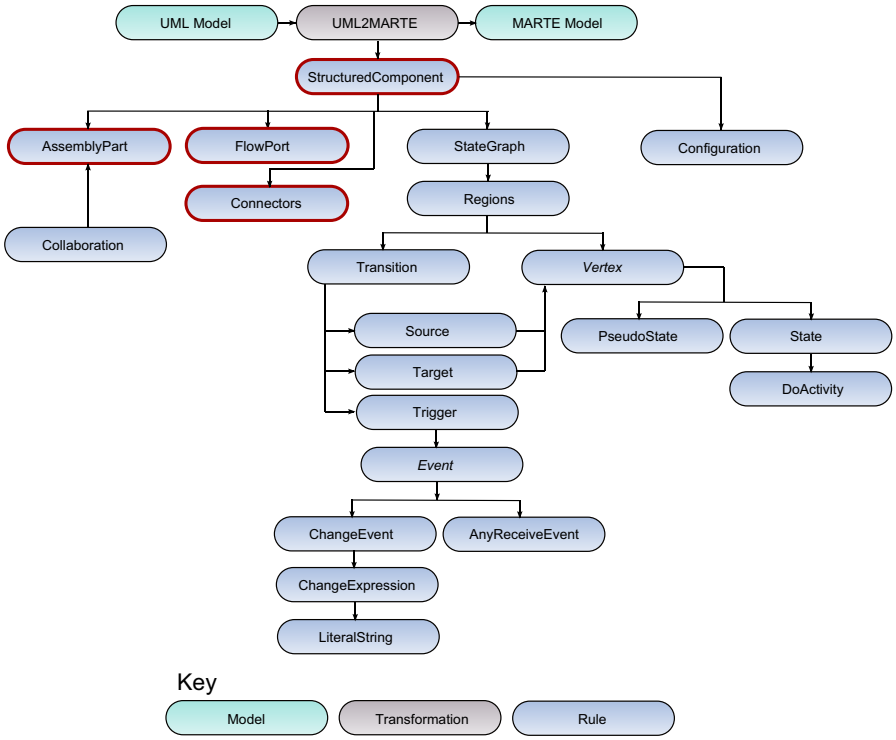


Figure 8.2: Overview of the *UML2MARTE* model transformation

8.1.2 MARTE 2 RTL transformation

Once the MARTE model has been created from the *UML2MARTE* transformation, we move onto the second model transformation in our design flow as illustrated in Figure 8.3. The *MARTE2RTL* transformation takes the MARTE model as input and generates two output models: the *RTL Model*, containing the concepts related to the hardware accelerator/control; and the *RTL PortType* model, containing the control and data types present in the MARTE model. The *MARTE2RTL* transformation chain, as illustrated in Figure 8.4 has been entirely developed during the course of this dissertation. The transformations rules related to the *RTL PortType* model permit creation of different control and data types related to the ports of the control concepts and the application functionality. Namely three types are created: the *RTL Primitive Type*, *RTL Complex Type* and *RTL Enumeration*.

With regards to the RTL model, the significant rules are as follows: A MARTE *StructuredComponent* having multiple component instances and no associated collaborations is transformed into an *HW Compound* component, while a similar structured component having related collaborations (either associated to itself or a sub component at any lower hierarchical level, i.e., a mode switch component or a macro component respectively) is converted into a *Control Compound* component. The collaborations that are present in the MARTE model are directly converted into their equivalent counterparts. A repetition context task having a RT is transformed into a *HW Repetitive* component, if it only contains *tiler* (either input or output tilers) connectors. In contrast, a RCT also having internal *interrepetition dependency* and *defaultLink* connectors is translated into a *Control Repetitive* component.

In similar manner, an atomic structured component having no related behavior is viewed as an atomic element of the hardware accelerator and is subsequently converted into an *HW Elementary* component. Inversely, if a behavior is associated with an atomic structured component, the component is converted into a *Control Node*. The tiler connectors present in the MARTE model are transformed into *RTL Tiler* components in the RTL model, as explained in the previous chapter. These components related to the hardware accelerator carry out ADO pre-computations, and help inf determining the data dependencies for the hardware functionality in the RTL model. Similarly, the abstract *Control DefaultRepetition* rule calls the *Control Interrepetition* and *Control DefaultLink* sub rules.

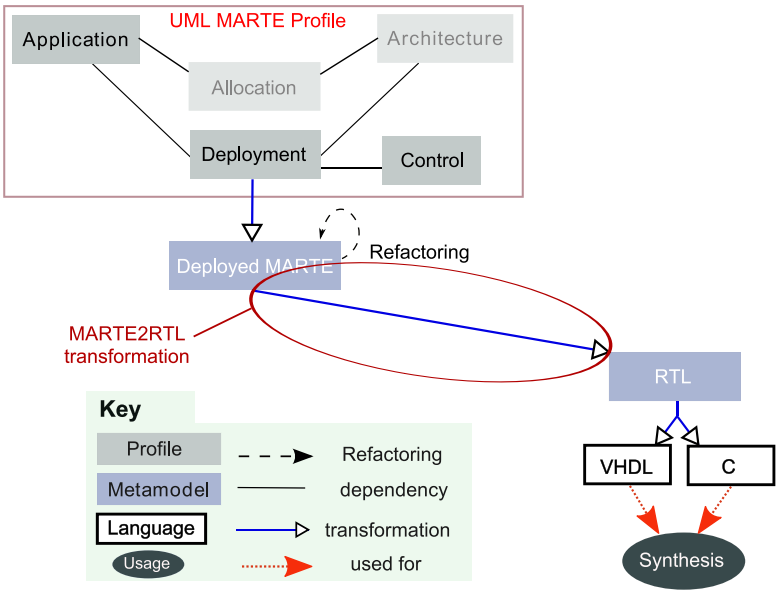


Figure 8.3: The *MARTE2RTL* model transformation in our design flow

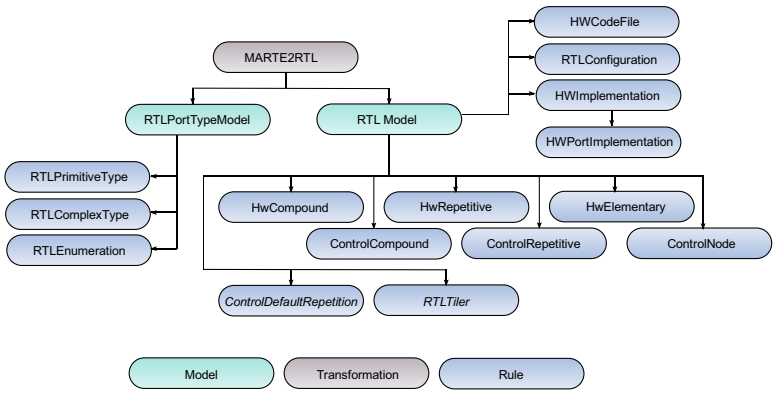


Figure 8.4: Overview of the MARTE2RTL model transformation

In order to convert the deployment information present in the MARTE model, the RTL model also contains equivalent concepts. A structured component which has been identified as a configuration is converted into an equivalent *RTL Configuration*. Similarly, the RTL model generates the *HW Implementation*(s) related to an *HW Elementary* component and the associated *CodeFile*(s). Finally the port implementations belonging to an IP related to an elementary component are created by means of the *HW PortImplementation* rule.

Figure 8.5 shows the rule hierarchy related to the *Control Node*. Many of these rules have a one to one equivalence with the rules related to state graph as presented in the earlier section. A *Control Node* contains its respective ports via the *RTLPort* rule. It also has an associated behavior

in the form of an *Automaton*, itself containing several transitions and vertices. A *Vertex* is an abstract concept and determines if the type of the vertex is either a pseudo state or a normal state. A pseudo state in the RTL metamodel is also converted into a state from which the initial transition takes place. A state contains an abstract *Function* that determines the next transited state and its corresponding mode value. In a manner similar to the transition rule present in *UML2MARTE* transformation, a transition in *MARTE2RTL* contains source and target states as well as triggers. A trigger is associated with an *Expression* that can be either an *AnyReceiveEvent* or *ChangeEvent*. In case of the former, the value of the expression is set as **all**, corresponding to a self transition. In case of the latter type, the expression is evaluated in an iterative manner and we get a list of *Operands* and *Operators*, as explained in chapter 7. The near equivalence between the rules related to a state graph in the *UML2MARTE* transformation; and the corresponding automaton in the *MARTE2RTL* permits an easy one-to-one correspondence.

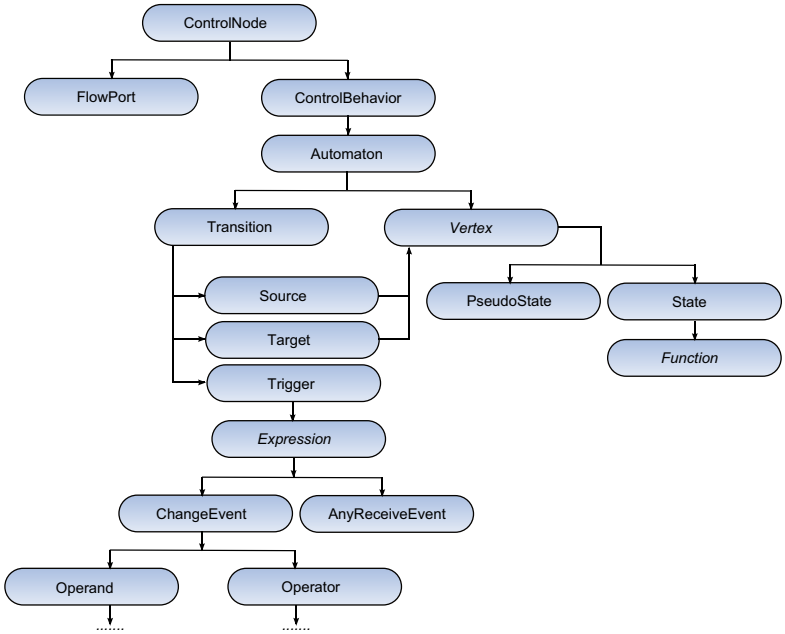


Figure 8.5: Transformation rules related to Control node

8.1.3 Implementing model-to-model transformations: QVT Operational (QVTO)

As described in section 3.1.3.6, current Gaspard2 framework makes use of the *Meta Object Facility Query/View/Transformation* (MOF QVT) [176] based model-to-model transformation languages. The MOF QVT specifications provide the standard model transformation guidelines in the OMG modeling framework. However, QVT is intended to be the industry standard only; and does not provides actual reference implementations. Currently three types of QVT model transformations have been hypothesized in the language dimension of QVT: mainly *QVT Operation (QVTO)*, *QVT Relations (QVTR)* and *QVT Core*. The Relations language has a strong relationship with the Core language, as it is transformed into the Core language for *execution*. The Relations language is intended to be a higher level declarative language as compared to the Core language which is defined at a lower abstraction level.

However, both the Relations and Core languages are currently under development within the Eclipse *Model-to-Model Transformation* (M2M) project[1]. Currently, at the time of writing of this dissertation, only QVTO is available, and has been chosen as the de-facto transformation language for the different models present in the Gaspard2 framework. QVTO makes extensive use of OCL and is mainly imperative in nature[2]. The language is capable to handle both unidi-

[1] http://www.eclipse.org/m2m/
[2] however as other model transformation languages, it supports a mixed declarative/imperative rule structure

rectional and bidirectional model-to-model transformations. The mixed imperative/declarative nature of QVTO provides an advantage over a purely declarative approach where there is no direct correspondence between the individual elements of source and target models. A QVTO transformation is also capable of converting *N* source models into *M* target models.

QVTO provides the advantage of *traceability*, as several language constructs are present to provide trace information in a model transformation. This traceability information is also available in the execution semantics, i.e., if an operation has been executed before with similar input parameters, a second execution does not creates new redundant target model elements, but instead will return the already created elements. *Mapping Operations* in QVTO perform the main task of generating output model elements from input model elements.

We now provide some basic concepts related to QVTO that are essential for understanding our transformation examples presented subsequently. The goal is not to provide in depth details related to QVTO, but to provide some significant key points which have been utilized in our model transformation chain.

8.1.3.1 Declaration of a transformation

A QVTO based transformation is written in a file containing a transformation signature and *main* mapping body to serve as an entry point. The transformation can have one or more *modeltype* explicitly defined declarations, for importing the different metamodels used in the model transformation. We provide a simple example of a model transformation which is a brief extract of the *MARTE2RTL* transformation.

```
1  modeltype ecore uses 'http://www.eclipse.org/emf/2002/Ecore';
   modeltype Foundations uses Foundations('http:///null/Foundations.ecore');
   modeltype GCM uses gcm('http:///marte/GCM.ecore');
   modeltype Extension uses extensions('http:///marte/CoreElements/extensions.ecore');
5  modeltype Primitives uses primitiveTypes('http:///marte/MARTE_PrimitiveTypes.ecore');
   modeltype VSL uses vsl('http:///marte/VSL.ecore');
   modeltype Lib uses 'http:///marte/MARTE_Library/MARTE_DataTypes.ecore';
   modeltype RSM uses rsm('http:///marte/RSM.ecore');
   modeltype dataTypes uses basicNfpTypes('http:///marte/MARTE_Library/BasicNFP_Types.ecore');
10 modeltype mmDeployment uses mmDeployment('http:///mmDeployment.ecore');
   modeltype CommonBehavior uses 'http:///null/Causality/CommonBehavior.ecore';
   modeltype mmRTL uses mmRTL('platform:/resource/fr.lifl.west.gaspard2.metamodel.rtl/model/rtl.ecore');
   ------------------------------------------------------------------------------------------------
   --This is the begining of the MARTE2RTL transformation, the transformation takes MARTE model as input
15 and produces two output models: RTL and RTL port type model

   transformation MARTE2RTL (in marte : Foundations, out RTL_Model : mmRTL, out RTL_PortType_Model : mmRTL);
    {
   import library Strings;
20
   main()
       {
     marte.objects()[Foundations::Model]->map toRTLModel();
     marte.objects()[Foundations::Model]->map toRTL_PortType();
25 ...
   ...
   ...

       }
30   }
```

Here the transformation definition is like a simple class declaration: containing import statements, a signature and a *main* mapping entry point. This provides an analogy between QVTO and normal OOP languages. Transformations are instantiated during their execution and have related properties and (mapping) operations.

The *modeltype* declaration at the start of the transformation assigns a unique alias to a metamodel which is being utilized in the context of the transformation. The *uses* part of the declaration determines the name of the model and its associated URI. It should be mentioned that it is not necessary to place the URI within the parentheses following the model name.

A model transformation contains several *parameters* that indicate the input and output models created during the transformation. These parameters can have a direction: either *in*, *out* or *inout*. Parameters of the type *in* are not changed during a transformation; and mainly used for read-only purposes, while *out* parameters signify the newly created results. Finally the *inout* type allows to modify or update an existing model. This type of transformation is also called

an *in-place transformation* and is mainly used for model cleansing or refinement. Finally, only a single *main* operation is allowed per transformation.

8.1.3.2 Mapping operations

The *Mapping Operations* are considered as the fundamental behavior of the transformations. They take one or more source elements and return one or more target model elements. A mapping usually has a name and a type. Here in the example provided below, the mapping operation is of the type **Model** having a name **toRTL_PortType()**. An input object is referred in the mapping by the *self* keyword, which is the standard OCL namespace syntax. A mapping itself can have parameters and the result of the mapping is usually declared after the parameter list (there are no parameters present in the current example). In the following example, the return conforms to the type **mmRTL::RTL_ModelType**. The @ puts the contents of the rule in the *RTL_PortType_Model* model.

A mapping operation can also contain a *result* keyword that is used to reference the return object or a tuple (in case of multiple resulted objects). Before the mapping body are optional *when* (a precondition or guard) and *where* (post-condition) sections which evaluate contained Boolean expressions. If the *when* clause is evaluated to be false, the mapping is not executed and a return object is not created, resulting in return of a literal *null*. Within the context of a transformation, the *null* can comply to any type and signifies the absence of a value. It can be used as the return of an operation, either explicitly or implicitly.

```
  mapping Foundations::Model::toRTL_PortType() : mmRTL::RTL_ModelType @RTL_PortType_Model
{
  --This transformation generates the port types and then resolves their types
        init
            {
                log ('GASPARD2 SoC Co-Design Framework - RTL Chain');
                log('-- model transformations done via QVT Operation Language');
                log('1-Generating RTL Control/Data types from UML - MARTE specifications');
            }
        name := self.name;
        type := self.ownedElement[vsl::DataTypes::DataType]. map toRTLPorttype() ->asSet();
        ...
        ...
        ...
        end
        {
         self.ownedElement[VSL::DataTypes::DataType]->map resolveTypeEnumerate();
         self.ownedElement[VSL::DataTypes::DataType]->map resolveTypePrimitive();
         self.ownedElement[VSL::DataTypes::DataType]->map resolveTypeComplex();
        }
    }
```

8.1.3.3 Mapping body

Within the mapping body, optional *init*, *end* and *population* blocks can be created. The *init* and *end* blocks are not mandatory in a mapping operation; and the remaining area is treated as a *population*. Generally, it is not required to use an explicit *population* section.

An *init* section allow to explicitly create objects, and here computation is carried out to initialize variables, etc.; before the effective instantiation of the mapping output is carried out. An *init* can be used for instantiating an object which is a subtype of that defined as the result in the mapping definition. The output values are present in the *population* area, while the termination of the mapping and finalization of the computation occurs in the *end* section before the mapping is returned.

8.1.3.4 Merging and Disjunction

Mappings in QVTO can extend other mappings by means of inheritance, can have their results merged with the results of another mapping, or they can be executed on the basis of success on their guard conditions. The *merge* mechanism for the fusion of different metamodels in our design methodology (as explained in section 6.4.1) is carried out via a QVTO merge mapping.

A *Disjunction* permits to specify several mappings from a single root mapping, the mapping which is first executed is the one that satisfies the guard conditions (type and *when* clause) of the

root mapping. The different mappings are in the form of a list, after a *disjuncts* keyword in the mapping declaration. During the execution of the mapping operation, each guard is evaluated until one is found to be satisfied; resulting in the associated mapping to be carried out. If no listed mapping satisfies the conditions, a null is returned. An example of *disjuncts* is present below in the QVTO rules:

```
mapping CommonBehavior::Event::EventtoControlEvent() : mmRTL::TransitionCondition
--The disjuncts check the event types, and performs appropriate actions
disjuncts
        CommonBehavior::AnyReceiveEvent::AnyReceiveControlEvent,
        CommonBehavior::ChangeEvent::ChangeControlEvent
        {

        }


  mapping CommonBehavior::AnyReceiveEvent::AnyReceiveControlEvent() : mmRTL::TransitionCondition

--When event is of AnyReceiveEvent type, its value is set to all
        when{self.oclIsTypeOf(CommonBehavior::AnyReceiveEvent)}
                {
        name := self.name;
        value := 'all';
        ...
  ...
                }

  mapping CommonBehavior::ChangeEvent::ChangeControlEvent() : mmRTL::TransitionCondition
 --When event is of ChangeEvent type, its value is set to the literal string value
        when{self.oclIsTypeOf(CommonBehavior::ChangeEvent)}
    {
        name := self.name;
        value := 'when'+' '+self.changeExpression.oclAsType(VSL::LitteralValues::LiteralString).value;
        ...
  ...
    }
```

8.1.3.5 Helper operations

A helper operation such as a *query* is intended to help simplify expression writing in mapping operations. The only restriction related to queries is that they are not able to create/update object instances; other than for predefined and intermediate types (such as variables). In queries, it is possible to define and assign local variables. The main difference between an helper and a query is that a query does not have any side effects on the parameters passed into it. An example of a query in the *MARTE2RTL* transformation is given below:

```
--This query creates a default logic type when a port type is not specified for
the hardware accelerator, and inserts it in the output model

query StructuredComponent::createDefaultLogicType() : mmRTL::RTL_PortType
{
        var marteType := self.container().oclAsType(Foundations::Model).map
  createDefaultLogicType();

  var model := self.getTypesModel();
        model.type += marteType;
        return marteType;
}
```

8.1.3.6 QVTO: Summary

QVTO provides a rich collection of operations and iterators. Numerous imperative operations such as *while* and *switch* are also present in QVTO. Additionally, imperative functions of OCL can also be found in the language. *Resolution* operators such as *resolve*, *resolveone* and *late resolve* reference trace data to resolve objects, or objects which are utilized as the source of an object creation. QVTO also provides library operations that permit to manipulate objects. Moreover, operations on models themselves are provided (such as model creation, copying/removing elements from models, etc.).

Similarly, other types of operations such as String and List operations are also present in QVTO. We refer the reader to the Eclipse modeling project[3] for the detailed syntax of the QVTO

[3] http://www.eclipse.org/resources/resource.php?id=493

language. Some examples of QVTO based rules present in our transformation chain are presented subsequently.

8.1.4 Rule examples

We now provide a few examples of the model-to-model transformations present in our design flow. Only a few examples are presented, in order to describe the general mechanism of these transformations, instead of writing every related minute detail. The illustrated examples are related to the *MARTE2RTL* transformation in our design flow. We initially provide examples of transformation rules related to the hardware accelerator portion in the RTL metamodel, before moving onto the control aspects.

8.1.4.1 Repetitive component

In this section, we have chosen to illustrate the transformation related to a hardware repetitive component in the RTL model. We first present a reference example of such a component modeled in the Gaspard2 framework. Figure 8.6 illustrates a `RepeatedAdditionStep5` component, having input and output ports with shapes equal to 4 and 2 respectively, along with associated data types. The component contains an interior repeated task `a` of the type `Addition`, that is repeated 2 times and has its respective input/output ports. The input/output tiler connectors connect the respective ports of the `RepeatedAdditionStep5` to the port instances of the instance `a`.

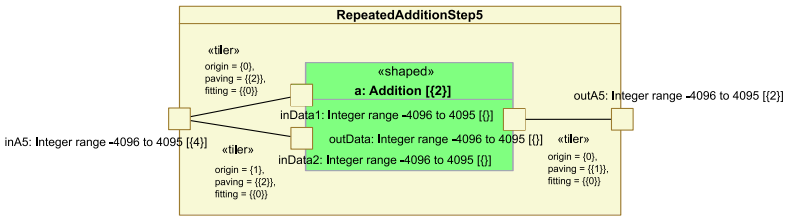


Figure 8.6: A repetitive Gaspard2 application component

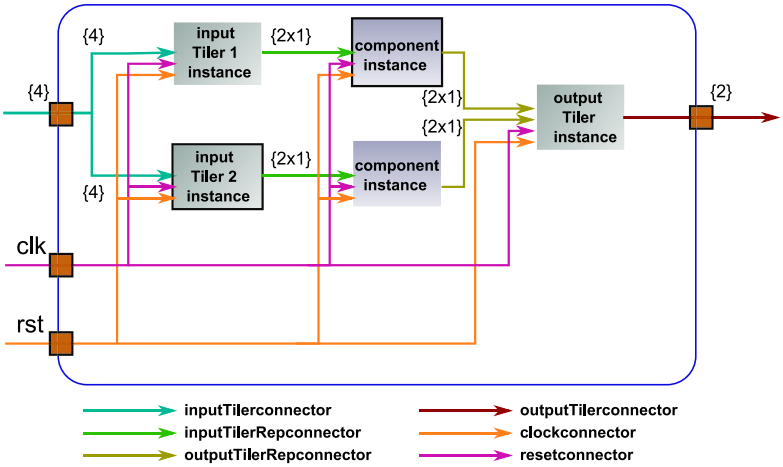


Figure 8.7: Generated result of the *MARTE2RTL* transformation. The ports of the input/output tiler instances and the component instance are not illustrated in the figure

As explained before, the tiler connectors are converted into hardware computation units due to the proposed hardware execution model for Gaspard2 applications. Thus in the *MARTE2RTL* transformation that takes the RTL model as input, each of the tiler connectors present in the model is transformed into the `RTL_Tiler` concept present in the RTL metamodel. The nature

of the `RTL_Tiler` (whether input or output) depends on the direction of the tiler connectors. If a tiler connector connects an input port of a component to an input port instance of its owned component instance, then an input tiler is created; while if a tiler connector connects the output port instance of a component instance to an output port of the owner component, the transformation creates an output tiler.

In the generated output result for the repetitive component, the tiler connectors are also instantiated as special component instances (tiler instances) that refer to their respective components (tiler components). Similarly, the repeated interior part is also instantiated having multiple repetitions and referencing its proper component. The Figure 8.7 illustrates the abstract representation of the output generated by the transformation. It should be mentioned that while this representation unrolls the multiple repetitions of the component instance `a` in order to facilitate the visibility, the actual output generates a single component instance having a RTL shape of 64. This shape value is taken by the corresponding model-to-text transformation to create the multiple repetitions (for VHDL, we use the GENERATE keyword for this purpose). This figure does not represent the electronic implementation result of the modeled application component, but is nearly an accurate structure. This proximity between the output of the model transformation (i.e., the RTL model) and the electronic level permits generation of the correct VHDL structure via the subsequent *RTL2CODE* transformation.

In order to achieve the desired structure, the transformation rule is written accordingly. Figure 8.8 shows the global overview of the transformation rule related to a hardware repetitive component. The *toHardwareAccRepetitiveComponent* rule takes an input pattern (a MARTE structured component having a repetitive structure); and checks its own associated conditions. If all the conditions are true, the target pattern: `HwRepetitiveComponent` is created. Additional sub rules present in this rule are subsequently invoked, resulting in creation of various elements of the output pattern such as clock and reset ports, input/output tiler instances and the component instance corresponding to the interior part of the modeled application component. Similarly, multiple connectors are created for connecting the port instances of the tiler instances and the application component instance, to the ports of the repetitive component. In the illustrated example, the clock and reset ports of the component instance are not created in the rule, but are created subsequently in a sub rule related to the owned component instance.

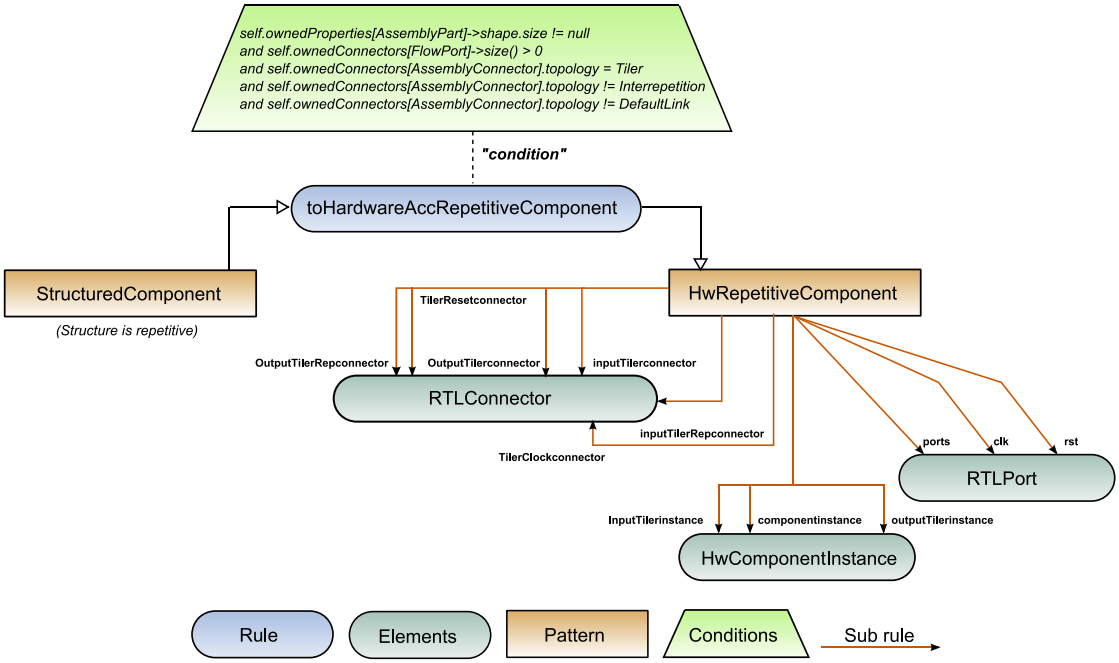


Figure 8.8: Transformation rules related to creation of a hardware repetitive component

Similarly as specified before, for a hardware accelerator component, clock and reset signals are mandatory; thus we need to create clock and reset ports for the different components; sim-

ilarly clock and reset port instances are created for the instances of these components. Clock and reset ports are equally connected for each tiler component (and its instances). Finally connectors are used to connect the various ports and port instances present in the repetitive component. The input/output *Tilerconnectors* are connectors having a shape value corresponding to the shape of the respective input/output ports of the repetitive component. While the input/output *TilerRepconnectors* help to connect the port instances of the (input/output) tiler instances to the different iterations of the component instance.

In the above example, an *inputTilerconnector* connected to the input tiler 1 instance has a shape value equal to 4 due to the equivalent shape value on the input port of the repetitive component. Correspondingly, the generated tiler component and its related tiler instances have input ports (or port instances) with the same shape to assure correct connectivity. The *inputTilerRepconnector* connected to the same tiler instance has a shape value equal to the repetition space of the component instance and the pattern shape of its input port instances. As in the following example, the repetition space of the component instance is 2 and pattern shape is 1; the corresponding shape of the *inputTilerRepconnector* is set equal to 2×1. Similarly, the shape of the output port (port instance) of the input tiler (tiler instance) is set to the same value. For an output tiler (and its instances), a reverse mechanism is adapted.

We now present an extract of the QVTO based transformation rule related to the repetitive component. This transformation calls several related sub rules in order to create the corresponding elements of the repetitive component.

```
mapping GCM::StructuredComponent::toHardwareAccRepetitiveComponent() :
mmRTL::HW_RepetitiveComponent

--The transformation checks that only tiler connectors are present in the repetitive
component and then generates clock, reset ports, the component instance, tiler
instances and associated connectors

when {
      self.ownedConnectors[GCM::AssemblyConnector]->size() > 0

      and self.ownedProperties[GCM::AssemblyPart].shape.size != null

      and   self.ownedConnectors[GCM::AssemblyConnector]->select(topology.oclIsTypeOf
      (RSM::linkTopology::InterRepetition))->isEmpty()

      and (self.ownedConnectors[GCM::AssemblyConnector]->asSequence()->at(1).
      topology.oclIsTypeOf(RSM::linkTopology::Tiler)

      or self.ownedConnectors[GCM::AssemblyConnector]->asSequence()->at(2).
      topology.oclIsTypeOf(RSM::linkTopology::Tiler)

      or self.ownedConnectors[GCM::AssemblyConnector]->asSequence()->at(3).
      topology.oclIsTypeOf(RSM::linkTopology::Tiler)

      or self.ownedConnectors[GCM::AssemblyConnector]->asSequence()->at(4).
      topology.oclIsTypeOf(RSM::linkTopology::Tiler))
     }

    {
        init
            {
            String.startStrCounter('Counting');
            }

        name := self.name;
        ports += self.oclAsType(GCM::StructuredComponent).ownedPorts.
        oclAsType(GCM::FlowPort).map toRTLPort();

        clk := self.map toHwClockPort();
        raz := self.map toHwResetPort();

        refComponentInstance := self.ownedProperties[GCM::AssemblyPart]->asSequence()
        ->first().oclAsType(GCM::AssemblyPart).map toHwRepComponentInstance();

        refTilerInstance += self.ownedConnectors[GCM::AssemblyConnector]->map
        toRTLInputTilerInstace();

        refTilerInstance += self.ownedConnectors[GCM::AssemblyConnector]->map
        toRTLOutputTilerInstace();

        connector += self.ownedConnectors[GCM::AssemblyConnector]->map
        toRTLConnectorInputTiler(String.incrStrCounter('Counting'));

        connector += self.ownedConnectors[GCM::AssemblyConnector]->map
```

```
        toHwRepetitiveInputTiler(String.incrStrCounter('Counting'));

        connector += self.ownedConnectors[GCM::AssemblyConnector]->map
        toRTLConnectorOutputTiler(String.incrStrCounter('Counting'));

        connector += self.ownedConnectors[GCM::AssemblyConnector]->map
        toHwRepetitiveOutputTiler(String.incrStrCounter('Counting'));

        connector += self.ownedConnectors[GCM::AssemblyConnector]->map
        toRTLTilerClockConnector();

        connector += self.ownedConnectors[GCM::AssemblyConnector]->map
        toRTLTilerResetConnector();
        ...
        ...
        }
```

8.1.4.2 Input Tiler component for Hardware accelerator

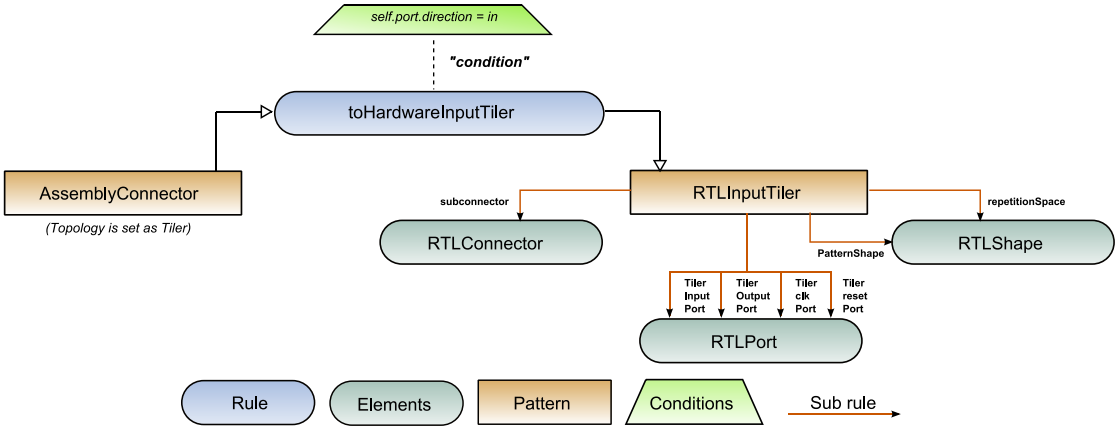


Figure 8.9: Transformation rules for the hardware Tiler component

Figure 8.9 shows the transformation rule related to the input tiler for the hardware accelerator. A similar approach is adapted for transformation of an output tiler. The *toHardwareInputTiler* rule takes an *AssemblyConnector* (having a stereotype equal to a tiler, as specified in the UML MARTE diagram) from the MARTE model and converts it into a tiler component. The generated input tiler component has its specific repetition space and pattern shape. It also has several ports: the input and output ports. When the tiler is instantiated in a repetitive component, the equivalent created corresponding port instances are linked to the *inputTilerconnector*(s) and *inputTilerRepconnector*(s) respectively.

Similarly a tiler component has its own clock/reset input ports. Finally the tiler component contains subconnectors that enable to determine the data dependencies related to data parallelism in a Gaspard2 application. The computation related to the subconnectors is carried out via the ADO pre-computations, specified in section 8.1.5; and are invoked by the *toInputTilerSubConnector* subrule in the mapping function. Similarly, the other different elements related to the input tiler are created via different sub rules, called from the *toHardwareInputTiler* rule:

```

--The transformation checks the condition that the tiler is of the direction in,
and then creates input, output, clock and reset ports for the tiler component,
and also calls a subrule for ADO pre-computations

mapping GCM::AssemblyConnector::toHardwareInputTiler(id : Integer) : mmRTL::RTL_InputTiler
        when{
                self._end->asSequence()->first().endPort.oclAsType(GCM::FlowPort).
    direction = GCM::DirectionKind::_in
                }
{
        init {
                var portEnd : Extension::AssemblyConnectorEnd := self._end[assemblyPart
    = null]->asSequence()->first();
```

```
    var portInstEnd : Extension::AssemblyConnectorEnd := self._end
    [assemblyPart != null]->asSequence()->first();

    var nametemp := 'TilerIN_' + id.toString()+portEnd.endPort.name.repr();
       }

  name := 'TilerIN_' + id.toString()+portEnd.endPort.name.repr();
  clk := self.map toHwClockPortConnector();
        rst := self.map toHwResetPortConnector();
        ports += portEnd.map toInTilerPort(nametemp);
        ports += portInstEnd.map toInTilerInstPort(nametemp);
        patternShape := portInstEnd.map toHwPatternShape();
        repetitionSpace := portInstEnd.map toHwRepetitionSpace();
        subConnector += self.map toInputTilerSubConnector();
        ...
        ...
}
```

8.1.4.3 ControlNode component

As seen in Figure 8.5, a control node in the control portion of the RTL metamodel contains its respective ports as well as a behavior. The behavior differentiates a control node from the elementary component in the hardware accelerator portion in the RTL metamodel. These elements associated to a control node are called by their respective sub rules.

We now present a rule hierarchy related to the control node, corresponding to the hierarchy illustrated in Figure 8.5. The *toControlNode* rule takes as input, a structured component in the MARTE model having an associated behavior; and converts it into the control node pattern. Afterwards it calles the *toRTLPort* and *toControlBehavior* sub rules. The latter itself calls the *toControlAutomaton* rule in order to create the associated automaton. This succession of rules continues as the automaton rule invokes the sub rules related to transitions and vertices by means of the *TransitiontoControlTransition* and *VertextoControlVertex* rules. We do not detail the whole hierarchy for creating the automaton related to the control node, but only describe some of the above mentioned rules, as illustrated below:

- Mapping rule for **Control Node**:

```

--the rule checks the condition that there is an associated state graph
with the structured component and calls a behavior as a subrule

mapping GCM::StructuredComponent::toControlNode() : mmRTL::ControlNode
    when {
          self.ownedBehavior[CommonBehavior::StateGraph]->size() != 0
         }
    {
        init{
                var nom := self.name;
                }
        name := self.name;
        ports += self.oclAsType(GCM::StructuredComponent).ownedPorts.
        oclAsType(GCM::FlowPort).map toRTLPort();


        ownedBehavior += self.ownedBehavior[CommonBehavior::Behavior]->
        map toControlBehavior();
        ...
        ...
    }
```

- Mapping rule for **Behavior**:

```

--the rule in turn creates an automaton

mapping CommonBehavior::Behavior::toControlBehavior() : mmRTL::Control_Behavior

  {
        init
        {
                log('Entering state graph');
      ...
      ...
        }
```

```
        name := self.name;
    automaton += self.subobjects()[CommonBehavior::Region]->
    map RegiontoControlAutomaton();
...
...

  }
```

- Mapping rule for **Automaton**:

```

--the rule calls subrules for creating states and transitions

mapping CommonBehavior::Region::RegiontoControlAutomaton() :mmRTL::Automaton
{


name := self.name;
transition += self.subobjects()[CommonBehavior::Transition]->
map TransitiontoControlTransition();

state += self.subobjects()[CommonBehavior::Vertex]->
map VertextoControlVertex();

...
...

}
```

8.1.5 ADO pre-computations for tiler components

As defined earlier, the tiler connectors in the high level model are converted into components by means of the *MARTE2RTL* transformation. For the hardware accelerator part, these components are transformed into hardware computational units (HCUs), permitting the hardware execution of data parallelism. They thus express the data dependencies expressed by the MARTE tiler connectors, and help to connect each element in the input/output pattern to a data element in its respective input/output array. This process is called the *ADO pre-computations* because the data dependencies are computed during design time as compared to run-time implementation on an FPGA. The pre-computations of an input tiler are symmetrical to the pre-computations associated with an output tiler, hence here only pre-computations related to the input tilers are demonstrated . Initially each of the tiler connector in a hardware RCT is transformed into a tiler component, and is subsequently instantiated in the RCT, as shown earlier in Figure 8.7.

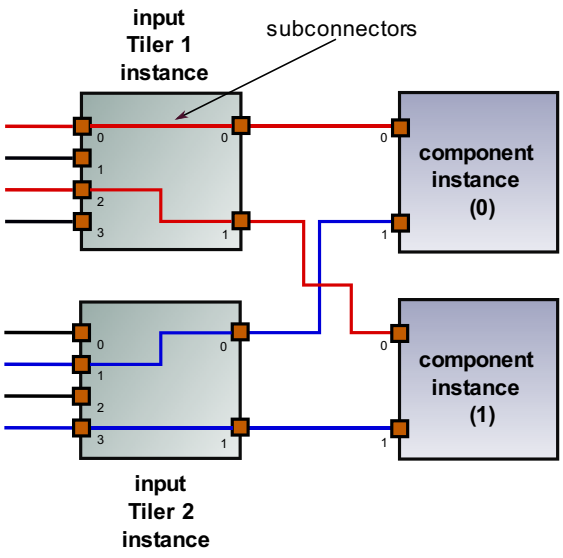


Figure 8.10: Tiler pre-computations: creation of an interconnection topology to determine data dependencies

The result of the pre-computations is the creation of an interconnection topology inside a tiler component; permitting to connect the input pattern(s) required by a repetition of a component instance, with the corresponding input array(s); by means of the information related to a tiler connector (i.e., origin, paving and fitting values). Other information relevant to the pre-computations are the shape values associated to the input array(s), the repetition space and the pattern shape of the repeated task. In this section, we do not describe how the pre-computations are carried out via the model transformations (these technical details are presented in appendix C), but instead only focus on its mechanism.

Figure 8.10 shows the result of the tiler pre-computations related to the tiler connectors expressed in Figure 8.6. Each input tiler component contains 4 input ports, due to the equivalent shape of the input array. Similarly, each input tiler component contains 2 output ports, i.e. (2×1); due to the result of the shape value of the repetition space of the component instance, and the mono-dimensional pattern shape associated to its port instances. This corresponds to the explanation given in section 8.1.4.1. The pre-computations determine the data dependencies based on the associated information given to each input tiler connector, and creates an interconnection between the respective source and target ports of each RTL input tiler component. The connectors used in the interconnection topology are the *HW SubConnectors* which have been introduced previously in section 7.3.2.8.

The algorithm related to the pre-computations first iterates on the (multidimensional) repetition space of the RT in a RCT, as well as each element present in the (multidimensional) pattern shape by means of two loops. For each element in a (multidimensional) pattern, a subconnector with an index value is created. A subconnector connects to its respective input/output ports in a tiler component due to the Array-OL mathematical expressions related to the origin, paving and fitting values. These expressions have been presented in detail in [36]. The index value related to source port of a subconnector determines the corresponding element in the input array, while the index value of its target port defines the position of the element in the pattern which is consumed by an iteration of the RT.

In [143], the authors proposed a similar approach for expressing data dependencies related to the tiler components, however, the proposed approach used Java based external black boxes integrated in their respective model transformations. This resulted in increased complexity of the transformation rules and required the developer to be expert in several languages. As compared to the above mentioned approach, the ADO pre-computations in our design flow are implemented by using QVTO based transformations, thus simplifying the effort for developing the corresponding model transformations.The QVTO rules for expressing the pre-computations have been described in appendix C.

8.1.5.1 Sliding windows data dependencies

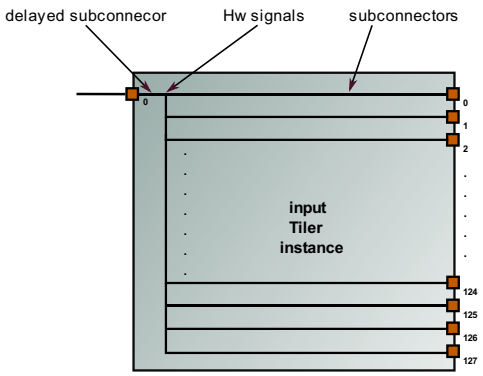


Figure 8.11: Tiler pre-computations: mechanism for sliding window data dependencies

The tiler connectors also allow to express special types of data dependencies, such as those related to temporal sliding windows. This type of data dependency is frequently used in

the domain of DIP applications, because it enables to filter the signals with respect to time. The ADO pre-computations also take this type of data dependency into account. Figure 6.28 shows an example of a tiler connector expressing a temporal sliding window. The tiler connector expressing this data dependency is the one connecting the `InDataTRM` port of the `TimeRepeatedMultiplicationAddition` component to the input port instance `InDataM` of the instance `rm` of the type `RepeatedMultAdder`.

Here the tiler connector resends the position of an element in an infinite data flow to the different iterations of the RT in the RCT. The pre-computations algorithm creates a special connector between an element of the pattern at an instant *t* to an element of the input array at the instant *t-n*. The model transformations generate a tiler component having an input port with a shape value equal to 1, corresponding to the infinite data flow. The output port of the tiler component has a shape value equal to 128, corresponding to the pattern shape consumed by the component instance. The tiler component thus connects the data present on its input port at instant *t*, delays that data during *n* cycles, and sends the data to one of its output ports. In the model transformation, this mechanism is achieved with the help of *HW Delayed SubConnector*, *HW Signal* and *HW SubConnector* concepts. A delayed subconnector is attached to different repetitions of an HW signal, on the basis of an associated shape value. Finally, each iteration of the signal is connected to a unique subconnector. Figure 8.11 shows the interior details of a tiler component that realizes this type of data dependency.

In a hardware functionality, this sliding window data dependency mechanism can be performed adequately by the utilization of shift registers. Thus during implementation in a target FPGA, shift registers are present in the tiler component for the execution of this data dependency, as illustrated in chapter 9.

The ADO pre-computations thus manage the data dependencies related to the data parallelism expressed in Gaspard2 applications at the high modeling level. The computations also takes into account, certain temporal dependencies such as related to sliding windows. The pre-computations step does not place a restriction on the number of dimensions present in a pattern or array. Thus data dependencies on multi-dimensions are also managed.

8.1.6 Advantages of QVTO over third party model-to-model transformation languages

As stated in the previous section, the current model-to-model transformations from a UML MARTE profile diagram to the eventual RTL model have been implemented with the QVTO language. An initial version of the RTL transformation chain has been developed for Gaspard2 in [143], however it has several drawbacks. Firstly, the initial model is not based on the MARTE profile, but instead the authors use their proper profile which is not an industry standard. The UML model is transformed into several intermediate models, followed by the eventual transformation into a static RTL model. This RTL model lacks sufficient details to completely generate the HDL code for a dynamic hardware accelerator. Secondly the model transformations are based on the internally developed MOMOTE transformation engine.

The advantage of QVTO over third party transformation languages is that it provides a proper syntax for writing the transformation rules. Thus developers always have a strong structure to follow, in order to develop the corresponding transformation rules. Moreover, as QVTO utilizes imperative Eclipse OCL constructs (for loop support, exceptions, variable initialization, etc.), a transformation rule written in QVTO is more comprehensible and requires less development effort as compared to its counterpart written in any other language.

Additionally, as QVT is an industry standard, QVTO based transformations can be understood by different designers and research teams, increasing synergy. Currently, the only drawback of current QVTO based transformations is that while they do support black box calls in the specifications, currently to this date, they cannot be implemented.

8.2 Code generation

This section provides the code generation stage of our design flow, related to hardware accelerator and a reconfigurable controller (for a partially reconfigurable SoC), from the RTL model. This generation is carried out by means of a *model-to-text* transformation, termed as *RTL2CODE* in our design flow. This transformation is represented by the eclipse in Figure 8.12, and currently generates VHDL code for the hardware accelerator(s) part of our design flow, as well as C/C++ code that is taken as input by the controller. The code related to hardware accelerator can also be executed in isolation, without regards to dynamic reconfiguration. However, this causes the creation of a static single hardware accelerator.

8.2.1 MDE code generation principles

According to the principles provided by MDE, the code generation can be viewed as *rewriting* a model in a textual form. Here, the concepts present in a model are not transformed, but translated into text, which is then utilized by the usual tools (compilers, simulators, etc.). For an effective code generation, it should be guaranteed that the abstraction level of the input model is close to the generated text. Thus, code generation can be viewed as a *one-to-one* transformation, in which each concept in the model generates certain part of the overall text. Hence code generation consists of producing, for each available concept, the syntax corresponding to this concept in the targeted language.

Presently, a large number of model-to-text transformation tools exist in literature. Additionally, OMG has proposed the MOF2Text standard[4] for these types of transformations. In the standard, a metamodel has been defined that helps in the development of these transformations by the presence of *patterns* of code generation. Similar to QVT, the tools respecting this standard should be capable of successfully reading a model before interpreting it in the eventual code. Nonetheless, currently there is not a single unique tool which completely implements the standard, yet the common idea in each tool is the presence of the patterns of code generation. Similarly tools such as Acceleo [7], MOFScript[5] and M2T[6] are also based on these code patterns.

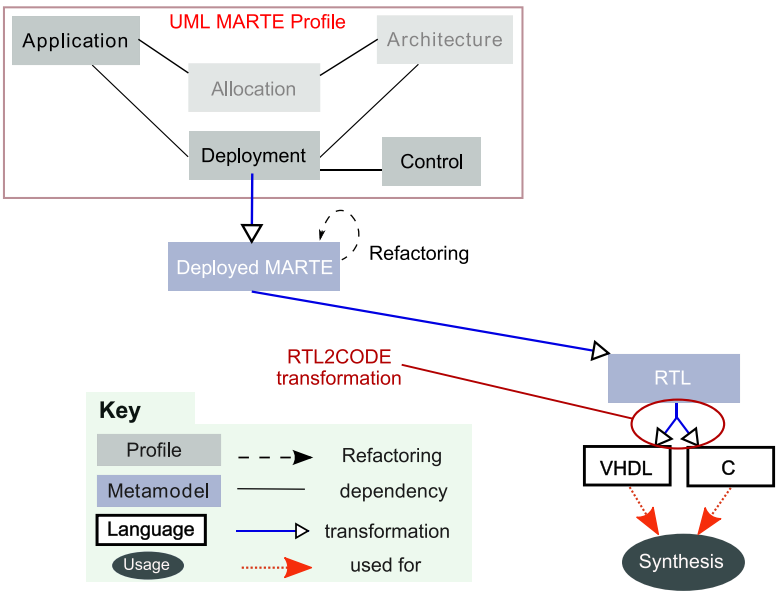


Figure 8.12: The *RTL2CODE* model-to-text transformation chain in our design flow

[4] http://www.omg.org/cgi-bin/doc?ad/2004-4-7
[5] http://www.eclipse.org/gmt/mofscript/
[6] http://www.eclipse.org/modeling/m2t/

8.2.1.1 JET Templates

Currently in the Gaspard2 framework, for model-to-text transformations, we have selected the usage of JET (Java Emitter Templates) [82] from the M2T project. Besides its capacity to be easily integrable in the Eclipse environment, we find the notion of code patterns (which is common among different model-to-text transformation engines).

8.2.1.2 Syntax of JET templates

In JET templates, we mainly find two types of elements: 1) the text which is written directly into the required output code (VHDL and C/C++) in this case and 2) the Java code which is called by delimiters <% and %>. The Java code can be itself of two different natures:

- <% SCRIPT %>: The script is directly produced in the intermediate Java class. This script serves for analysis; such as evaluating loops whose length depends on the input model.
- <%= EXPRESSION %>: The expression generally depends on the input model. For example, it is possible to write the name of a concept: *RepeatedMultAdder*, by means of the expression <%= REPEATEDMULTADDER.GETNAME()%>. It is also possible to call sub templates corresponding to other concepts: <%= TS.GENERATE(REPEATEDMULTADDER.GETPORTS()) %> calls a template corresponding to a port notion related to current *RepeatedMultAdder* concept by means of reference *Ports*.

In the Gaspard2 framework, we do not directly utilize JET, but take advantage of an additional layer that links to Ecore, the Eclipse modeling technology; for simplifying JET utilization. This software layer has been developed internally in the DaRT research team and has been termed as MoCodE (Model to Code Engine). This layer also permits generation of the output code in several separate files, which is not possible with JET.

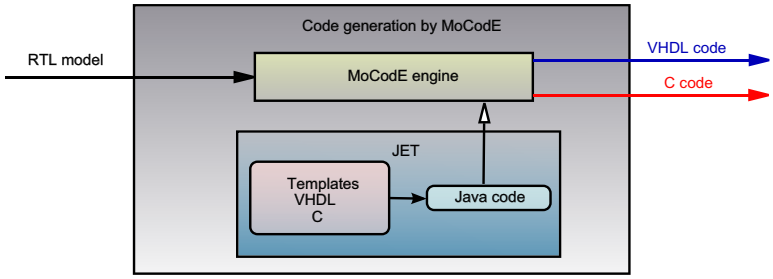


Figure 8.13: Conception flow for JET

Figure 8.13 represents an abstract representation of JET functionality in the Gaspard2 environment along with the MoCodE software layer, when a model (or models) (at the left side of the figure) is transformed into text (right side of the figure). A *JET Generator* permits loading of the initial model(s), and applies *JET Genlets* before saving the generated code in a file. A *JET Genlet* is a Java class having a *GENERATE* method, that takes the arguments of the model (or a part of the model); and returns a string of characters. These genlets are produced from *JET Templates*: scripts that specify the relationships between a concept in the input model and the output code.

The basic principle of MoCodE is to associate a source code pattern with each element in the input metamodel. At the start of execution, only the root of the model is transformed. The code present in the pattern traverses the model and seeks the transformations of the desired elements, with the help of the *GENERATE* function. Gradually, each of the different patterns scans a portion of the input model and call other patterns. The transformation engine determines the pattern corresponding to an element while searching for the pattern with the same name as the type of element processed. Moreover, a developer can identify the elements that are to be generated in their respective separate files. For other types, the string obtained by the execution of a pattern on a given element is inserted into the file where its generation has been called. The mechanism of writing a pattern is inspired from web technologies such as PHP and JSP.

The code generation in JET is carried out by JET generator, while the templates are managed by MoCodE. MoCodE also enables optimization of the templates, for example, enabling hierarchy between concepts. We now present the choice of generating VHDL and C/C++ code directly from the RTL model, instead of moving towards their respective metamodels.

8.2.2 Possible choices for code generation

A RTL model is not directly exploitable by classical tools for simulation and synthesis purposes; contrary to the output produced code. Two approaches are possible for code generation, which are explained below:

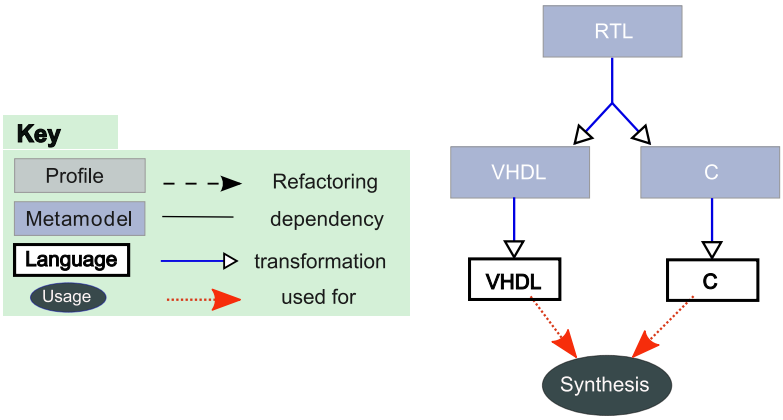


Figure 8.14: An alternative choice from the RTL metamodel

First solution: Creation of separate metamodels that respect the syntax of VHDL and C/C++ languages respectively. This is turn forces two additional model-to-model transformations, from the RTL model to the respective *Code* models as shown in Figure 8.14. From these models, eventual code generation can be carried out.

Second solution: Generation of code directly from the RTL model. The advantage of this method is that no other intermediate metamodels need to be inserted in our design flow. The abstraction level of the RTL model is sufficient enough for the generation of VHDL and C/C++ code for our requirements.

The first solution introduces certain disadvantages. Firstly, the development effort is increased as additional metamodels and corresponding transformations are introduced in the design flow. Additionally, for a designer wishing to generate the syntax of the hardware accelerator in another HDL such as Verilog, he will need to create an additional metamodel respecting Verilog syntax. Similarly, for the reconfiguration controller part, we have stated that the state machine code can be either generated in C/C++ or VHDL depending upon the choice of the selected reconfiguration. In case where both C/C++ and VHDL metamodels are present, either the controller concepts have to be duplicated in both metamodels, or present only in a unique metamodel. This decreases the flexibility currently present in our design flow.

For these above mentioned reasons, the second solution has been adapted, resulting in code generation directly from the RTL model by means of the *RTL2CODE* transformation. This solution is more generic in nature, but also decreases the development and maintenance efforts demanded of the developers.

8.2.3 VHDL code generation for hardware accelerators

One of the objectives of design flow is the generation of *correct* and *synthesizable* VHDL code which can be taken as input by commercial synthesis tools for implementation in a FPGA. This VHDL code corresponds to the different implementations of the modeled application, or the

various configurations of a dynamically reconfigurable hardware accelerator. In order for the output VHDL code to have the same characteristics, such as data and task parallelism that were present in the initial application model, we have imposed certain conditions which are taken into account during the writing of the JET templates. Some of the critical conditions for code generation are given below:

Accelerator Structure: The structure of the hardware accelerator(s) written in VHDL should be equivalent to the modeled application at the MARTE profile level. Thus application hierarchy is conserved in the final code.

Code generation for accelerator components: Each component in a configuration of the hardware accelerator is written in a separate file, the name of the file corresponding to the name of the component.

Expressed parallelism: The data parallelism of the Gaspard2 applications is expressed in the accelerator(s) by means of the VHDL GENERATE keyword. Similarly, multidimensional ports are not linearized during the code generation phase.

Configurations: Depending upon the number of configurations at the high modeling level, we find the same number of hardware accelerator implementations. A separate folder is created for each configuration/implementation of the hardware accelerator. Although for each configuration, a large part of the generated VHDL code (top level code and the code corresponding to the instantiated sub components) remains the same, the code corresponding to the elementary components is changed for each configuration. This choice was selected to remove ambiguity and to ease the creation of partial bitstreams. While it is possible to create only one hardware accelerator and manually change the implementations related to the elementary components; and afterwards create different configurations in an non-automatic manner; this is a tedious task which augments in complexity depending upon an increase in the number of elementary components or configurations.

We now move onto providing some examples of the templates utilized for the code generation. In the following sections, we present an example related to a generic hardware accelerator component, followed by an example related to a repetition context task that expresses data parallelism in a hardware accelerator.

8.2.3.1 Code generation for hardware accelerator components

This subsection provides the basic template for generation of a typical component in a hardware accelerator, irrespective of its nature (compound, repetitive or elementary). This template represents the black box representation of the component by illustrating only its interfaces. The interface of these components is determined by their input/output ports which can be multidimensional in nature. From the point of view of code generation, a template that creates the code for a component invokes the sub templates for creating the ports of the component.

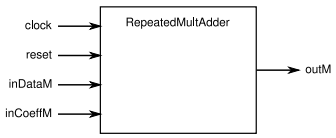
We now present an example of a template associated with an INPUTPORT. This template generates the name of the port, followed by inserting the keyword **: IN**; and terminates by calling another template for determining the port type.

```
<%=element.getName()%> : IN <%=ts.generate(element.getType())%>
```

Thus the construction of a component interface requires the utilization of templates for creating input and output ports, as illustrated above. Whatever the type of the port and the manner in which it is referenced by a component, the mechanism of code generation is similar by the basis of the TS.GENERATE() keyword. TS.GENERATE() calls the template corresponding to the element, in its parameter. In this case related to interface generation, the parameter is either an input or output port. The template responsible for generating the interface of a generic hardware accelerator component is as follows:

```
ENTITY <%=element.getName()%> IS
PORT
(
<%=ts.generate(element.getClock())%>;
<%=ts.generate(element.getReset())%>
<%for (Port p : (List<Port>) element.getPorts())
%>;
  <%=ts.generate(p)%><%
%>);
END <%=element.getName()%>;
```

In the RTL metamodel, the clock and reset ports of a hardware accelerator component are referenced by *clk* and *rst* metarelations allowing for creation of these specific ports. The code generation of these ports is separated from normal modeled ports referred in the metamodel by means of the *ports* reference; and is carried out due to the line 6 in the template. Figure 8.15 exhibits the code generation for a component having 4 input ports and 1 output port. The MARTE based modeling of this component has been previously illustrated in Figure 7.3.




```
1  ENTITY RepeatedMultAdder IS
   PORT
   (
   clock : IN STD_LOGIC;
5  reset : IN STD_LOGIC;
   inDataM : IN TABLE_TYPE_128_Integerrange8to7;
   inCoeffM : IN TABLE_TYPE_128_Integerrange1to1;
   outM : OUT TABLE_TYPE_64_Integerrange4096to4095);
   END RepeatedMultAdder;
```

Figure 8.15: The left side of the figure represents model of a component in the RTL metamodel, while the right side demonstrates the code generated with the aid of the template described previously

This template permits to create the interface related to a hardware accelerator component in the RTL metamodel, called ENTITY in VHDL. The generation of the code related to the behavior of this component depends upon the type of the component itself. In case of an elementary component, a related IP (corresponding to the associated configuration) is instantiated inside the component, along with a mapping between the ports of the two components. For a compound component expressing task parallelism, component instances are created along with connectors, etc. The next section describes the template corresponding to the code generation of loops present in a repetition context task (RCT) having a repeated task (RT).

8.2.3.2 Code generation for loops in RCT related to the hardware accelerator

The creation of loops in a RCT depends on the *shape* (i.e., the repetition space) associated with a component instance (RT) in the RCT. This shape helps in the instantiation of an RT. The illustrated code represents a part of the template related to the code generation of a *HW Repetitive* component, for enabling multiple instantiations of its interior repeated task.

```
<%int indexRepetition = 0;
for (Integer v : (List<Integer>)  ...
     ... element.getRefComponentInstance().getDim().getValue()){
%>genit<%=indexRepetition%> : for ...
            .....it<%=indexRepetition%> in 1 to <%=v%> generate
  <% indexRepetition++;
}%>

<%=ts.generate(element.getRefComponentInstance())%>

<% indexRepetition = 0;
```

```
for (Integer v : (List<Integer>) ...
      ... element.getRefComponentInstance().getDim().getValue())
%>end generate;
  <% indexRepetition++;
%>
```

The following code presents the generated code for the RCT *RepeatedMultAdder* showed earlier in Figure 7.3. As the RCT has a RT with a repetition space of a single dimension, only one loop is present in the VHDL code, permitting to instantiate multiple repetitions of the RT, equivalent to the shape of its associated repetition space. The port mapping for the different repetitions depends on a sub template, which is illustrated in line 5 of the template written above.

```
1
genit0 : for it0 in 1 to 64 generate

...
5
end generate;
```

It should be made clear that we have only illustrated an example for a RCT having RT with a mono-dimensional repetition space. A repetition space having N dimensions generates N nested loops in the VHDL code for a RCT, provided that no dimension is an infinite temporal one. For an infinite repetition space associated with a RT in a RCT, such as the *RepeatedMultAdder* repeated task in the *TimeRepeatedMultiplicationAddition* component in Figure 6.28, the illustrated template generates the VHDL code illustrated below. Hence at each rise of the clock, a new instance of the *RepeatedMultAdder* is generated, for a sequential temporal execution:

```
1
genit0 : for it0 in 1 to 1 generate

...
5
end generate;
```

8.2.4 Code generation for reconfiguration controller

This subsection deals with code generation of the controller managing the context switch related to the different implementations of the hardware accelerator. The generated code is in the form of C/C++ language due to the choice of utilizing an internal embedded processor in the target FPGA, as specified earlier in chapter 5.

The modeled automata is transformed into a state machine in C/C++, with continuous infinite transitions in order to be an equivalent transformation of the mode automata.

8.2.4.1 Converting mode automata model into code

For the code generation, in Figure 8.16 we first present an abstract representation of the mode automata presented earlier in Figure 6.34. It is obvious that this representation is not an UML diagram, but is illustrated to provide a general global overview of the mechanism related to code generation of mode automata.

The *MARTE2RTL* model transformation presented earlier in the chapter, converts all the modeled concepts present in the high level models into their near equivalent concepts in the RTL model (such as states, collaborations etc). For code generation, MoCodE based templates are written to parse through these concepts, in order to get the required information for the generation of the controller code.

Initially the base template is called from the top hierarchical level control compound component, i.e., the *Deployed Automata* component representing the modeling of the mode automata. From this template, sub templates/rules are invoked for computing different operations, such

as determining the number of present states, events etc. We now present the basic steps related to code generation:

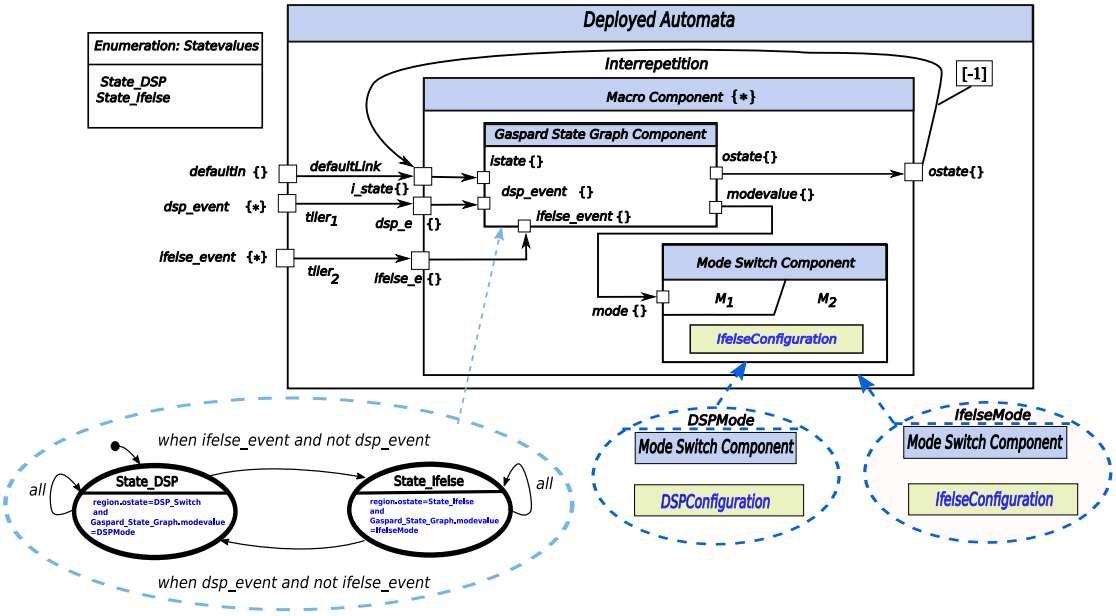


Figure 8.16: Abstract overview of the deployed automata

Information about states. Initially the information related to the different states present in a Gaspard state graph is analyzed. An initially invoked template searches the modeled enumeration representing the different states in the state graph and generates a corresponding ENUMERATION in the final code, having entries equivalent to the enumeration literals of the modeled enumeration. In this case, the enumeration `Statevalues` is converted into an enumeration in the generated code.

Event ports. Similarly, another enumeration related to the event ports of the mode automata is created in the code. A template returns the list of the names of the modeled input event ports. While at the modeling level, all event ports are of the Boolean Type; during the code generation in C/C++, the name of the event itself is taken into consideration. While it is also possible to utilize the same methodology at the high modeling level, this complicates the design specifications and the corresponding model transformations. Additional semantics have to be integrated into the RTL metamodel to take into account details: such as creation of a special type of port having no associated type. In contrast, associating the Boolean type assigns a unique type to every input/output event port, helping in its identification in the corresponding model transformations; and serves its purpose for the general control model introduced in chapter 6.

Relating configurations with a state. A state machine in C/C++ can be generally implemented by means of nested switch-case statements. For a switch-case construct, each case can relate to a specific state in the Gaspard state graph. Each of these cases calls its respective nested switch-case construct for handling the events arriving on that particular state. The arriving events result in a transition (self transition or transition to another state). In case of a self transition, no action is carried out, while in a transition to other state, the associated function/doActivity is taken into account. In the model-to-text transformations, a template is called that determines the mode value of the resulting state and compares that value with the name of the collaborations attached to the mode switch component. In case of a match, a sub template determines the name of the configuration present in the related collaboration, the related name is subsequently used in the nested switch-case statement as the operation to be carried

out. This operation can be viewed as changing the partial bitstream related to an implementation of the hardware accelerator, where the name of this bitstream corresponds to the name of the configuration.

Creation of an empty configuration. The generate code also contains a special case related to an empty hardware accelerator configuration which is normally created by the synthesis tools carrying out partial reconfiguration. This is done to reduce power consumption levels in the dynamically reconfigurable system. As this configuration does not contain any interior logic related to the hardware accelerator, it is not modeled at the deployment level. The modeling of an empty configuration can complicate the high level modeling and the associated transformations. Henceforth, the *RTL2CODE* transformation explicitly adds information related to this concept for the final code generation. This step has not been illustrated in the template extract given below, but is illustrated in the next chapter.

Conversion of defautLink, tiler and interrepetition connectors. The defaultLink and interrepetition components in the RTL model are converted into variables that help to determine the initial, current and next states in the state machine. As we only deal with flat state machines, that have no embedded hierarchical state machines, mechanisms related to the history are not necessary; and consequently are not created. This choice is explained earlier in chapter 6, as the states represent the global configurations related to the modeled application.

As compared to the above mentioned dependencies, the tiler components related to the control model are collectively transformed into a VHDL component that allows to represent an infinite flow of control events. This is due to the reason that control events are generally unpredictable and can arrive at any time instant, while the control model introduced in Gaspard2 introduces a control flow having an infinite temporal dimension. This point has been illustrated in the next chapter. An extract of JET template related to the state machine is given subsequently:

```
<% integer caserepetition = 0;
for (Integer cr: (List<Integer>) element.getEnumState.literalvalue.listvalue()){
%>Case <%=ts.generate(Enumstate.getName())%> :

  Switch (e)

      <% integer nestedcaserepetition = 0;
      for (Integer cr: (List<Integer>) element.getEventPorts.listvalue()){.......
     %> Case <%=ts.generate(Event.geteventName())%> :
      ...
      ...
      ...
      break;
      <% nestedcaserepetition++; } %>

      default;
theStateAfterTransition =  <%=ts.generate(Interrepetition.getNextState())%>;

      break;


  break;
  <% caserepetition++; %>

  ...
  ...
}%>
```

Here *e* denotes the enumeration related to the list of event ports. The given example has two nested for loops, each related to a case statement, the first loop is related to the case associated with the different possible states, while the second is related to the events arriving at a particular state. It then calls subsequent sub templates to determine the action related to a specific event.

Afterwards, templates related to the empty configuration and other parts of the controller code help to complete the state machine part of the controller. An extract of the resulting code from the root template related to the modeled deployed automata is given as follows:

```
{
 case State_DSP:
    switch(e)
    {
        case 'ifelse_event':
         // Change current configuration to IfelseConfiguration
         theStateAfterTransition = State_Ifelse;
         ..
         ..
        break;

        default: //  self transition in case of a random event/all
        theStateAfterTransition = State_DSP;
                    break;
    }
    break;

    case State_Ifelse:
    switch(e)
    {
        case 'dsp_event':
         // Change current configuration to SwitchConfiguration
         theStateAfterTransition = State_DSP;
         ..
         ..
        break;

        default: //  self transition in case of a random event/all
        theStateAfterTransition = State_Ifelse;
                    break;
    }
    break;

 }

...
...
```

It should be mentioned that the code illustrated above has been simplified for user comprehension. The details related to the actual configuration switch functions and the empty *blank* configuration have not been presented here; and are detailed in the next chapter.

8.3 Synthesis results

In the previous sections, we have detailed the creation of the tiler components that determine the data dependencies in the dynamic hardware accelerator generated from our design flow. Once VHDL code related to these components is generated from the RTL model by means of the *RTL2CODE* model-to-text transformation, the code can be verified by synthesis in traditional commercial tools such as Xilinx ISE. Figure 8.17 shows the equivalent synthesis result of the modeled application component illustrated in Figure 8.7.

The generated code for the two input tilers related to this synthesis result is presented below. The component TILERIN_41INA5 represents the tiler connecting to the `inData1` port instance of the repeated component instance, while the second tiler connector is represented by the component TILERIN_26INA5. The tilers are assigned unique names, along with input/output ports for incoming and outgoing data. They also have clock and reset ports (that are not utilized for the hardware functionality).

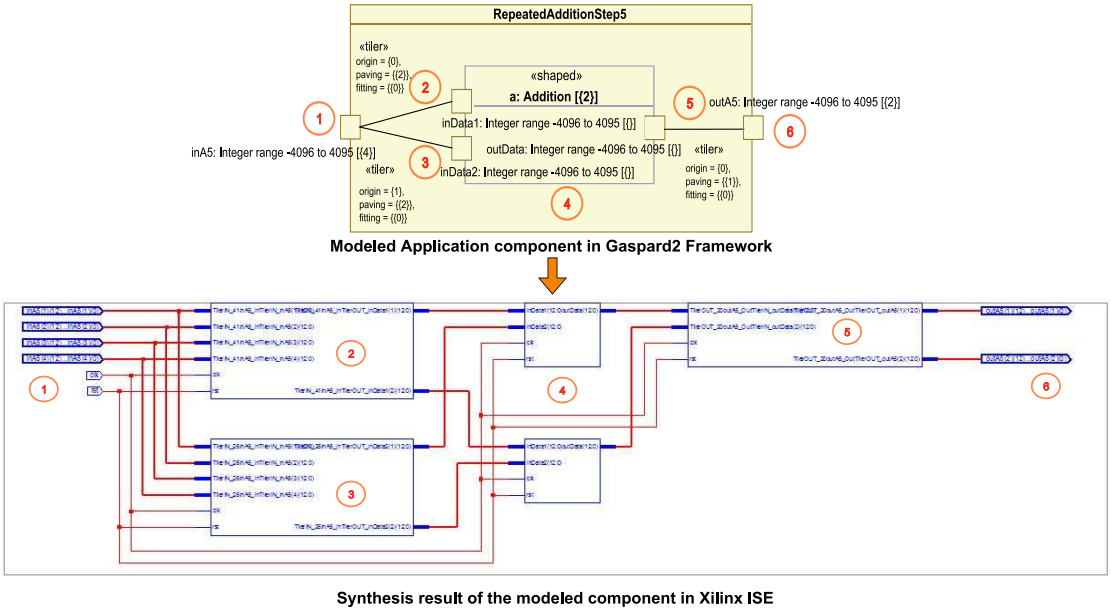


Figure 8.17: Synthesis result of the modeled application component

By means of the ADO pre-computations described before, the tiler components have simple *subconnectors*, which in their VHDL translated form, carry out mapping of the input data to output data. In the illustrated example, both input and output data are mono dimensional in nature[7], however the tiler components can also treat multidimensional data. This claim can be verified in the next chapter.

- Code for **TilerIN_41inA5:** represented as box number 2 in the synthesis result:

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
use IEEE.numeric_bit.all;
use IEEE.numeric_std.all;
use IEEE.STD_LOGIC_ARITH.ALL;

library imran;
use imran.userlibrary.all;

ENTITY TilerIN_41inA5 IS
PORT
(
clk : IN STD_LOGIC;
rst : IN STD_LOGIC;
TilerIN_41inA5_InTilerIN_inA5 : IN TABLE_TYPE_4_Integerrange4096to4095;
TilerIN_41inA5_InTilerOUT_inData1 : OUT TABLE_TYPE_2_Integerrange4096to4095);
END TilerIN_41inA5;

ARCHITECTURE archiTilerIN_41inA5 OF TilerIN_41inA5 IS

BEGIN

TilerIN_41inA5_InTilerOUT_inData1(1) <= TilerIN_41inA5_InTilerIN_inA5(1);
TilerIN_41inA5_InTilerOUT_inData1(2) <= TilerIN_41inA5_InTilerIN_inA5(3);
```

[7]The types TABLE_TYPE_4_INTEGERRANGE4096TO4095 and TABLE_TYPE_2_INTEGERRANGE4096TO4095 are generated separately in a library; and has been defined in a manner to conserve the multidimensionality of the port if it is present

```
END archiTilerIN_41inA5;
```

- Code for **TilerIN_26inA5:** represented as box number 3 in the synthesis result:

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
use IEEE.numeric_bit.all;
use IEEE.numeric_std.all;
use IEEE.STD_LOGIC_ARITH.ALL;

library imran;
use imran.userlibrary.all;

ENTITY TilerIN_26inA5 IS
PORT
(
clk : IN STD_LOGIC;
rst : IN STD_LOGIC;
TilerIN_26inA5_InTilerIN_inA5 : IN TABLE_TYPE_4_Integerrange4096to4095;
TilerIN_26inA5_InTilerOUT_inData2 : OUT TABLE_TYPE_2_Integerrange4096to4095);
END TilerIN_26inA5;

ARCHITECTURE archiTilerIN_26inA5 OF TilerIN_26inA5 IS

BEGIN

TilerIN_26inA5_InTilerOUT_inData2(1) <= TilerIN_26inA5_InTilerIN_inA5(2);
TilerIN_26inA5_InTilerOUT_inData2(2) <= TilerIN_26inA5_InTilerIN_inA5(4);

END archiTilerIN_26inA5;
```

8.4 Conclusions

This chapter presented our contributions related to the model-to-model/text transformations present in our design flow. The concepts introduced in the previous two chapters are subsequently transformed by the two model-to-model transformations: *UML2MARTE* and *MARTE2RTL*. We first provided a general overview of the two transformations, followed by the choice of utilizing a standard model transformation language to carry out their implementations. Afterwards, some transformation examples have been provided that are present in our design flow.

Once the model-to-model transformations have been executed, the resulting RTL model provides an accurate estimation of the details related to RTL. Similarly, the control concepts are enriched enough for eventual code generation. Using the model-to-text transformation described in our design flow, we generated the code for different implementations of a dynamically reconfigurable hardware accelerator, as well as the code for reconfiguration management. Finally, an example related to the synthesis of a modeled application component is provided in the chapter. We now move onto the validation of our design methodology, by providing a case study related to a complex DIP Gaspard2 application.

Chapter 9

Case study

Vehicle based anti-collision radar detection systems are becoming increasingly popular in automotive industry as well as in research. These devices provide additional safety by anticipating collisions and subsequent accidents; and can become mandatory aboard vehicles in the years to come. The principle of such systems is to avoid collisions between the equipped vehicle and the one in front, or other kind of obstacles (pedestrians, animals, etc.). The algorithms forming the basis of these complex systems require large amounts of regular repetitive computations. This computational necessity requires the execution of these algorithms in parallel hardware circuits, such as hardware accelerators [30]. We first provide a general overview of these systems, followed by the modeling of their key components and eventual code generation. Finally the chapter provides implementation details for integrating aspects of dynamic reconfiguration in these systems.

9.1 Anti-collision radar detection system

The anti-collision radar detection system which has been studied during the course of this dissertation is illustrated in Figure 9.1. The radar consists of two antennas; and emits a signal modulated with a *Pseudo Random Binary Sequence*[1] (PRBS), resulting in formation of a reference code [200]. The PRBS has interesting correlation as well intercorrelation characteristics [72]. When the emitted wave encounters an obstacle, it is reflected back in the direction of the vehicle; thus

[1] http://en.wikipedia.org/wiki/Pseudorandom_binary_sequence

creating an echo captured by means of the second antenna. The goal is to compare the received wave with the emitted one, in order to find some similarities. The echo is converted into a signal containing information related to the distance of the detected obstacle. Unfortunately, this information cannot be directly interpreted due to the presence of time delays and noise in the incoming signal. The PRBS present in the received signal is recognized by means of a delay estimation correlation module (DECM) present in the embedded system; and determines the time of flight (distance between the car and the object). Periodic computation of this distance permits estimation of the relative speed of the car and the object.

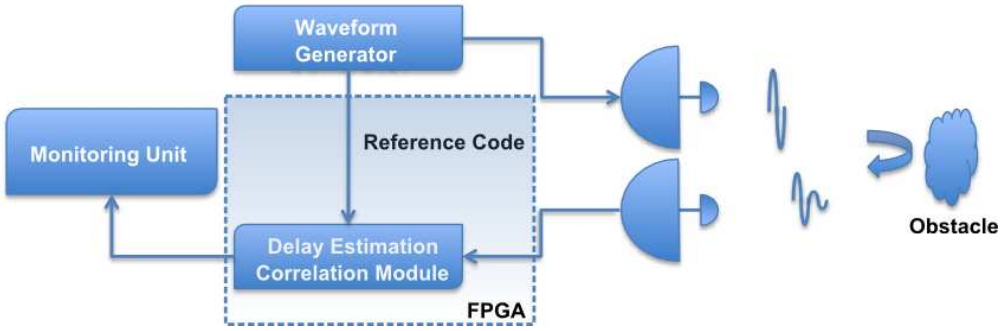


Figure 9.1: Block diagram of the anti-collision radar detection system

Additionally, it is not mandatory to exploit all the precision of the returned signal, since the information contained in the least significant bits is embedded with noise. In [72], the authors recommend to use only 4 bits of the incoming signal, because it is a good trade-off between precision and the used resources.

For the radar system, the DECM can be implemented on an FPGA, as these reconfigurable SoCs permit execution of the detection algorithm, for determining the necessary information present in the incoming signal. This information corresponds to the PRBS utilized in the emission of the wave. The role of the detection algorithm is to highlight the similarities between the reference code and the received signal: when the received signal corresponds with the reference code in the emitted wave, a peak is observed in the results, indicating the presence of an obstacle. The inverse case means that the received signal contains only a noised signal; little or no information is present related to the reference code and therefore, objects are not effectively detected. Hence detection of an object can be centred on one critical task: executing a correlation on the emitted and received wave.

Figure 9.2 shows the result of a simulated correlation measurement in MATLAB[2]. The result of a correlation between the reference code of a 127 length PRBS and the simulated received signal (integrated with time delays and noise), results in a peak. The position of the peak corresponds to the delay that we have introduced in the simulation. As the radar emits and receives a signal continuously in a temporal dimension, the correlation step is also repeated unceasingly, resulting in peaks at different time intervals. This repetition permits to determine the relative speed of the object in question. In the figure, we illustrate the results of two correlations. A peak in the correlation result indicates successful detection of an obstacle, whose distance d to the radar is given by:

$$d = c\alpha/2 \tag{9.1}$$

Where c is the speed of the propagated signal (equal to 3.10^8 meters/seconds, corresponding to the speed of light); and α is the time delay.

In this section, we have presented the structure of the anti-collision radar detection system. The DECM module is the key element of this radar detection system, and the correlation computation is extremely time consuming especially for longer PRBSs. Our case study is mainly concerned with this functionality, and we now present the correlation detection algorithm for this key component.

[2] http://www.mathworks.com/products/matlab/

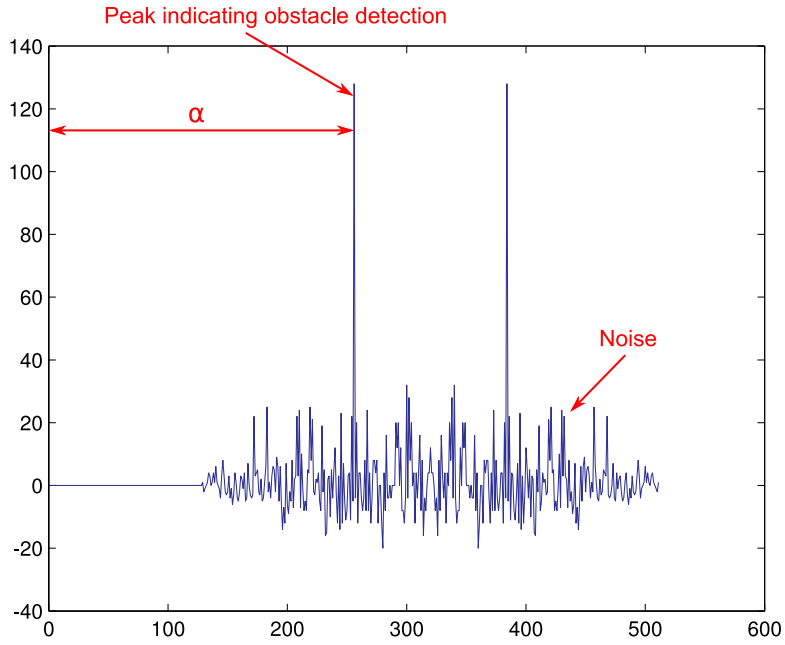


Figure 9.2: MATLAB result of the correlation between simulated emitted and received waves

9.1.1 Delay estimation correlation module

Correlation algorithms are among the type of digital processing largely employed in DSP (digital signal processing) based systems. They offer a large applicability range such as linear phase and stability. A large number of correlation algorithms exist in literature, such as presented in [30, 250]. The overall goal is to decrease the algorithm's signal to noise ratio (SNR), effectively increasing its maximum detection range for better anticipation of a possible collision.

A correlation algorithm normally takes some input data values and computes an output which is then multiplied by a set of coefficients. Afterwards, the result of this multiplication is added together to produce the final result. While a software implementation can be utilized for implementing this functionality, the correlation functionality will be sequentially executed, as illustrated in [30]. Where as a hardware implementation allows the correlation functions to be executed in parallel, increasing the processing speed. However the implementation may not be flexible for minute changes, thus a reconfigurable DECM is an ideal solution, as it offers the flexibility of a software implementation while retaining the capability to construct customized high performance computing circuits.

9.1.1.1 Correlation algorithm

The mathematical expression of a classical correlation algorithm is described as:

$$C_{cy}(j) = \frac{1}{N} \sum_{i=0}^{N-1} c(i) \cdot y(i+j) \tag{9.2}$$

Where $c(i)$ represents the reference code for creating the emitted wave, $y(i+j)$ the received signal (shifted in time) and N is the length of the referenced code. In this dissertation, we propose to study a case where our radar uses a pseudorandom binary sequence of length of 127 chips, for comparing the generated results with the MATLAB simulation results. In order to produce an effective computation result, the algorithm requires 64 multiplications between the 127 elements of the reference code and the last 127 received samples. The result of this multiplication produces 64 data elements. The overall sum of these 64 data elements produces the final result. This result can be sent as input to the monitoring unit of the anti-collision radar detection system. As we focus on the correlation of a signal with a reference code consisting

of 127 elements; and the standardization $\frac{1}{127}$ does not improves the detection quality itself, the above mentioned algorithm can be modified into the following expression:

$$C_{cy}(j) = \sum_{i=0}^{126} c(i) \cdot y(i+j) \tag{9.3}$$

This correlation algorithm can also be termed as a *Finite Impulse Response* (FIR) filter, and has been studied extensively in literature [247]. Similarly, dynamically reconfigurable filters have been proposed in [50, 180]. Our proposed case study can be thus compared to some of the existing hand tuned implementations, in order to compare the normal design time required for such an application functionality, relative to a model driven one. We have initially presented the correlation algorithm which is the basis of the detection operation in a radar system, we now move onto the UML modeling of this functionality in the following section.

9.2 MARTE based modeling of the DECM

This section presents the UML modeling of the DECM application functionality with the MARTE profile. Initially, details related to the DECM application functionality are presented, followed by its subsequent deployment and mode automata construction phases.

9.2.1 Top level of the DECM

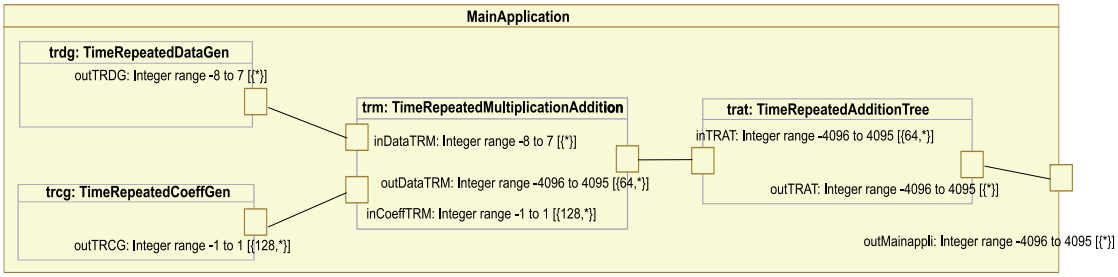


Figure 9.3: The top level view of the DECM

Figure 9.3 represents the top level of our modeled DECM module. The component instance `trm` of the `TimeRepeatedMultiplicationAddition` component determines the global multiplications while instance `trat` of component `TimeRepeatedAdditionTree` determines the overall sum. The `TimeRepeatedMultiplicationAddition` component itself carries out a partial sum between received elements of the reference code and the received signal at each rise of the clock, the output being sent to the `TimeRepeatedAdditionTree` component to execute the global addition operation. The instance `trdg` of component `TimeRepeatedDataGen` produces the data values for the generated incoming signal while the instance `trcg` of component `TimeRepeatedCoeffGen` produces the reference code.

For the generated signal, as the port `out_TRDG` of `trdg` instance has an infinite data flow (produces data at each tick of the clock); it has a corresponding shape specification value set to {*}. As we are only required to retain the 4 MSB (most significant bits) of the input signal, a type INTEGER RANGE -8 TO 7 is utilized for this signal and is associated with the corresponding port. The reference code is initially composed of 127 elements, and in order to standardize this input port with a power of 2; a 128*th* element is added in the reference code. The value of this element is neutral and does not affect the final computation result.

The reference code in the DECM can have different values: from a range of -1 to 1 where 0 allows encoding of the added element. Therefore, the type of the port `out_TRCG` of the component instance `trcg` is set to INTEGER RANGE -1 TO 1 with a shape of $\{128, *\}$. The choice to model the reference code in the form of a temporal data stream (on the basis of $\{*\}$ in its dimension) permits modification of the code during execution of the algorithm. Nevertheless, since

we only have assigned a single IP to the elementary component responsible for generating the reference code, this functionality has not been treated in the case study. However, it is possible to assign IPs signifying different reference code values or lengths, resulting in an increased number of configurations and the eventual switch. Finally, Figure 9.4 shows the hierarchical structure of `TimeRepeatedDataGen` and `TimeRepeatedCoeffGen` components related to reference code and the generated signal respectively. Each of these components contain their respective elementary components.

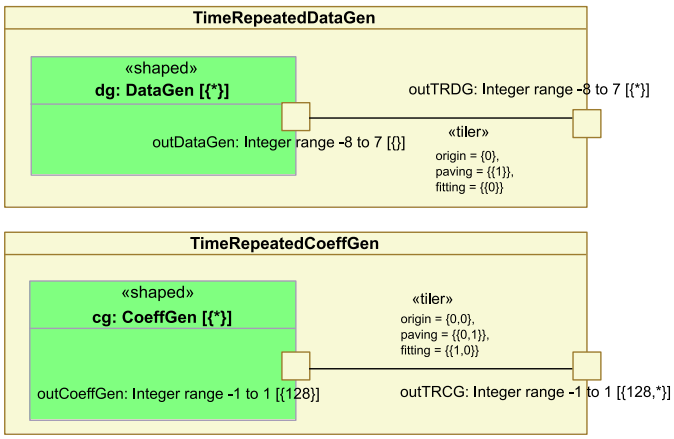


Figure 9.4: The `TimeRepeatedDataGen` and `TimeRepeatedCoeffGen` components

The output `out_TRAT` port of the instance `trat` also indicates an infinite data flow as illustrated by its dimension {*}. However, the algorithm permits us to specify that the maximum value of the output (type of the output port of `trat`) will be between −4096 and 4095; hence the associated primitive type is set as INTEGER RANGE -4096 TO 4095. In order to standardize the system output, another component could be created between `trat` and `trdc` to convert the output into an INTEGER. This step has not been taken in this case study. The generated output is then sent out to the other modules present in the radar system.

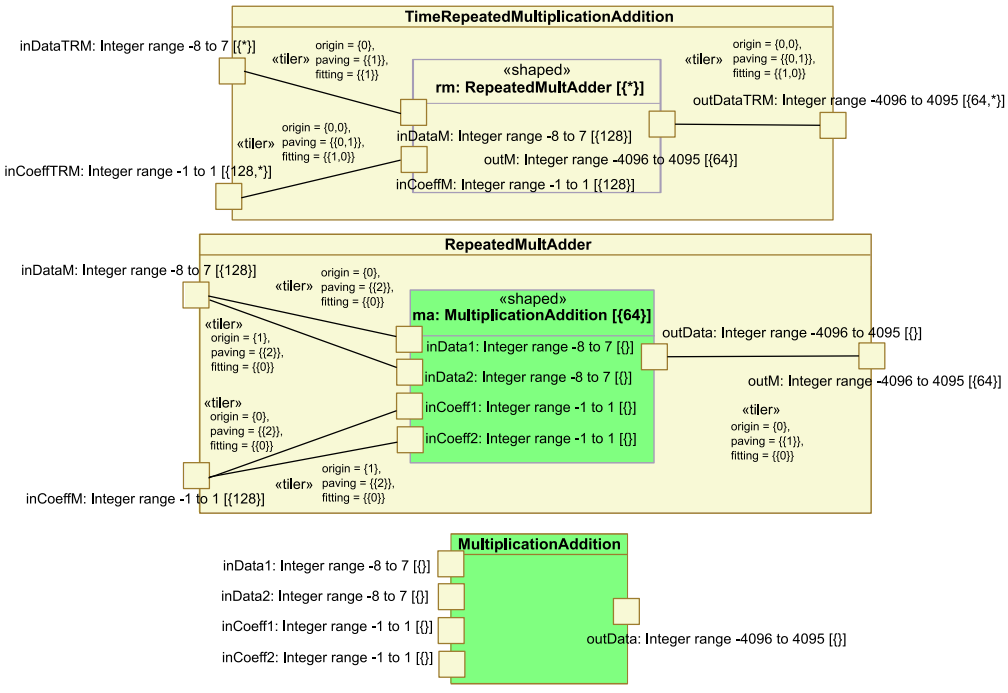


Figure 9.5: Modeling of the Multiplication stage

9.2.2 Modeling of the Multiplication step

The modeling of the global multiplication step is shown in Figure 9.5 by means of two components. This step is equivalent to the modeled application illustrated earlier in Figure 6.28. The component `TimeRepeatedMultiplicationAddition` expresses the repetition in time while `RepeatedMultAdder` expresses the repetition in space. At the level of `TimeRepeatedMultiplicationAddition`, the port `inDataM` of the instance `rm` consumes a pattern of 128 data elements by means of a sliding window data dependency in time of length 127. This sliding window is expressed via the `Tiler` connector connected to ports `inDataTRM` and `inDataM`; and expresses the data dependency between the two ports. Descending to an hierarchical level, the component `RepeatedMultAdder` realizes 64 multiplications between the data and coefficients on the ports `inDataM` and `inCoeffM`, by means of the repeated instantiations of the elementary component `MultiplicationAddition`.

9.2.3 Modeling of the Addition step

Figure 9.6 represents the component realizing the overall addition of 128 data elements. Similar to the mechanism illustrated previously, the component `TimeRepeatedAdditionTree` expresses the repetition in time while `AdditionTree` expresses the repetition in space. For the `AdditionTree`, its input port `inAdditionTree` has a dimension equal to {128} while the shape of output port `outAdditionTree` has a value set to {}, indicating a pattern consisting of one data element.

In the `AdditionTree`, The addition computation has been decomposed in a tree, with each stage of this tree carrying out partial additions. While it is also possible to use a systolic topology for carrying out the additions, in [30], the authors have illustrated the benefits of using a tree topology. Here, the dimensions of the ports between each stage in the task pipeline reduce by a factor of 2 ($128 \rightarrow 64 \ldots 2 \rightarrow 1$). Figure 9.7 represents the second last stage, which realizes a partial addition of four elements on an input port and produces two elements on its output port. The computation task `Addition` is repeated two times and is elementary in nature. This component has been equally presented earlier in Figure 8.6.

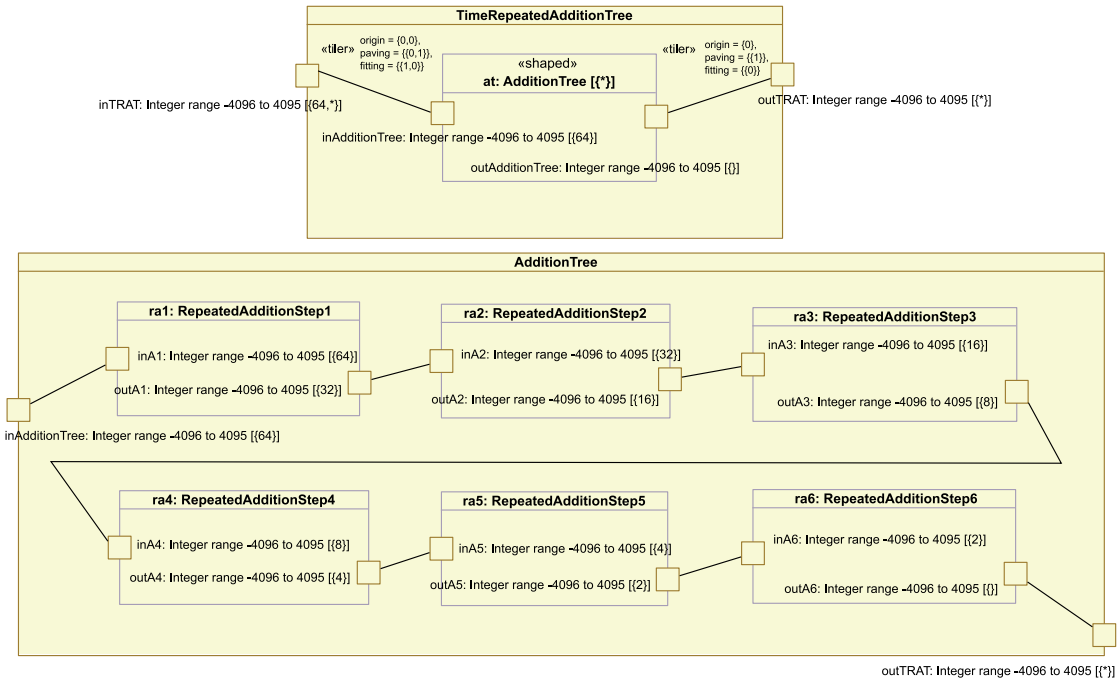


Figure 9.6: The Addition tree component

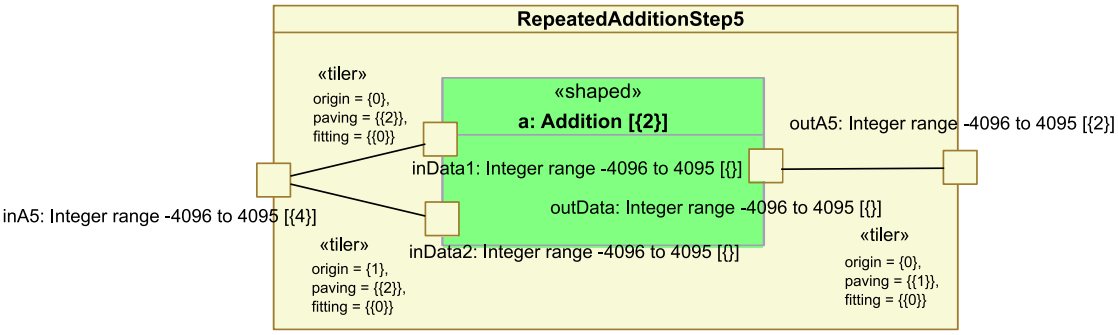


Figure 9.7: An addition step in the Addition tree

9.2.4 Deployment

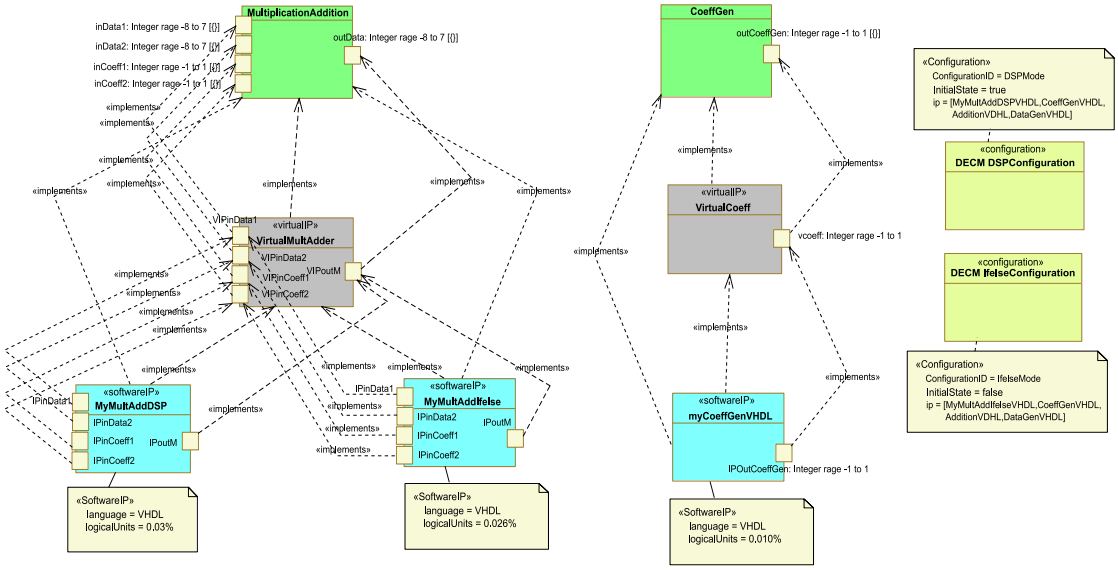


Figure 9.8: deployment of the elementary components of the DECM

Once the application is modeled, we need to carry out the deployment phase. For this application, we find a total of four elementary components (ECs): namely `DataGen`, `CoeffGen`, `MultiplicationAddition` and `Addition`. These elementary components are the basic building blocks of the DECM functionality. For the `MultiplicationAddition`, we have developed two different IPs: `MyMultAddDSP` and `MyMultAddIfelse`, respectively in the GaspardLIB. All the other elementary components each have only a single IP available for final implementation. For `MultiplicationAddition`, both IPs express the same functionality, but each is expressed differently. The first IP is written using a DSP like expression, while the other utilizes the if-then-else construct.

While this difference may seem minimal related to the overall application functionality, during FPGA implementation, each IP related to the `MultiplicationAddition` consumes a different amount of available FPGA resources. For example, during RTL synthesis, the first implementation requires multipliers and adders, while an if-else requires multiplexers. The multiple repetitions (64 in this case) of the elementary component attached to the selected IP in a configuration consumes a significant portion of the FPGA resources.

Figure 9.8 shows an extract of the deployment phase related to two of the elementary components: `MultiplicationAddition` and `CoeffGen`. For the first EC, both the available IPs are deployed, each having related properties for helping the designer in a DSE strategy. As the second EC has only one related IP, it is deployed accordingly. All the other ECs are deployed

in a similar manner and are not illustrated in the figure. Subsequently, we create two global configurations related to the DECM functionality, which are treated as its various implementations. The configurations `DECM DSPConfiguration` and `DECM IfelseConfiguration` each contains a list of associated IPs. For the case study, the `DECM DSPConfiguration` has been set as the initial configuration; and its configuration ID signifies the mode value utilized in the subsequent mode automata model. Similarly, the second configuration also has its relevant configuration ID.

9.2.5 Modeling of mode automata

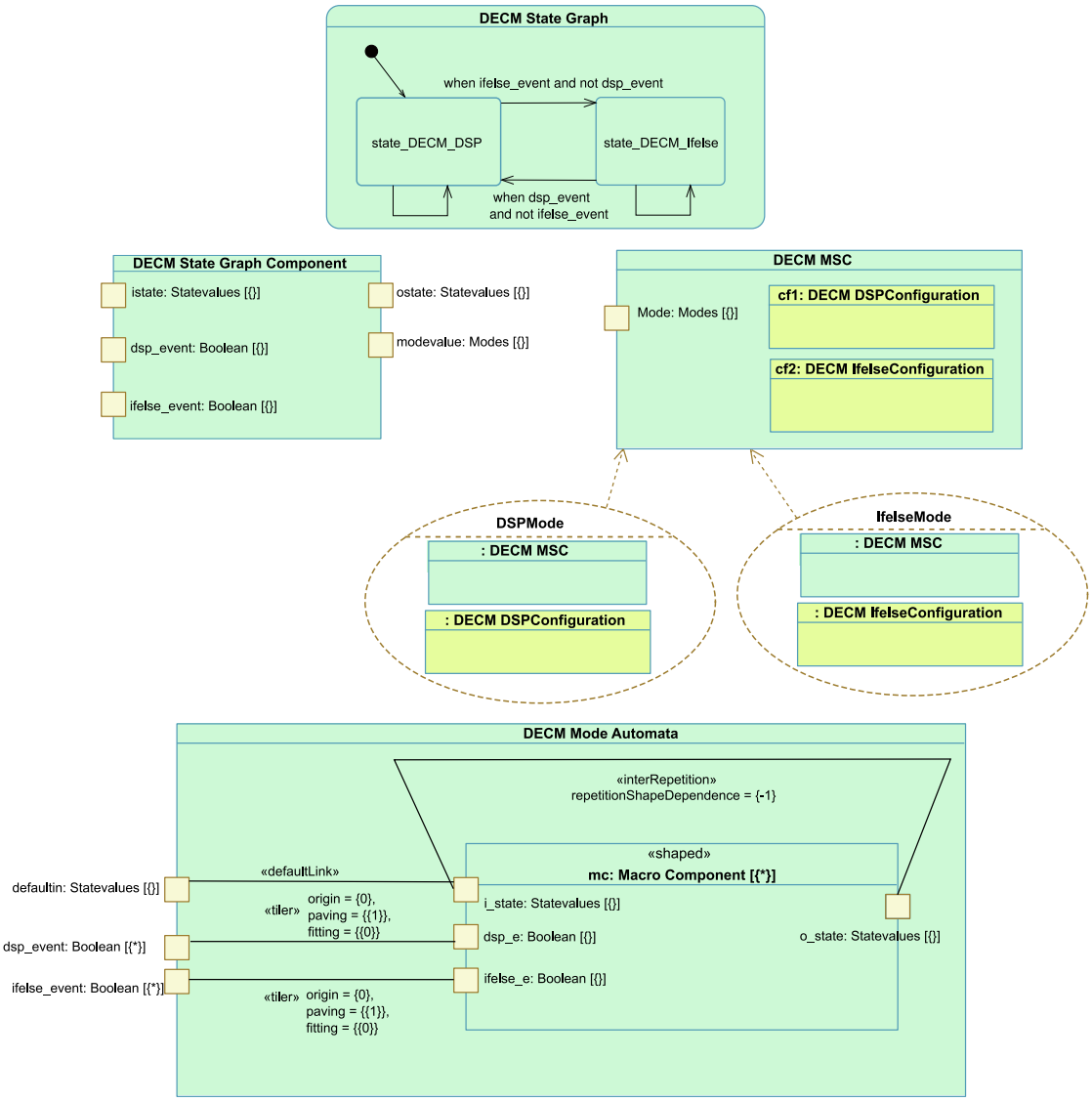


Figure 9.9: mode automata concepts for the DECM

Once the deployment phase is carried out and all the elementary components are deployed, the modeling of the mode automata related to the DECM is initiated; with the mode automata serving to switch between the different DECM configurations.

Figure 9.9 shows the various concepts related to the construction of the mode automata. This modeling approach is similar to the examples illustrated earlier in section 6.5.3, thus redundant explanatory information is unnecessary. The illustration omits some concepts such as the creation of a macro component. Here the `DECM State Graph` contains two states:

`state_DECM_DSP` and `state_DECM_Ifelse`; corresponding to the respective configurations modeled previously. This state graph determines the behavior of `DECM State Graph Component` serving as a control component. Its counterpart, the controlled `DECM MSC` component contains several collaborations, each signifying the internal behavior of this mode switch component on the basis of their interior parts and the mode value present on the port of the mode switch component. The combination of the control and controlled component forms the basis of a macro component for representing a single transition in the mode automata. For continuous transitions, the macro is placed in a repetition context task: the `DECM Mode Automata` component along with respective tilers, interrepetition and defaultLink connectors.

9.3 Code Generation of hardware accelerator and controller

As observed from the last section, in the modeling of the DECM functionality, the returned signal does not come from an external source, but is generated inside the application itself. This is not due to the limitation of the modeling approach, but is due to the reasoning that a complex data management mechanism is needed coupled with an intelligent controller; for managing the incoming/outgoing data to/from the dynamically reconfigurable FPGA region during a configuration switch. This mechanism is not generic is nature and is normally tailored for a particular application, as compared to the range of application domains targeted by Gaspard2. As currently our framework does not support a high level modeling approach for resolving these issues, both the reference code and the data signals are generated in the application functionality by means of elementary components. In case of a configuration switch, the executing data is discarded and a computation related to the new configuration begins its execution. Similarly, the output of the application during a reconfiguration switch may produce unpredictable results, leading to an unsafe state. A solution to this problem is presented later on in the chapter.

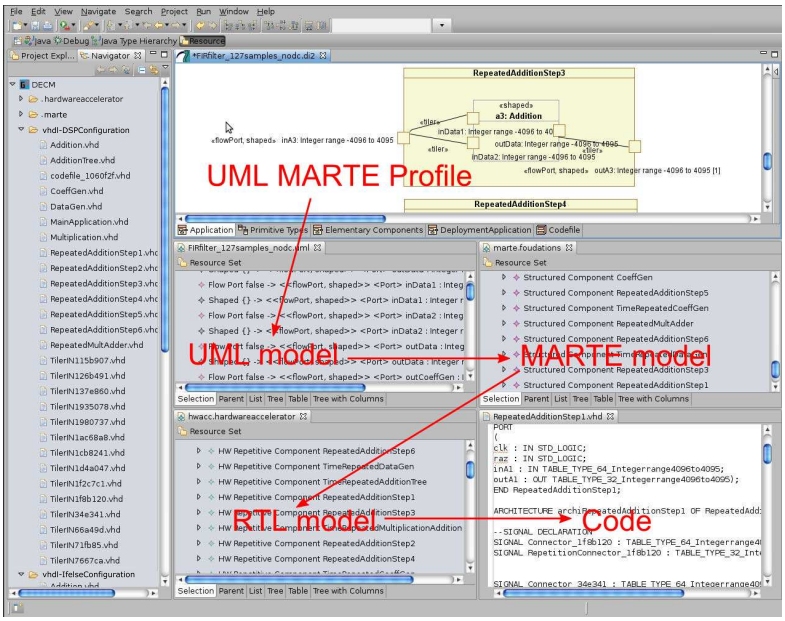


Figure 9.10: The transformation flow related to our design flow

As mentioned previously, one of the goals of our design flow is the creation of a dynamically reconfigurable hardware accelerator with several configurations, that can be swapped dynamically. The initial UML model is transformed by the various model transformations present in our design methodology, for generating the different implementations related to the hardware functionality. The control model is equivalently converted into a state machine for eventual utilization by the reconfiguration controller. Figure 9.10 represents a screenshot of the Gaspard2 environment and the different models present in our design flow. The UML model corresponds to the modeled control integrated deployed application functionality; and is directly generated

from the UML diagram with integrated MARTE profile, by UML modeling tools such as Papyrus. This model is taken as input by the model-to-model transformations presented in the previous chapter, for creation of several intermediate models, such as the RTL model (corresponding to the RTL metamodel) introduced in chapter 8. Finally, one of the steps of our design flow consists of the generation of the code related to the hardware accelerator and the configuration controller, by means of a model-to-text transformation.

We now present some of the simulation results related specifically to the generated hardware accelerator for validating the functionality associated with its different implementations.

9.3.1 Simulation of hardware accelerator implementations

Figure 9.11: First peak/correlation of the DSP configuration

Figure 9.12: Second peak/correlation of the DSP configuration

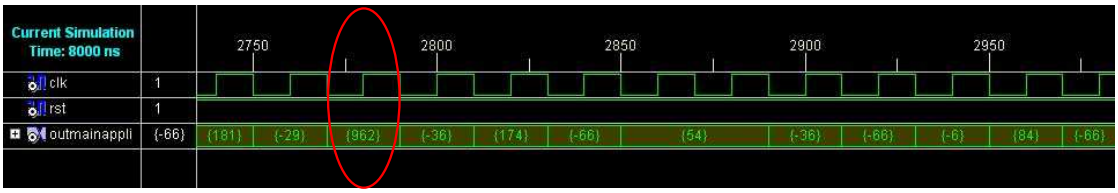


Figure 9.13: First peak/correlation of the If-then-else configuration

The verification of the modeled application and its eventual equivalent hardware execution (i.e., different implementations) is first carried out by means of simulation using the industry standard ModelSim[3] simulation tool.

Once the code for the various configurations has been generated from the model transformations, we move on to the simulation part for verification of these functionalities. Figures 9.11 and 9.12 show two peak values, related to results of the two correlations for the DSP configuration in a time window consisting of 8000 ns. This window is just sufficient enough to observe two peaks, corresponding to the MATLAB simulation result illustrated in Figure 9.2. As the simulation results are a near perfect match to the earlier results, the generated configuration is considered valid. Similarly, Figures 9.13 and 9.14 represent the simulation results for the second configuration. The simulation results verify the hardware execution related to the different implementations of the high level application functionality. We now move onto the implementation phase of a dynamically reconfigurable DECM.

[3] http://www.model.com/

Figure 9.14: Second peak/correlation of the If-then-else configuration

9.4 Implementing a partial dynamically reconfigurable DECM

In the previous sections, we have presented the initial details related to the application selected for this dissertation, along with its modeling at the MARTE profile level. Afterwards via the design flow presented during the course of this thesis, the integrated model-to-model/text transformations generate the source code from the high abstraction level input models. Once the source code has been generated, we move onto implementing a partial dynamically reconfigurable SoC [207]. This section deals with the implementation details and provides the validation of our design methodology.

9.4.1 Overview

Until now during the course of this dissertation, we have presented an abstract representation of the partial dynamically reconfigurable system. We now illustrate the in depth details related to this system. Firstly, the details related to the PDR based SoC are given, afterwards we present the steps taken for integrating our contributions to the system, followed by the results and the deduced conclusions.

9.4.2 Chosen architecture for implementing Partial Dynamic Reconfiguration

Selection criteria. We first investigated the architectural choices available for implementing PDR in Xilinx FPGAs. In Figure 9.15 we present the global structure of our reconfigurable architecture, implemented on the Xilinx Virtex-II Pro XC2VP30 FPGA on a XUP Board[4]. This particular type of structure is popular in the domain related to dynamically reconfigurable FPGAs, and various variants have been built from this classical structure, such as presented in [44, 4]. The choice of selecting the classical structure was a) to compare our system with other existing PDR based systems in literature, and b) to provide the basic template for a model driven dynamically reconfigurable system that can be optimized by the domain experts, in order to generate their customized versions.

Choice of a controller. In our selected system structure, we make use of the embedded hard-core PowerPCs present in the Xilinx Virtex-II Pro series FPGAs. One of the PowerPCs is selected as the reconfigurable controller and the state machine code generated from the high level control model in our design flow is executed on this processor. This code is generic enough to be implemented in either a hard or soft core microprocessor, but requires some fine tuning to be completely compatible. For example, it needs integration with code managing the ICAP primitive core. The fine tunings related to the executing code are presented later on in the chapter. Additionally, the code can be executed equally on any of the two present PowerPCs, and it is a design choice to select a specific PowerPC. In the case study, one of the PowerPC acts as the controller, while the other one is inactive[5]. For implementing self dynamic reconfiguration, it is also

[4] http://www.xilinx.com/univ/xupv2p.html

[5] The choice of selecting a specific PowerPC is an arbitrary one, as our current design flow does not offer a complete flow from high level FPGA modeling and allocation to final code generation

possible to utilize a softcore Microblaze processor as the controller. However, a softcore processor itself uses the CLB reconfigurable resources of the FPGA. As we want to leave the maximum space possible for the dynamically reconfigurable region, a hardcore controller approach seems an appropriate choice. More details related to this particular selection are highlighted in section 9.4.5.

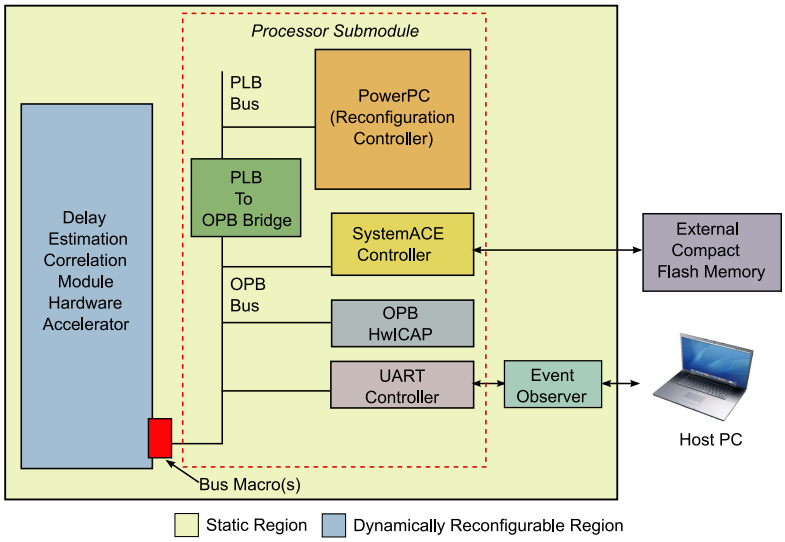


Figure 9.15: Block diagram of the architecture of our reconfigurable system

The PDR system. The PDR system can be mainly divided into two main regions. The static region and the dynamically reconfigurable one. The static region mainly consists of a processor submodule containing the reconfiguration controller and other necessary peripherals for dynamic reconfiguration. In this submodule, the reconfiguration controller connects directly to the high speed 64-bit *Processor Local Bus* (PLB) and links with the slower slave peripherals, which are themselves connected to the 32-bit *On-Chip Peripheral Bus* (OPB); via a *PLB to OPB Bridge*. The buses and the bridge are a part of the IBM Coreconnect technology [117]. The OPB bus is attached to several peripherals, such as *SystemACE controller* for accessing the partial bitstreams placed in an external onboard Compact Flash (CF) card. The ICAP core is also present in an OPB peripheral (Xilinx OPB HwIcap module) and carries out partial reconfiguration using the read-modify-write mechanism. Finally the user inputs/events are taken by the processor submodule by means of a RS232 UART controller module operating at a baud rate of 115200 (bits per second).

The processor submodule is connected to a dynamically reconfigurable hardware accelerator via bus macros. This hardware accelerator is equivalent to the hardware functionality generated from the modeled application in our design flow, and serves as the partial reconfigurable region (PRR) in the overall system. The various implementations/partial reconfigurable modules (PRMs) related to the PRR are consistent with the configurations modeled at the deployment phase. The bus macros connected to the outputs of the hardware accelerator have a special enable/disable signal, it permits the controller to disable the macros during a configuration switch to another state. Once a successful switch is carried out, the bus macros are enabled again. Thus during the switch, no output is generated from the PRR, causing the system to always remain in a safe state.

Although the accelerator can be placed with the fast PLB bus, it is an implementation choice to connect it with the OPB bus. An internal memory can also be used to store the partial bitstreams depending upon the size of the targeted configuration. Similarly, an external memory can also be employed to speed up reconfiguration times. These choices are further discussed in section 9.4.5.

Finally the processor submodule system is connected to an event observer, that is described later on in detail. The event observer receives the event values and relays them to the RS232

UART controller of the processor submodule. Users can send inputs (events), from the host PC, related to the configuration switches of the PDR system, by means of a hyperTerminal. A program running on the hyperTerminal gives a user the choice of switching between the available configurations.

When a specific input value associated to an unique state is received by the PDR system, the associated state transition is carried out. We now move onto providing the details related to the Xilinx EAPR design flow for the implementation part related to this system. For this, we first provide the integration of the generated state machine code into the reconfiguration controller.

9.4.3 State machine code for configuration switch

As detailed previously in chapter 6, the generated state machine code from our transformations is responsible for managing the context switch of the related configurations. However, this code needs to be integrated with a manually written hard macro code responsible for some RTL details, such as initializing the module responsible for carrying out the internal reconfiguration: namely the ICAP primitive module. While in chapter 8, an example of the code generated for the configuration switch was provided, detailed information was omitted; such as that related to the integration of an empty configuration, and the configuration switches. We now present a more accurate version for execution in a reconfigurable controller, leaving out the code related to the hard macro. An extract of the state machine code is illustrated below:

```
1
...
...
5     void Automata(State initialState, Event e)
{
        static State currentState= initialState;
        static State nextState = initialState;
        while (true)
10    {
                nextState=Transition(e, currentState);
                currentState=nextState;
          }
}
15
State Transition (Event e, State currentState)

{
 static State s = currentState;
20 static State theStateAfterTransition;
 switch(s)
  {
   case State_DSP: //at state DSP
   switch(event)
25  {
    case 'i': //switch to If-then-else state

    xil_printf("\r\n Performing reconfiguration for DECM If-then-else Configuration
    \n\r");
30  XIo_Out32(XPAR_OPB_DCR_SOCKET_0_DCR_BASEADDR,0x00000000);  // disable PRM
    XTime_GetTime(&tstamp1);
          XHwIcap_CF2Icap(&MyIcap, "config_if-then-else.bit"); //change configuration
    XTime_GetTime(&tstamp2);
    config_time = (unsigned int) (tstamp2-tstamp1); //clock cycles for reconfiguration
35  XIo_Out32(XPAR_OPB_DCR_SOCKET_0_DCR_BASEADDR,0x00000001);  // enable PRM
    xil_printf("\r\nConfig. Time : %5.2ld\r\n", config_time);
    ...
    ...
    theStateAfterTransition = State_Ifelse;
40  menu();
    ...
    ...
    break;

45  case 'b': //switch to blank state
```

```
    xil_printf("\r\n Performing reconfiguration for DECM Blank Configuration \n\r");
    XIo_Out32(XPAR_OPB_DCR_SOCKET_0_DCR_BASEADDR,0x00000000);  // disable PRM
    XTime_GetTime(&tstamp1);
    XHwIcap_CF2Icap(&MyIcap, "config_blank.bit"); //change configuration
    XTime_GetTime(&tstamp2);
    config_time = (unsigned int) (tstamp2-tstamp1); //clock cycles for reconfiguration
    XIo_Out32(XPAR_OPB_DCR_SOCKET_0_DCR_BASEADDR,0x00000001);  // enable PRM
    xil_printf("\r\nConfig. Time : %5.2ld\r\n", config_time);
    ...
    ...
    theStateAfterTransition = State_Blank;
    menu();
    ...
    ...
    break;

    default: //  self transition in case of a random event/all
    theStateAfterTransition = currentState;
    break;
   }
  break;

  case  State_Ifelse: //at state If-then-else
  switch(event)
   {
    case 'd': //switch to DSP state

    xil_printf("\r\n Performing reconfiguration for DECM DSP Configuration \n\r");
    XIo_Out32(XPAR_OPB_DCR_SOCKET_0_DCR_BASEADDR,0x00000000);  // disable PRM
    XTime_GetTime(&tstamp1);
          XHwIcap_CF2Icap(&MyIcap, "config_dsp.bit"); //change configuration
    XTime_GetTime(&tstamp2);
    config_time = (unsigned int) (tstamp2-tstamp1); //clock cycles
    //for reconfiguration
    XIo_Out32(XPAR_OPB_DCR_SOCKET_0_DCR_BASEADDR,0x00000001);  // enable PRM
    xil_printf("\r\nConfig. Time : %5.2ld\r\n", config_time);
    ...
    ...
    theStateAfterTransition = State_DSP;
    menu();
    ...
    ...
    break;

    case 'b': //switch to blank state

    xil_printf("\r\n Performing reconfiguration for DECM Blank Configuration \n\r");
    XIo_Out32(XPAR_OPB_DCR_SOCKET_0_DCR_BASEADDR,0x00000000);  // disable PRM
    XTime_GetTime(&tstamp1);
    XHwIcap_CF2Icap(&MyIcap, "config_blank.bit"); //change configuration
    XTime_GetTime(&tstamp2);
    config_time = (unsigned int) (tstamp2-tstamp1); //clock cycles for
    //reconfiguration
    XIo_Out32(XPAR_OPB_DCR_SOCKET_0_DCR_BASEADDR,0x00000001);  // enable PRM
    xil_printf("\r\nConfig. Time : %5.2ld\r\n", config_time);
    ...
    ...
    theStateAfterTransition = State_Blank;
    menu();
    ...
    ...
    break;

    default: //  self transition in case of a random event/all
    theStateAfterTransition = currentState;
    break;
   }
  break;


  case  State_Empty: //at blank state
```

```
   switch(event)
    {
120   case 'd': //switch to DSP state

      xil_printf("\r\n Performing reconfiguration for DECM DSP Configuration \n\r");
      XIo_Out32(XPAR_OPB_DCR_SOCKET_0_DCR_BASEADDR,0x00000000);  // disable PRM
      XTime_GetTime(&tstamp1);
125         XHwIcap_CF2Icap(&MyIcap, "config_dsp.bit"); //change configuration
      XTime_GetTime(&tstamp2);
      config_time = (unsigned int) (tstamp2-tstamp1); //clock cycles for reconfiguration
      XIo_Out32(XPAR_OPB_DCR_SOCKET_0_DCR_BASEADDR,0x00000001);  // enable PRM
      xil_printf("\r\nConfig. Time : %5.2ld\r\n", config_time);
130   ...
      ...
      theStateAfterTransition = State_DSP;
      menu();
      ...
135   ...
      break;

      case 'i': //switch to If-then-else state

140   xil_printf("\r\n Performing reconfiguration for DECM If-then-else Configuration
      \n\r");
      XIo_Out32(XPAR_OPB_DCR_SOCKET_0_DCR_BASEADDR,0x00000000);  // disable PRM
      XTime_GetTime(&tstamp1);
         XHwIcap_CF2Icap(&MyIcap, "config_if-then-else.bit"); //change configuration
145   XTime_GetTime(&tstamp2);
      config_time = (unsigned int) (tstamp2-tstamp1); //clock cycles for reconfiguration
      XIo_Out32(XPAR_OPB_DCR_SOCKET_0_DCR_BASEADDR,0x00000001);  // enable PRM
      xil_printf("\r\nConfig. Time : %5.2ld\r\n", config_time);
      ...
150   ...
      theStateAfterTransition = State_Ifelse;
      menu();
      ...
      ...
155   break;


      default: //  self transition in case of a random event/all
      theStateAfterTransition = currentState;
160   break;
     }
   break;
  }
return theStateAfterTransition;
165 }

...
...
```

The code is similar to that presented in the precedent chapter, but is enriched with details for carrying out the actual configuration switch related to the dynamically reconfigurable hardware accelerator. The model-to-text transformation responsible for generating the state machine code integrates an empty configuration in the code illustrated above, along with a respective associated event for the eventual switch. These additions are currently hard coded in the corresponding JET templates, for respecting PDR semantics. An empty configuration can be used in a scenario, when power consumption levels need to be reduced related to a reconfigurable SoC. For the construction of an automata, the interrepetition and defaultLink concepts presented in the RTL model are converted to variables, via the corresponding JET templates. Specifically, the defaultLink is converted into an initial state variable, while the interrepetition determines the current or next executing state. The *main* function of the configuration code initially loads the full bitstream which is a merge of static and DSP partial bitstreams (not shown here); and calls the *Automata* function. This function itself calls a *Transition* function for the configuration switch to different states (or self transitions), when receiving the appropriate events.

The model transformations do not generate all of the code illustrated above, but leave out certain minute details; for providing a generic template. These details are given by the designer working at the Register Transfer Level, targeting a particular FPGA series. The details are related to information such as addresses for accessing the configuration frames, the targeted FPGA, etc. Similarly, certain ICAP related library functions are available for the hardcore PowerPC processor, but are absent for the softcore Microblaze, such as the XTime timer related headers and declared functions[6], which help in determining the reconfiguration time during a configuration switch. A Microblaze processor thus needs an explicit timer module for measuring this time period. The omitted details in the generated state machine code help in producing a basic code template for execution in any target Xilinx FPGA, supporting a hard/soft core controller. Similarly, the input events related to each particular state have been given a unique value in the model transformations. These values are entered as characters by the user from the host PC, on the basis of the *menu* function presented subsequently. Finally, the state machine code is then integrated with the hard macro code related to the ICAP module, for managing the reconfiguration controller. We now present an extract of the function displayed on the host PC, it permits the user to enter some unique characters, each tied to a specific configuration. It should be mentioned that this code is also generated by the model-to-text transformation in our design flow. The choice of assigning input events to specific characters is handled by the transformation in an arbitrary manner.

```
1  void menu(void)
   {
           xil_printf("-------------------------------------------------------\r\n");
           xil_printf("      Reconfiguration test for DECM functionality \n\r");
5     xil_printf("      Press d for DECM DSP configuration \n\r");
      xil_printf("      Press i for DECM Ifelse configuration \n\r");
      xil_printf("      Press b for DECM Blank configuration \n\r");

      ..
10    ..
   }
```

9.4.4 EAPR Flow

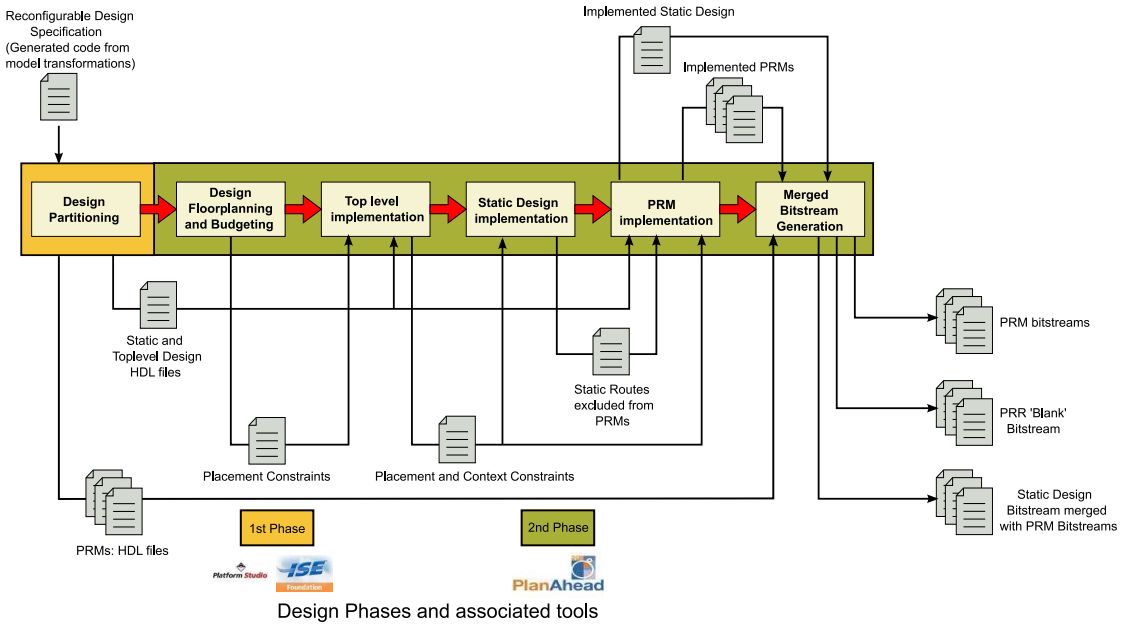


Figure 9.16: The EAPR flow used for the case study

[6] www.xilinx.com/ise/embedded/edk_libs_ref_guide.pdf

Figure 9.16 presents the overview of the strategy for conceiving partial dynamically reconfigurable FPGAs using Xilinx EAPR flow. The methodology consists of several significant steps, which we have categorized into two main phases. The EAPR methodology has been chosen as it is openly available, and can be used to construct a wide variety of PDR systems. We now provide the details related to this methodology and the associated phases, with respect to our case study. In-depth details related to the mechanism of the EAPR flow can be found in [260].

9.4.4.1 Phase 1: Planning/design partitioning and synthesis

Once the code has been generated from our model driven design flow, we move on to the initial design partition phase of our PDR system according to the EAPR flow. The processor submodule for the PDR system is initially created by means of the Xilinx Platform Studio Embedded Development Toolkit[7]. Figure 9.17 illustrates the block diagram of the subsystem created in tthe platform studio. The source code for the controller is selected to be executed on the PowerPC 405_0, with a clock frequency of 100 MHz. The second PowerPC while present in the figure, is not connected to any clock signals and is therefore disabled. A PLB Block-RAM (BRAM) interface controller permits interfacing between the PLB and a Block-RAM of size 128 KB. This size is sufficient to store the data and instructions of the executable processor code, and on-chip-memory (OCM) is not required. Using FPGA BRAMs to store the data/instructions allows the processor code and initialized variables to be written directly into the memory, when the FPGA is configured initially.

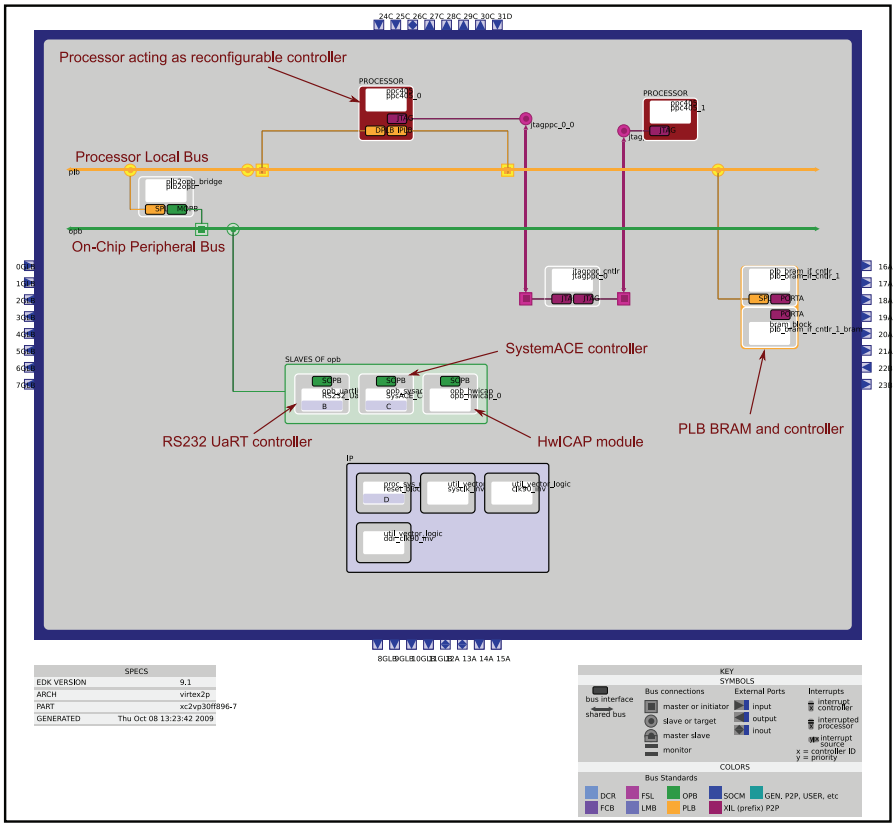


Figure 9.17: The processor subsystem created via the EDK tool

According to the EAPR flow recommendations, *Digital Clock Managers* (DCMs) are not present in our processor subsystem and are instantiated at the top level of the PDR architecture. Once the processor subsystem has been configured, a netlist is generated for the hardware

[7]For the implementation, the 9.1 version of EDK has been used, as the EDK 9.2 version supporting EAPR does not support the XUP board. Additionally, the new versions of EDK are not compatible with the EAPR tools

design, while the software portion related to the device drivers and the controller code is complied in parallel. Afterwards, we move onto integrating the generated dynamically hardware accelerator into the PDR system. The following subsection details this integration.

Integration of the hardware design into the PDR system As elaborated earlier, [143] proposed a model driven flow for creating a single static hardware accelerator, which is intended to be implemented in a target FPGA as a black box and there is no notion of heterogeneity in the final design. However, for implementing self dynamic reconfiguration, the PDR system is composed of mandatory heterogeneous communicating components (the reconfiguration controller, buses, memory controllers, ICAP core, etc.). It is hence essential that the hardware functionality generated from our design flow is successfully integrated into the PDR architecture in the form of an IP core, along with the static region. Xilinx provides the notion of an *Intellectual Property-core InterFace* (IPIF) module, that acts as an hardware bus wrapper specially designed to ease custom IP core interfacing with the IBM Coreconnect buses using IP interconnections (IPIC).

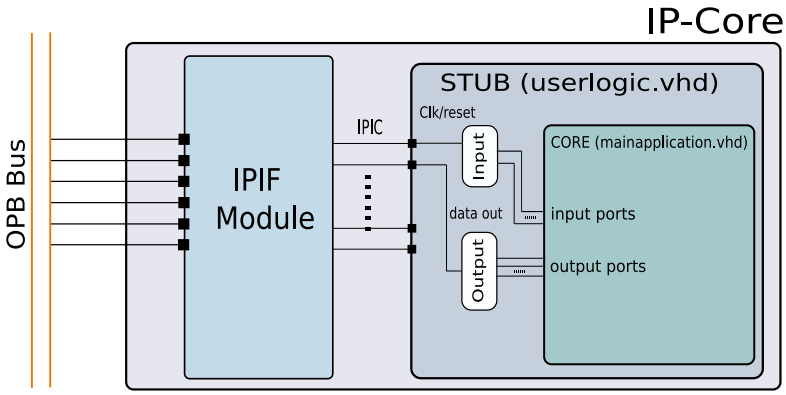


Figure 9.18: An abstract overview of the IP-Core

The advantage of the IPIF glue is that designers do not have to learn the complex protocols related to the IBM Coreconnect buses, all they need to do is choose the required functionality available in the IPIF, and know how to attach their own IP core to the IPIF. As the IPIF is a parametric softcore IP, designers can eliminate the unwanted components in the interface, selecting only the desired functionalities. The IPIF can also be used for other purposes such as connecting the OPB bus to a *Device Control Register* (DCR) bus: another bus of the Coreconnect technology [117]. There exist two versions of the IPIF: a PLB IPIF[8] for PLB attachment, and OPB IPIF[9], for OPB attachment. A custom peripheral that connects to any of the two buses must meet the principles of the OPB/PLB protocol: matching of interface signals, for example. In our design flow, the dynamically reconfigurable accelerator is connected with the OPB bus but can also be integrated with the PLB bus. An accelerator generated via the model transformations is intended to be integrated as a slave peripheral connected to the chosen bus, for communicating with the controller.

However at the modeling level, the designer does not have any in-depth knowledge of the targeted PDR architecture. Thus the wrapper creation can be viewed as one of the low level details. In order to make the accelerator compatible with the interface signals of the OPB and in turn the IPIF module, we need to make appropriate arrangements. Initially, the IPIF wizard present in the EDK tool can be used to generate the template for the custom peripheral, containing the communication logic (IPIF module) and the inner user logic module. We treat the user logic (the userlogic.vhd) file as a stub in which the final hardware design (the *core*) is instantiated. The stub allows the bus master to read/write the output and input signals of the core respectively. Moreover, as seen earlier in the section related to the modeling of the application functionality, in Gaspard2, we make use of user defined types on the input/output ports of our application. These user defined types such as INTEGER RANGE -4096 TO 4095 cannot be directly

[8] http://www.xilinx.com/products/ipcenter/plb_ipif.htm
[9] http://www.xilinx.com/products/ipcenter/OPB_IPIF_Architecture.htm

integrated into the PDR system and need to be converted into basic types such as STD LOGIC VECTOR. The stub provides an additional hierarchy for performing the conversion between the standard and user defined types. Figure 9.18 illustrates the final structure of our IP-core. The average cost of the FPGA resources consumed for the creation of the IPIF wrapper is about **45** slices, for our targeted FPGA, this amounts to about **0.3%** of total available slices. Afterwards the IPIF wizard can be invoked again to import this peripheral into the PDR system, resulting in a successful integration of the hardware accelerator.

Currently the integration of the accelerator is a manual process, however model transformations in our design flow are being extended to automatically generate a customized wrapper for our hardware functionality, bypassing the IPIF functionalities. The transformation can create the top level IP-Core VHDL file, an interface module and the stub module that itself contains the core sub-module. This approach can be seen as a complementary approach as present in [149]. Another dilemma related to this extension is elevating the low level RTL details to the modeling level. Works such as [132, 165], using IP-XACT standard at MDE and MARTE levels can be extremely beneficial in future.

Once all the configurations of the accelerator are imported as OPB peripherals (having different version names) in EDK, the project files (***peripheral_xst.prj***) related to each version of the accelerator are then modified manually as specified in the EAPR flow[10], before eventual synthesis in ISE[11] is carried out.

Creating the top level of the PDR system Once the above mentioned OPB peripherals are created, we move onto creating the top level of our PDR system. The top level VHDL file: equivalently called the top.vhd file, consists of the global logic, such as clock primitives (e.g. DCMs and BUFG global clock buffers); instantiations of I/O ports; base design, partial reconfigurable region (PRR), bus macros; and signal declarations. An initial user constraint file is also created for this top level manually. The base design instantiation corresponds to the processor subsystem, while the PRR corresponds to the dynamic hardware accelerator.

The output signals from the instantiated PRR region are sent to appropriate bus macros. These bus macros connect the output signals to the external ports of the FPGA, in order to send the result to the monitoring system of the anti-collision radar detection system. We also integrate an event observer in this top level vhdl file, whose functionality is provided subsequently. Via the model transformations, we generate a code skeleton related to the top level HDL file, which is then modified manually.

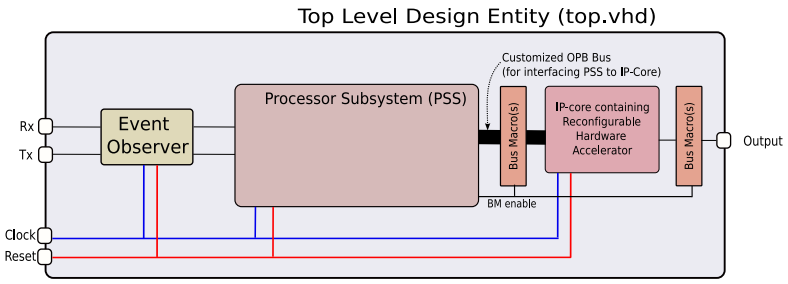


Figure 9.19: An abstract overview of the top level VHDL file. This figure is equivalent to that presented in Figure 9.15

Integrating event observer at the RTL level As elaborated previously in this dissertation, it is evident that control events are generally non deterministic in nature and depend upon the user input, while data computations in Gaspard2 are deterministic and arrive in a regular manner. Nevertheless, in Gaspard2, control events are treated in the form of control arrays similar to data arrays. This equivalence signifies the arrival of a control event at each instant of time t, as specified at the high level modeling diagrams. However, this is not always the reality and

[10] EAPR user guide: http://www.xilinx.com/support/prealounge/protected/index.htm
[11] The Xilinx ISE 9.1 version is used for the synthesis process related to the PDR system

the arrival of control events usually depends upon environmental requirements, QoS criteria or designer specifications.

In order to respect the control semantics specified earlier, the notion of an `EventObserver` is thus introduced at the RTL level in the highest hierarchical reconfigurable system entity. The internal mechanism of the observer is created due to the presence of certain model-to-text transformation rules, which first determine the number of event input ports present in the mode automata component in the RTL model. Afterwards, the rules analyze if each of these ports is connected to a respective tiler component. If the condition is set to true and a tiler component is attached to input each event port, that event is considered as a valid event[12]. Finally the observer and its interior mechanism is created by a subrule. We avoided adding the concept of an event observer at the MARTE profile level in order to distance the user from management of the incoming events, which is viewed as a lower abstraction level detail. Figure 9.19 illustrates the presence of the `EventObserver` in the partial dynamically reconfigurable system.

The *EventObserver* takes user inputs at irregular time intervals and produces events at each instant, for regular arrival of control events into the program being executed in the reconfiguration controller. This component has input and output event ports *EventIn* and *EventOut* respectively, as well as the *Clk* and *Rst* ports for clock and reset signals. The *EventIn* port is connected to the top level *UART_Rx* input port of the top level structure while the *EventOut* port is connected to the processor subsystem's *UART_Rx* input port. The algorithm related to the *EventObserver* is presented below using an informal semantic:

```
Sensitivity List (Clk, Event)
if
  CLK is TRUE and Event
  then
  EventOut = Event;
else if
  CLK = TRUE and NOT Event
  then
  EventOut = Default Value;
end if;
End Sensitivity List
```

The user input can arrive irregularly at any instant of time, where as an event value is needed at each instant of time t in order to respect Gaspard2 semantics. The *EventObserver* listens on its input port, and at each rise of clock, checks if an event is present or not. In the first case, the event is sent to the processor subsystem and in turn the reconfiguration controller which causes a successful state transition (or a self transition). In the second case, if there is no user driven input event at time t, then the *EventObserver* generates a default event e_d causing a self transition in the state machine. This value can be viewed as a special value among the set of values corresponding to the *all* expression, related to the high level modeling illustrated in Figure 6.31, the expression catches any event not specified in related transitions and causes a self transition in the state graph. If ξ is the set of all possible events and E is the set of events related to the different configurations (including the event related to an empty configuration), then the overall relation is then expressed by the equation:

$$E = \{e\_1, e\_2, e\_3\} \ , \ all = \{\xi \setminus E\} \cup \{e\_d\} \tag{9.4}$$

Here events e_1, e_2 and e_3 correspond to three unique events assigned to three different configurations respectively. A self transition in the state machine does not switches an executing configuration, while a transition to a different state causes the controller to perform a switch to the corresponding configuration. While this notion introduces regularity in the arrival of control events, it is possible that a control event and the eventual configuration switch causes a disruption in the data flow of the application implemented as a reconfigurable hardware accelerator. It is thus critical to determine the precise moment for an effective configuration switch

[12] Although MARTE NFP/OCL constraints can also be used for this verification, they are currently not fully integrated in Gaspard2, and thus not utilized in our design flow

while avoiding the failure of the required application functionality. Our works could benefit from the notion of *degree of granularity* proposed in [172] which effectively responds to the synchronization of the control/data flow. However, the disadvantages of this approach have been previously cited in Chapter 6. Additionally, it is possible that several control events arrive at a time period shorter than that required for a given configuration. In that case, the events are queued and the related configuration switches are executed sequentially.

Summary of phase 1 Once the top level of the system is created, we move onto synthesizing the various portions: namely the OPB peripherals and the top level in ISE, while the static (processor subsystem) portion has already been synthesized in the EDK. During the synthesis, care should be taken to insure that the synthesis parameter IOBUF is enabled only for the top level; and disabled for the other portions. This parameters permits integration of I/O buffers.

We now present the partial synthesis results of some of the modeled application components in our case study carried out with the Xilinx ISE on the XUP board. Figure 9.20 shows the global view of the synthesis of the modeled DECM application. Similarly, Figure 9.21 illustrates the synthesis result of the `AdditionTree` component. The modeling of this component with the UML MARTE profile illustrates simple connectors, which determine the dependencies between 64 data elements, then 32, and so on. The synthesis results illustrate the equivalent hardware components, connected by signals.

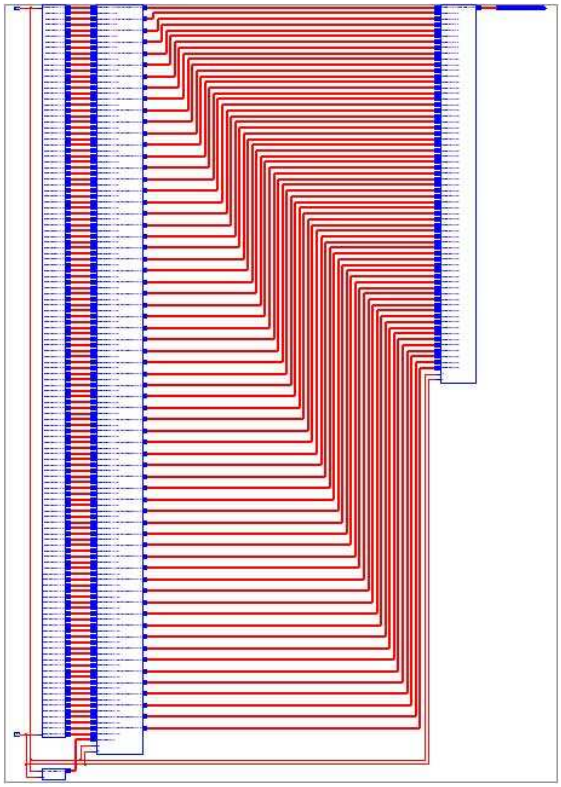


Figure 9.20: Synthesis result of the top level of the DECM

Figure 9.22 represents an extract of the synthesis results related to the `RepeatedMultAdder` component in the modeled application. The interior elementary component `MultiplicationAddition` is repeated 64 times, while the input and output tiler connectors are transformed into equivalent components. These components resolve the data dependencies between the input/output arrays and the patterns consumed by the various repetitions of the elementary component. Finally, in the DECM functionality, a sliding window is necessary in order to carry out the necessary multiplications. This data dependency is generated by means of a tiler component in the `TimeRepeatedMultiplicationAddition` component. Figure 9.23 illustrates the correspondence between this modeled concept and

its equivalent synthesis result. The code related to the transformed tiler component at the hardware accelerator is presented in appendix C.

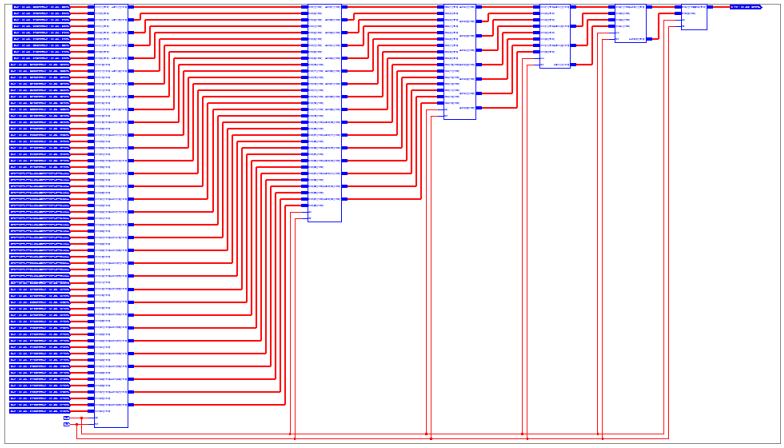


Figure 9.21: Synthesis result of the *AdditionTree* component

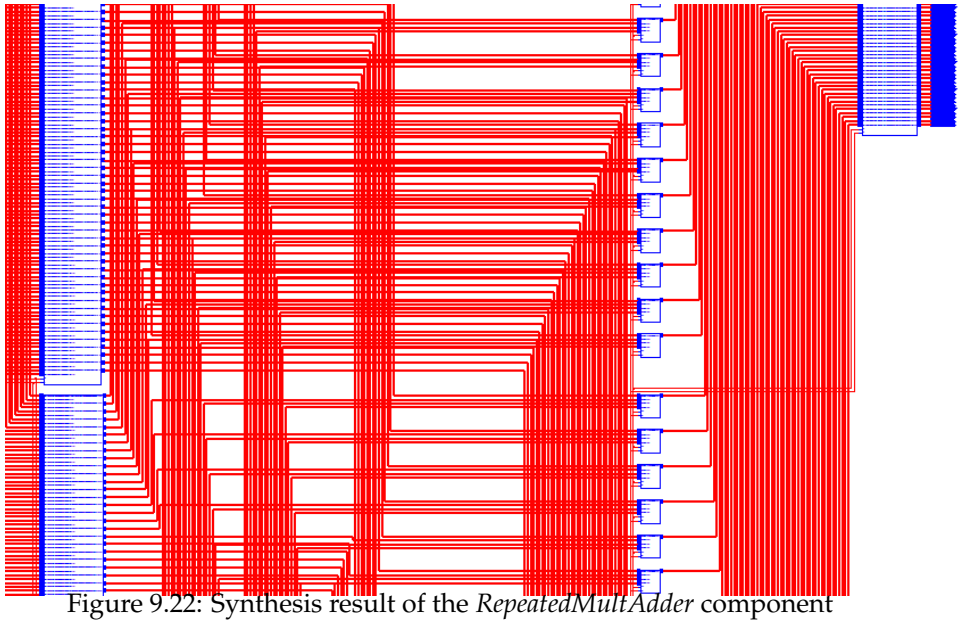


Figure 9.22: Synthesis result of the *RepeatedMultAdder* component

9.4.4.2 Phase 2: Utilization of PlanAhead tool

Once the synthesis process has been completed, we move onto the budgeting phase of the EAPR methodology. The budgeting helps in determining the size and location of PRRs, as well as the placement of bus macros between the static and dynamic regions. While the budgeting can be done manually, it is a time consuming and error prone process. To avoid these issues, we make use of the Xilinx PlanAhead tool[13] [169] that greatly automatizes the EAPR flow, and is normally deployed between synthesis and place-and-route phases.

PlanAhead permits the creation of static and partial reconfigurable regions, in the form of rectangular physical blocks (*PBlocks*), in a graphical environment. These blocks when once created, can be placed and sized on the floorplan of a target FPGA. The *PBlocks* help in physical partitioning of the design, and can be hierarchically composed. With respect to *PBlocks*, different estimations of performance are immediately available to aid the designer in optimizing the

[13] The version 10.1 of PlanAhead has been utilized in our case study

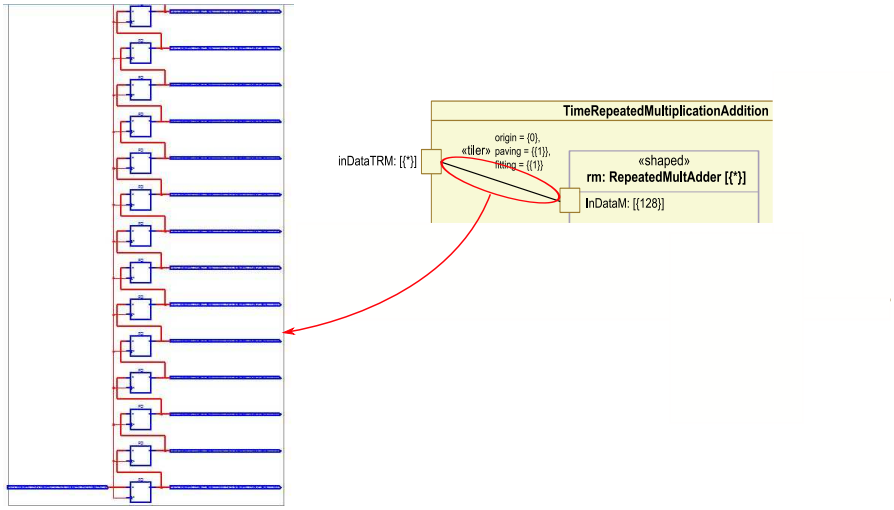


Figure 9.23: Synthesis result of the tiler expressing the sliding window data dependency

design layout, before final place-and-route. As the EAPR flow is based on a modular design methodology, the partially reconfigurable region or PRR in our design is assigned to a *PBlock*; for providing related estimates of consumed FPGA resources.

The size of a *PBlock* related to an allocated PRR can be easily modified in PlanAhead, if the associated resources such as slices, LUTs, etc.; are not sufficient enough for an effective allocation. Additionally, all the different partial reconfigurable modules (PRMs) associated to a PRR are assigned to this block. Once the optimum size and location of the PRR is determined, bus macros are placed in a CLB column inside the *PBlock* that is close to the PRR boundary, so that the macros can physically straddle the static/dynamic regions. Figure 9.24 illustrates an example of a bus macro present between the static and dynamic regions in our PDR system. Additionally, PlanAhead automatically generates the user constraint files for each module containing information related to bus macros and other associated details.

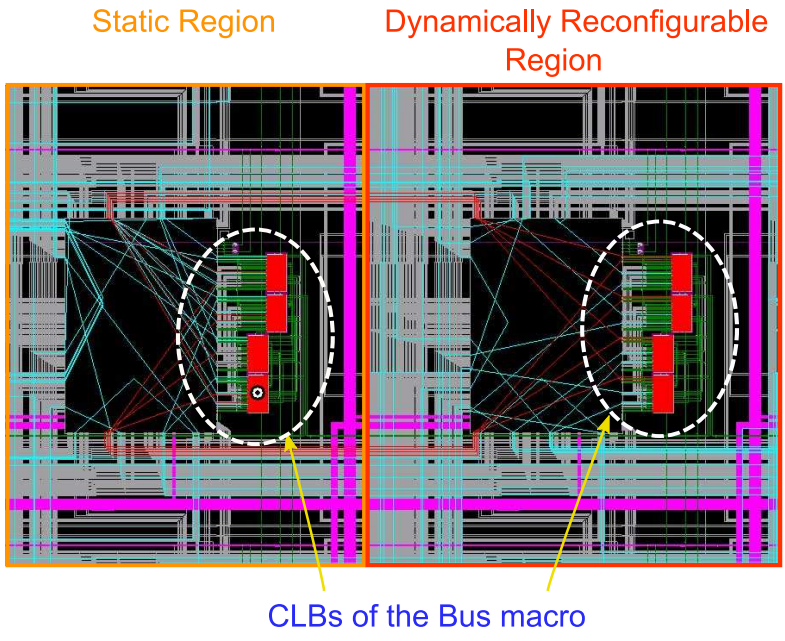


Figure 9.24: Example of a bus macro straddling the static/dynamic region boundaries

The shape and layout of the PRR depends entirely upon the designer and the environmental requirements. During the case study, the PRR containing the hardware accelerator was given a

basic rectangular shape, having a height greater than its relative width, as illustrated in Figure 9.25. This PRR was placed on the top right side on the floorplan of the targeted FPGA. The size of the PRR was optimized to be sufficiently large enough to incorporate both the related PRMs (the DSP and the If-then-else implementations of the accelerator). Table 9.1 gives the FPGA resources required for the implementation of the PRR, determined by PlanAhead.

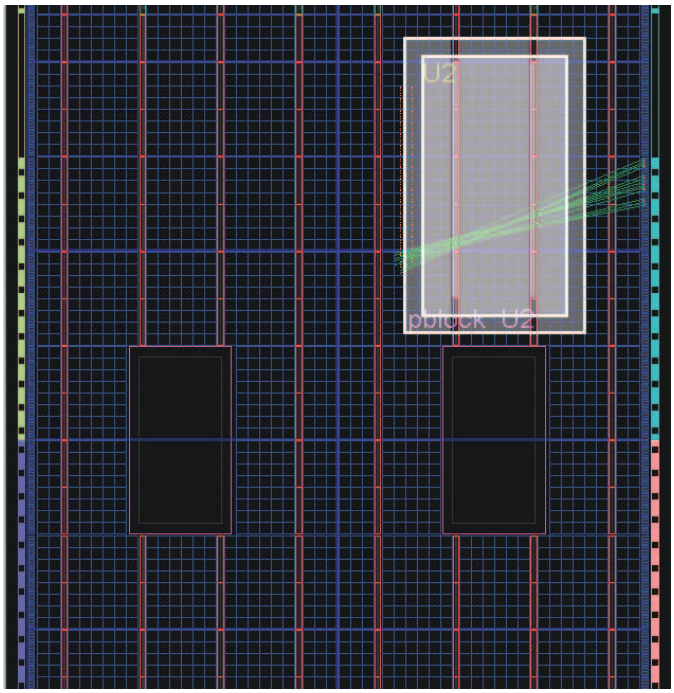


Figure 9.25: Placement of the PRR in the PlanAhead environment

Slices	1400
Slice FlipFlops	2800
LUTs	2800
18x18 Multipliers	-
BRAM16s	-
TBUF	-

Table 9.1: PBlock requirements for the accelerator (PRR) and its associated configurations (PRMs)

Once the bus macros and the top level primitives are properly placed in the floor plan, design rule checks (DRCs) are executed to determine the presence of errors or warnings. After the successful completion of DRC step, it is possible to create the corresponding static/partial bitstreams related to the PDR system.

For this, the PlanAhead tool first generates the top level context by creating a file in native Xilinx format: containing information related to clock resources, bus macro placements, static routes inside PRRs, static and PRM module placements, etc. This information is then passed onto the phases related to the static and PRM implementations. Once the static bitstream and the different PRM bitstreams (for the PRR) are created, a merge mechanism creates a complete design (a full bitstream comprising of a merge of the static and DSP configuration bitstreams).

This resulting bitstream is the bootup full bitstream for the targeted FPGA. Similarly, a *blanking* bitstream is also created for the PRR, via the EAPR tools. The *blanking* bitstream is empty in the sense that it does not contain any PRM logic, except the static routes crossing through the PRR. This bitstream can be loaded when a PRM related to the PRR is not required, resulting in reduced power consumption.

Finally, the compact Flash memory provides the storage for the partial and full bitstreams. The concatenated full bitstream is first converted into a *SystemACE* file[14], in order to be utilized by the SystemACE controller. Afterwards, when the FPGA is powered on for the first time, the initial full configuration is loaded and the program associated to the PowerPC reconfigurable controller begins executing; and is displayed on the screen of the host PC via a hyperTerminal program. The user enters some predefined inputs, each input corresponding to a particular configuration. These inputs help in providing the configuration switch related to the hardware accelerator; and the result (the time taken for a reconfiguration between two configurations) related to the switch is displayed accordingly on the host PC, verifying that an effective reconfiguration has taken place.

	DSP Configuration	If-then-else Configuration
Slices	1272/13696 (9.287%)	1186/13696 (8.659%)
Slice FlipFlops	2084/27392 (7.608%)	1944/27392 (7.096%)
LUTs	1584/27392 (5.782%)	1836/27392 (6.702%)
Reconfiguration Time (secs)	1.45	1.41

Table 9.2: Results related to the two configurations for the hardware accelerator. The percentage is in overview of the total FPGA resources. The results related to the blanking configuration have not been illustrated in the table

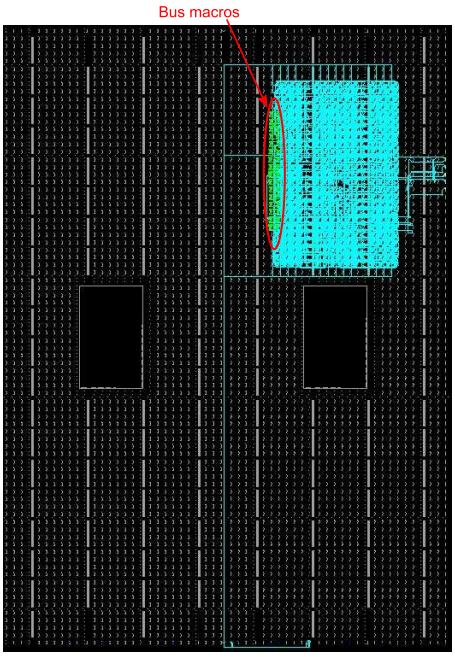


Figure 9.26: Partial bitstream related to the DSP configuration

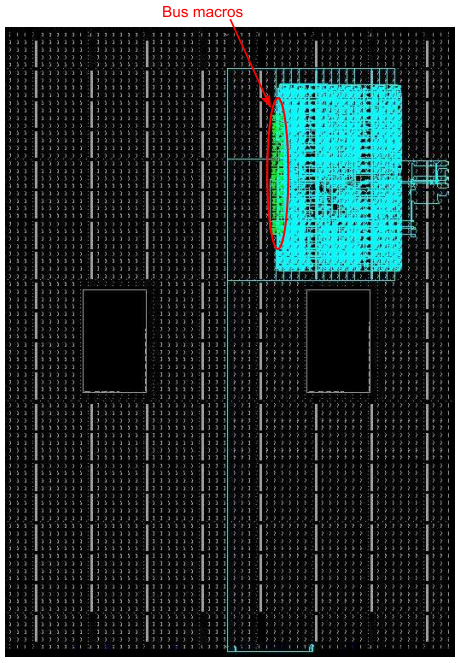


Figure 9.27: Partial bitstream related to the If-then-else configuration

Table 9.2 shows the results related to the two configurations. The first configuration consumes more FPGA resources in comparison to the second one. Additionally, the reconfiguration time to switch from the If-then-else configuration to the DSP one is higher, comparatively to the inverse case. This is because the ICAP core needs to modify several additional frames for the DSP configuration, as compared to the latter. While the reconfiguration time is extremely high for both configurations, this is due to the low bandwidth (115200 bps) of the RS232 controller and the large size of the partial bitstreams. Using an external SRAM memory can greatly decrease the reconfiguration times, similarly various other optimizations can be carried out with

[14] Details related to SystemACE file generation can be found at http://www.xilinx.com/

respect to the implementation, as illustrated subsequently in the next section. Figures 9.26 and 9.27 show the two partial bitstreams of the hardware accelerator respectively, while Figures 9.28 illustrates the static bitstream containing the processor subsystem and the event observer.

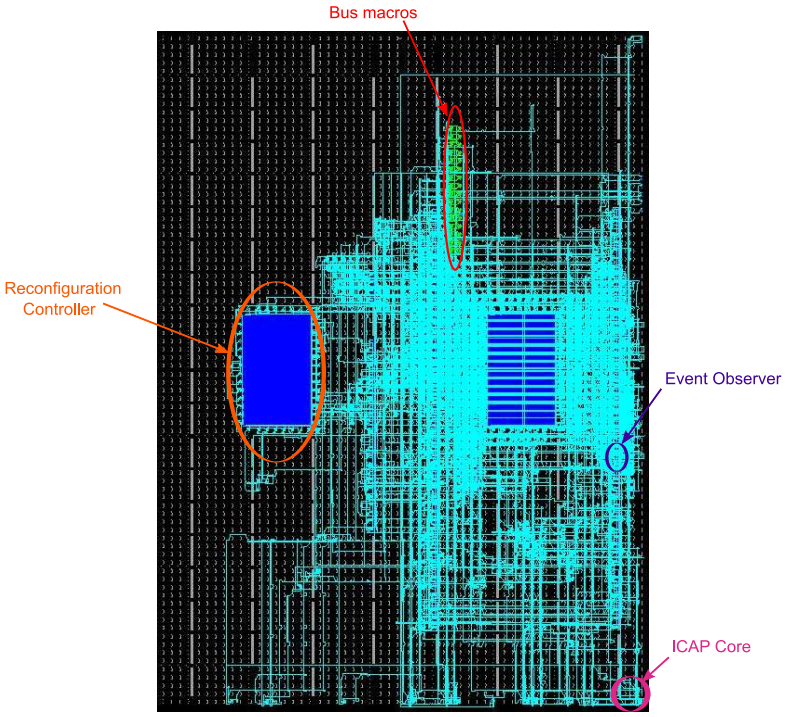


Figure 9.28: static bitstream

Finally 9.29 represents the initial bootup full bitstream that is a merge of the static and the DSP configuration partial bitstreams. The functionality of the different configurations during the configuration switch has been verified by the output to the monitoring unit in the radar system. During a configuration switch, the DECM does not generates an output; and hence no detection is carried out. Once a context switch occurs, the monitoring unit starts receiving the generated DECM outputs related to the switched configuration.

The results generated by the monitoring unit after a successful reconfiguration related to a hardware accelerator configuration are compared with those generated during a static implementation of the related configuration. A match was found between the two results, taking into account the time taken during the reconfiguration. The match ensured that an effective reconfiguration was carried out. The outputs of the DECM can also be verified using the Xilinx Chipscope tool[15]. However, this last step has not been carried out, during this dissertation.

It should be observed that during the case study, we only focused on one of the QoS criteria related to the available IPs, i.e., the FPGA resources consumed on an FPGA. However, other criteria such as dynamic power consumption, DSP blocks, performance throughput can also be selected, depending about designer requirements.

9.4.5 Design Space Exploration related to PDR

As explained previously in chapter 1, during this thesis, an application driven approach has been adapted. This is because a large number of related works already exist in the domain of partial dynamic reconfiguration, with special focus on optimizing architectural details. Consequently, the applications needed to be executed on the reconfigurable system are usually ignored. We now provide a brief exploratory study illustrating some of the optimizations possible at the RTL level. Additionally, this study proves that partial dynamic reconfiguration is a convoluted process and is directly dependent on several critical factors such as placement

[15] http://www.xilinx.com/support/training/abstracts/chipscope.htm

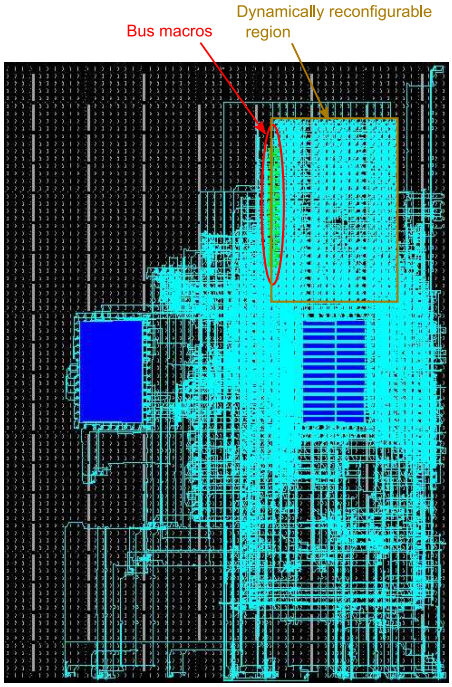


Figure 9.29: Full bitstream related to the PDR system

and shape of a PRR; available FPGA resources, timing constraints related to the created system; performance required by the system, etc. The PDR system presented previously is one of the implementation choices available to a designer. With respect to implementing PDR in a targeted FPGA, we analyze some of the main customizations possible:

- ***Controller type:*** For implementing self reconfiguration, the reconfiguration controller can either be an hardcore PowerPC or a softcore processor such as a Microblaze. The advantage of using an hardcore processor is that no additional resources such as slices are required, as compared to a softcore based approach. In Figure 9.30, the different configurations of the hardware accelerator are controlled by means of a softcore Microblaze processor. The processor is connected to the slave peripherals by means of an OPB bus. This controller introduces an additional overhead of **475** slices in the targeted FPGA. However, the reconfiguration times related to the two configurations reduce by a factor of approximately **1.7%**, with the softcore processor executing at 100 MHz. This is due to fact that while a PowerPC based system executes the related instructions in a single cycle, the presence of extra PLB bus and PLB to OPB bridge introduces additional latency during the reconfiguration. Normally a hardcore controller provides better performance, at the compromise of the throughput related to reconfiguration time. Table 9.3 shows the comparison of the two controllers with respect to reconfiguration times.

- ***PRR layout/shape:*** Similarly, modification of shape and the layout of the PRR has a direct impact on reconfiguration times. In Figures 9.31 and 9.32, the shape of the PRR has been modified by rotating it 90 degrees in a counter clock wise direction, making its width larger than the related height. The reconfiguration times for both configurations associated to this modified PRR are nearly two times greater than the time observed for the non modified PRR, using a hardcore processor. This is due to the reason that the ICAP performs the read-modify-write mechanism on a number of frames double to that of the original PRR. Table 9.4 provides the FPGA resources required for the PRR, while Table 9.5 gives the reconfiguration times.

- ***Bus macros type and placement:*** Bus macro placement also has a direct influence on the reconfiguration times, as well as system timing constraints. Unfortunately, the EAPR flow does not provides the designer with an optimum solution for the placement of these hard

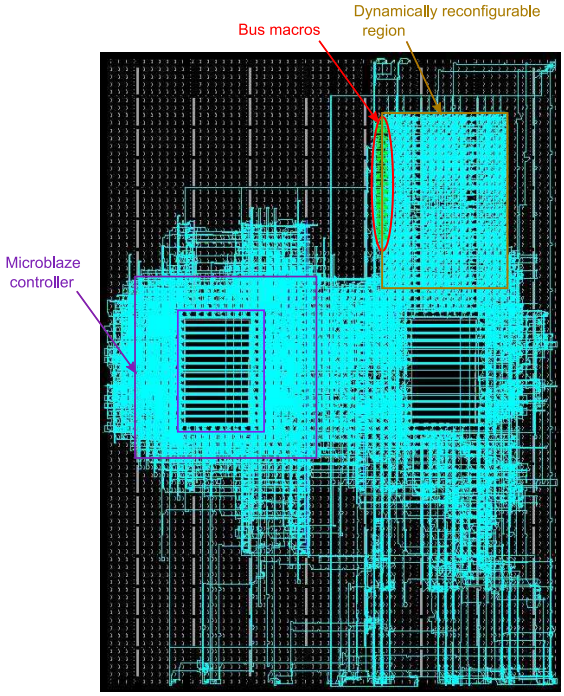


Figure 9.30: The PDR system with a Microblaze reconfiguration controller

	DSP configuration	If-then-else Configuration
PowerPC	1.45	1.41
Microblaze	0.85	0.79

Table 9.3: Comparison of the reconfiguration times (in secs) for both controller types

macros. Similarly the direction and type of the bus macros depends on the designer and target system requirements. A poorly placed bus macro can drastically change the overall result. It is up to the designer to determine the optimum position for the bus macros. This is one of the most painstakingly error-prone steps related to the EAPR flow.

- ***ICAP attachment:*** Moreover, for a hardcore based system; it is a designer's choice to attach the ICAP core with the PLB or the OPB bus. This choice is not present for a softcore based system. A DMA (Direct Memory Access) module can also be incorporated into the system, for enhancing the system performance. Similarly customized ICAP controllers can be created to speed up the reconfiguration times [146].

- ***Attachment of the hardware accelerator:*** Another option related to the optimization is the choice related to the attachment of the hardware accelerator. It can either be attached to the PLB/OPB buses in a hardcore based system or also to the FSL/OPB buses in a softcore based solution. Each different scenario causes different end results.

- ***Memory placement:*** Currently the stored partial bitstreams are placed in an external compact Flash memory. However, the usage of an external SRAM memory can also be integrated. This memory permits the partial bitstreams to be preloaded from the CF during initialization for decreasing the reconfiguration time.

- ***Targeted FPGA:*** Finally, PDR is also greatly affected by the choice of the targeted FPGA. Currently the EAPR methodology has been fully tested on the Xilinx Virtex-IV and Virtex-II/Pro series FPGAs only. While the older series FPGAs contain only a single ICAP core, FPGAs from Virtex-IV and onwards contain two ICAP primitives operating at greater frequencies. This hardware evolution can greatly improve the reconfiguration process.

Additionally, the reconfiguration granularity of these architectures has also been reduced, permitting to place different PRRs in the same FPGA device column.

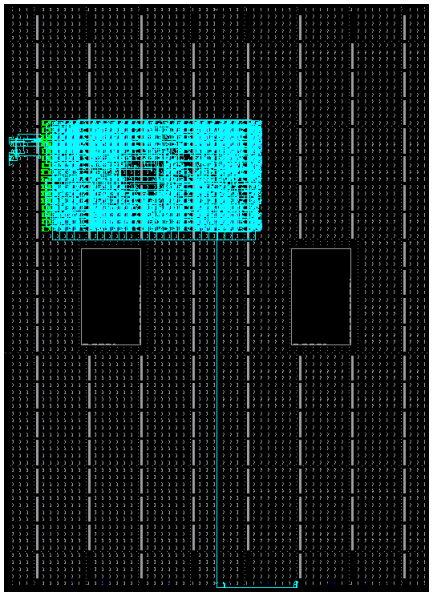


Figure 9.31: DSP configuration bitstream for the modified PRR

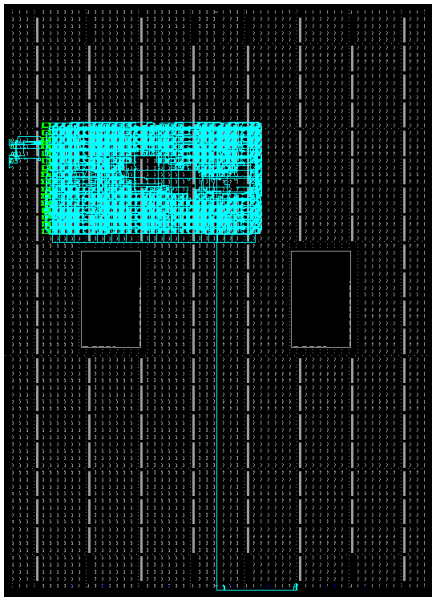


Figure 9.32: If-then-else configuration bitstream for the modified PRR

Slices	1600
Slice FlipFlops	3200
LUTs	3200
18x18 Multipliers	-
BRAM16s	-
TBUF	-

Table 9.4: PBlock requirements for the accelerator in the modified PRR

	DSP Configuration	If-then-else Configuration
Slices	1272/13696 (9.287%)	1186/13696 (8.659%)
Slice FlipFlops	2084/27392 (7.608%)	1944/27392 (7.096%)
LUTs	1584/27392 (5.782%)	1836/27392 (6.702%)
Reconfig. Time (secs)	2.87	2.85

Table 9.5: Results related to the two configurations

Summary As seen from the section related to dynamic reconfiguration, different combinations are possible for the creation of a PDR system. Each of these scenarios result in different end results, configuration throughputs, power consumption levels, etc. Thus it is not possible to determine the optimum de-facto system related to self configuration. This choice depends upon the targeted application domain, the selected architecture, designer requirements and the various tools required for the eventual implementation. Currently the EAPR flow is greatly affected by the underlying tools; and the designer has an initial hard learning curve to become an expert for manipulating the related tools. Similarly, the compatibility between the tools and the

architectures is also a critical issue. A case in point is that while the new Virtex-V and Virtex-VI series FPGAs support dynamic reconfiguration, the latest versions of the synthesis tools are not fully compatible with the EAPR flow[16].

9.5 Conclusion

This chapter provides a case study in order to validate our design flow for modeling and implementing dynamically reconfigurable systems. A critical portion of a complex anti-collision radar detection system has been initially modeled using the Gaspard2 environment. Afterwards, using the model transformation chain associated to our design flow, we generate the necessary code for implementing the PDR system. The functionality of the generated hardware accelerator has been verified by initial simulation. Later on, the generated code is taken as input by the EAPR flow for implementing the PDR system.

While a large part of RTL implementation related to the PDR system is carried out manually, this is due to limitations of the current design tools which makes it extremely difficult to elevate the design abstraction levels. Similarly, architectural modifications at this level have a significant impact of the overall results. This is evident as discussed in the last section related to the PDR system optimizations. We now present the conclusions and perspectives related to our thesis.

[16]The EAPR flow is currently only compatible with ISE/EDK 9.2, as compared to the latest versions of these tools

Conclusions

Summary

The works presented in this dissertation are carried out in the context of development of dynamically reconfigurable SoCs, dedicated to data intensive parallel computation (DIP) applications. These contributions have been integrated in a SoC Co-Design framework: Gaspard2. The framework benefits from a high abstraction level approach and is compliant with the MARTE profile for design and development of real-time embedded systems. It is important to note that through out the integration process, great attention has been paid to inspire from component based SoC conception methodologies based on models, for resolving the problems of escalating complexity related to SoC design. The result of the works is the proposal of a compilation chain for the development of dynamically reconfigurable SoCs, automatized and entirely revolving around principles related to Model-Driven Engineering. Finally, the thesis is a combination of contributions in different domains such as SoC Co-Design, Model-Driven Engineering and MARTE, as illustrated in the introductory chapter.

Application oriented high level design flow

The starting point of our design flow was to adapt an application-driven approach related to the construction of a dynamically reconfigurable SoC. As explained during the course of this document, nearly all of the efforts related to the reconfigurable domain focus on optimizing architectural details at the RTL level. In turn, the intended application functionalities to be executed on these systems are generally left ignored, or are not realistic in nature. While some researches promote elevation of design abstraction levels, they are normally incompatible with each other and do not achieve the advantages offered by design methodologies such as Model-Driven Engineering. Focusing on an application based design flow; we aim to provide MARTE compliant models related to key aspects of a reconfigurable SoC: mainly the application functionality linked to a dynamically reconfigurable hardware accelerator of the system; and control semantics, which perform the switching operations related to different implementations of the modeled application; and are in turn associated with a reconfiguration controller.

Generic control semantics

For the above mentioned control semantics, we first looked at control features present in the domain of component based software engineering, as discussed in chapter 2. A large number of component models exist in literature, but few offer features of dynamic adaptation and high abstraction levels. Moreover, models concentrating on real-time embedded systems are usually heavy weight in nature and incompatible with each other. For this we turn towards MARTE that offers a light weight approach based on UML *components*, as illustrated in chapter 3. One of our initial contributions is the proposal of a component based generic mode automata control semantics that can be applied to different levels of SoC Co-Design. The semantics can be expressed via MARTE profile, helping a designer to define control features at high modeling levels. While the semantics can be integrated at different design levels, we offer an interesting comparison and propose a novel solution: introduction of control at a level independent of other design levels, offering advantages of re-usable models and QoS based scenarios.

IP based configuration approach

Subsequently, the control semantics are integrated at the IP deployment level in Gaspard2. This amalgamation helps in introducing the notion of system *configurations* in Gaspard2. Each configuration can be associated to different system QoS criteria and can be viewed as an implementation of the modeled application: comprised of unique collection of *Intellectual Properties* (IPs), which are related to application elementary components. The combination of control and configurations at the deployment level makes it possible for a designer to change a configuration related to the modeled application. The change occurs due to the QoS choices present at the deployment level, with each different configuration offering diverse end results.

Extension of the MARTE metamodel

The proposed contributions were then added in MARTE, resulting in an extension of the current MARTE metamodel. Equally the MARTE profile respecting the metamodel was extended, for helping designers to represent the previously mentioned proposals in UML graphical modeling tools such as Papyrus. The metamodel extensions mainly relate to addition of MARTE compatible state machine concepts for modeling of mode automata; and the deployment level concepts inspired from Gaspard2. These extensions help the corresponding model transformations to interpret the high level models specified with the MARTE profile, for creation of intermediate models and aid in eventual code generation.

Hardware execution model for dynamic hardware accelerator

For successful translation of a high level application into a hardware accelerator, intended as the dynamically reconfigurable region; we propose an extension of the hardware execution model currently present in Gaspard2. While different execution models are possible for Gaspard2 applications; in the context of partial dynamic reconfiguration, we propose a parallel execution model that preserves characteristics of hierarchy, task and data parallelism of the modeled application, in the transformed hardware functionality.

RTL metamodel with dynamic aspects

One of the significant contributions carried out during the course of this dissertation is the enrichment of the existing RTL metamodel in Gaspard2. The RTL metamodel proposed in our design flow inspires from MARTE itself and adds notions such as components, behavior and automata. The RTL metamodel itself comprises of two significant portions, the first one related to concepts for transforming the modeled application into a hardware accelerator. Additionally, the configurations related to the application are transformed into different implementations of the accelerator. The second portion of the metamodel is related to the control semantics, it converts the modeled state machines into a mode automata for eventual execution in the reconfigurable controller. The RTL metamodel is independent from any specific targeted language for code generation. The concepts are generic enough to produce HDL (VHDL or Verilog code) for the hardware accelerator portion, or C/C++ or HDL code for control. Due to a personal choice, only VHDL code has been generated from the model transformations, while the choice of generating C/C++ code for the controller has already been explained earlier.

Model transformations

A significant contribution of this thesis is the development of MDE model-to-model/text transformations in the context of Gaspard2. The model-to-model transformations written in QVTO, take as input the UML diagram specified with the MARTE profile and transform it into successive intermediate models, each corresponding to its proper metamodel. Each intermediate model adds additional information, with the intended goal of bridging the gap between high level diagrams and the generated code. The model-to-model transformations finally result in the RTL model, which provides an abstraction level near to the electronic Register Transfer Level. The proximity between the RTL model and the eventual code helps the model-to-text

transformation in generating the code for different implementations of the dynamic hardware accelerator and the reconfiguration controller.

Experimental validation

Finally we provide an experimental validation of our design methodology by providing a case study related to an anti-collision radar detection system. A key component of this system is modeled in Gaspard2 and is eventually converted into a dynamic hardware accelerator. The simulation results related to different implementations of the accelerator verify that a correct transformation has occurred. Finally, the hardware accelerator is integrated into the PDR system in the form of an IP-core. The reconfiguration controller carries out the configuration switch related to the different implementations, by executing the code generated from our design flow. The outputs produced by the reconfigurable hardware accelerator are validated by a monitoring module in the radar system, helping in validation of our design methodology.

Discussion

Reduced development time

One of the advantages of our design flow is the reduction of design time required for the specification of a complex dynamically reconfigurable system. With regards to the case study, in [72], the authors developed a hand tuned version of the application in approximately five months. Afterwards, they carried out the development of a parametric VHDL based correlator generator tool. The tool was implemented in C++ and permitted to change correlation parameters, such as the length of PRBS, data resolution, etc. With respect to an hand tuned implementation, the development time for a model driven design flow may be comparatively a bit lengthier. However, once the transformations are developed, a designer can change the application specifications in a matter of hours. Additionally, while the hand-tuned development time is related to one specific application, the model transformations in our design flow can be used for a wide range of applications supported by Gaspard2.

Advantages of MDE

Finally, an MDE design flow simplifies the construction of a SoC, by providing simple graphical notations, which are easily comprehensible. These notations liberate users from heavy syntax and grammar of classical languages, facilitating in rapid system development. Additionally, the model transformations can be viewed as more efficient and flexible with regards to classical compilers. An example can be of the correlator generator tool discussed above. Change in the functionality of the tool can be a complex and difficult task, as compared to model transformation rules that are modular and independent from each other. Thus new functionalities can be easily integrated, while existing ones can be modified due to the nature of the transformations. Similarly, intermediate models can be introduced in the compilation chain, helping in evolution of a design flow. Finally, the tools associated with or dedicated to MDE are evolving, permitting good support for MDE based development. As MARTE move towards final standardization process, future versions of UML modeling tools will support MARTE profile, from high level models to code generation phases.

Perspectives

Need of a complex reconfiguration mechanism

Currently in our design flow, the integrated reconfiguration controller is simplistic in nature, and depends upon external user inputs for carrying out the configuration switches. An intelligent controller can be extremely beneficial in a future extension of the flow, permitting a context switch without user intervention. Thus a resulting hardware accelerator from a modeled

application can be executed on a targeted architecture, in turn actively communicating with the controller. Upon termination of its current functional execution, the accelerator can signal the controller to perform an effective reconfiguration. The controller thus should be capable to integrate an efficient data management mechanism for preserving the data flow before/after the switch. Additionally, a real-time operating system (RTOS) or middeware can be employed for handling this mechanism in a complex reconfigurable SoC. One of the dilemmas of this approach is expressing the low level details related to a complex controller at high design abstraction levels. Another present problem is the model of computation, namely Array-OL associated with our design flow. While our introduced extensions expand the MoC with dynamic aspects, they are insufficient enough to address data flow management strategies. Hence a robust dynamic MoC is required, that can be expressed in MARTE profile and is capable of handling the various issues related to dynamic reconfiguration.

Control at different SoC Co-Design levels

As seen earlier in the document, we have presented a generic control semantics that can be applied onto different levels of a SoC Co-Design framework. While during the course of this thesis, we have focused on its integration at the IP deployment level in Gaspard2, it can be equally incorporated in other models; such as the application level. This will permit to change one modeled application functionality by another, while respecting the control semantics. Equally, a global controller can be integrated into the system for handling local controllers at different design levels. Additionally, control at deployment level can be extended for the formation of complex configurations. The granularity level of the configurations can be reduced, i.e., a configuration can be allocated to a specific partition of the available application, resulting in the creation of *partial configurations*. Moreover, nested configurations can be created, culminating in the creation of a nested state graph and a subsequent mode automata having several hierarchical levels. In consequence, careless switching of these partial configurations may produce instability in the end results, necessitating a mechanism for managing this extension.

Extension of current execution model

The compilation chain presented in our design flow currently transforms the whole specified modeled application into a single hardware accelerator; however, new transformation rules can be added to the compilation chain, making it possible to implement key kernels of the application onto a single or separate hardware accelerators, while other non critical portions can be sequentially executed on available resources such as hard/soft microprocessors. This adds an additional layer of complexity in the design, for providing synchronization mechanisms of the related control/data flow and the communication in the overall system. This manner of allocation can be simplified by providing high level constructs showcasing the complete system (application, architecture and allocation). Furthermore, while we currently focus on a parallel hardware execution of the modeled application, a sequential execution may be interesting from a QoS point of view. The choice of an execution should be expressible at the modeling level, providing designers with different execution choices. Additionally, parametrable code generation can be an interesting approach, by utilizing the concepts of templates present in UML specifications. Correspondingly, current model transformations should be evolved to interpret these new concepts specified via the MARTE profile.

Targeting heterogeneous MPSoC based systems

Similarly, MPSoC based systems can be targeted from a future extension of our design flow. Currently, research related to this aspect has been started in the thesis of Chiraz Trabelsi in our project-team. The goal is to implement an MPSoC system with heterogeneous processors such as PowerPC and Microblaze, with each processor attached to a monitoring module for observing the processor execution activities. The module can then send input to a global controller for an affective configuration of the processor in question. The development of such a system at the MARTE profile level needs high level modeling of the system, as illustrated in appendix A.

Additionally, new transformation rules need to be developed, as well as intermediate metamodels. An interesting point can be the generation of SystemC code for the system, and afterwards, move onto HDL translation. This could bridge the gap between the simulation and synthesis compilation chains currently existing separately in Gaspard2.

Bibliography

[1] 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. 22, 26

[2] 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. 59

[3] A. Koudri et al. Using MARTE in the MOPCOM SoC/SoPC Co-Methodology. In *MARTE Workshop at DATE'08*, 2008. 25, 75, 82

[4] A. Tumeo and M. Monchiero and G. Palermo and F. Ferrandi and D. Sciuto. A Self-Reconfigurable Implementation of the JPEG Encoder. *ASAP 2007*, pages 24–29, 2007. 26, 82, 130, 190

[5] AADL. The architecture analysis & design language (aadl): An introduction. http://www.sei.cmu.edu/publications/documents/06.reports/06tn011.html, 2006. V, 34, 56

[6] A. S. AB. Rubus operating system. http://www.arcticus-systems.com/, 2009. 37

[7] Acceleo. MDA code generator, 2009. http://www.acceleo.org/pages/home/en. 169

[8] M. D. Adams. The JPEG-2000 still image compression standard. Technical Report Report N2412, ISO/IEC JTC 1/SC 29/WG 1, JPEG, septembre 2001. 61

[9] N. Aizenbud-Resher, R. F. Paige, J. Rubin, Y. Shalam-Gafni, and D. S. Kolovos. Operational semantics for traceability. In *ECMDA Traceability Work-shop (ECMDA-TW) 2005*, 2005. 48

[10] D. H. Akehurst, O. Uzenkov, W. G. Howells, K. D. Mcdonald-maier, and B. Bordbar. Compiling uml state diagrams into vhdl: An experiment. In *Forum on specification and Design Languages (FDL'07)*, 2007. 131

[11] A. Amar, P. Boulet, and P. Dumont. Projection of the Array-OL specification language onto the Kahn process network computation model. In *Proceedings of the International Symposium on Parallel Architectures, Algorithms, and Networks, Las Vegas, Nevada, USA*, December 2005. 66

[12] C. André. Syncharts: a visual representation of reactive behaviors. Technical Report RR 95–52, rev. RR (96–56), I3S, Sophia-Antipolis, France, Rev. April 1996. 230

[13] C. André, A. Mehmood, F. Mallet, and R. Simone. Modeling spirit ip-xact in uml-marte. In *Design Automation and Test in Europe (DATE), MARTE Workshop*, 2008. 118

[14] 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. 88

[15] Atat, Y., and Zergainoh, N. Simulink-based MPSoC Design: New Approach to Bridge the Gap between Algorithm and Architecture Design. In *ISVLSI'07*, pages 9–14, 2007. 76

[16] R. B. Atitallah, E. Piel, S. Niar, P. Marquet, and J.-L. Dekeyser. Multilevel MPSoC simulation using an MDE approach. In *SoCC 2007*, 2007. 71, 72

[17] 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. 19, 20, 22, 23

[18] 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. 88

[19] B. Bailey, G. Martin, and A. Piziali. *ESL Design and Verification: A Prescription for Electronic System Level Methodology*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2007. 43

[20] S. Balakrishnan and C. Eddington. Efficient dsp algorithmdevelopment for fpga and asic technologies. Technical report, Synplicity, 2001. 130

[21] G. Bastide, A. Seriai, and M. Oussalah. Adapting software components by structure fragmentation. In *SAC '06: Proceedings of the 2006 ACM symposium on Applied computing*, pages 1751–1758. ACM, 2006. 34

[22] 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. 22, 25, 26

[23] S. Becker and R. Reussner. The impact of software component adaptors on quality of service properties. In *First International Workshop on Corordination and Adaptation Techniques for Software Entities (WCAT'04), L'objet 12*, pages 105–125, 2006. 34

[24] T. Becker, W. Luk, and P. Cheung. Enhancing relocatability of partial bitstreams for run-time reconfiguration. In *15th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM 2007)*, pages 35–44, 2007. 27

[25] S. Bell, B. Edwards, J. Amann, R. Conlin, K. Joyce, V. Leung, J. MacKay, M. Reif, B. Liewei, J. Brown, M. Mattina, M. Chyi-Chang, C. Ramey, D. Wentzlaff, W. Anderson, E. Berger, N. Fairbanks, D. Khan, F. Montenegro, J. Stickney, and J. Zook. Tile64 - processor: A 64-core soc with mesh interconnect. In *IEEE International Digest of Technical Papers on Solid-State Circuits Conference (ISSCC 2008)*, pages 88–598, 2008. 11, 33

[26] P. Bellows and B. Hutchings. Jhdl - an hdl for reconfigurable systems. In *Proceedings of the IEEE Symposium on FPGAs for Custom Computing Machines (FCCM '98)*, page 175. IEEE Computer Society, 1998. 129

[27] R. Ben Atitallah. *Modèle et Simulation des système sur purce multiprocesseurs - Estimation des performances et de la consommation d'énergie*. PhD thesis, USTL, 2008. 58, 70, 73, 111, 112, 114

[28] L. Benini and G. D. Micheli. Networks on chips: a new SoC paradigm. *Computer*, 35(Issue: 1):70–78, 2002. 11

[29] L. Besnard, T. Gautier, and P. L. Guernic. *Signal Reference Manual.*, 2006. www.irisa.fr/espresso/Polychrony. 62

[30] S. L. Beux, V. Gagne, E. M. Aboulhamid, P. Marquet, and J.-L. Dekeyser. Hardware/software exploration for an anti-collision radar system. In *49th IEEE International Midwest Symposium on Circuits and Systems (MWSCAS '06)*, volume 1, pages 385–389, 2006. 129, 180, 182, 185

[31] S. L. Beux, P. Marquet, and J.-L. Dekeyser. Model driven engineering benefits for high level synthesis. Technical Report 6615, INRIA, 2008. 133

[32] J. Bézivin. On the unification power of models. *Software and System Modeling (SoSym)*, 4(2):171–188, 2005. 42

[33] B. Blodget, P. James-Roxby, E. Keller, S. McMillian, and P. Sundararajan. A self-reconfiguration platform. In *Proceedings of the Field-Programmable Logic and Applications (FPL'03)*, pages 565–574, 2003. 26

[34] K. Bondalapati and V. K. Prasanna. Reconfigurable computing systems. In *Proceedings of the IEEE*, number 7, pages 1201–1217, 2002. 13

[35] L. Bossuet, G. Gogniat, and J.-L. Philippe. Communication-oriented design space exploration for reconfigurable architectures. *EURASIP Journal of Embedded Systems*, 2007(1):2–2, 2007. 18

[36] P. Boulet. Array-OL revisited, multidimensional intensive signal processing specification. Research Report RR-6113, INRIA, http://hal.inria.fr/inria-00128840/en/, February 2007. 60, 63, 65, 67, 92, 134, 167

[37] P. Boulet. Formal Semantics of Array-OL, a Domain Specific Language for Intensive Multidimensional Signal Processing. Technical Report 6467, INRIA, France, March 2008. http://hal.inria.fr/inria-00261178/en. 63, 65, 134

[38] A. Brito, M. Kuhnle, M. Hubner, J. Becker, and E. Melcher. Modelling and simulation of dynamic and partially reconfigurable systems using systemc. In *Proceedings of the IEEE Computer Society Annual Symposium on VLSI (ISVLSI'07)*, pages 35–40. IEEE Computer Society, 2007. 25

[39] E. Bruneton, T. Coupaye, M. Leclercq, V. Quéma, and J. Stefani. The fractal component model and its support in java: Experiences with auto-adaptive and reconfigurable systems. *Software: Practice and Experience*, 36(11-12):1257–1284, 2006. 34, 35, 36, 38

[40] J. Buck, S. Ha, E. Lee, and D. Messerschmitt. Ptolemy: A framework for simulating and prototyping heterogeneous systems. *International Journal of computer Simulation*, 4:155–182, April 1994. Special issue on Simulation Software Development. 76, 88

[41] T. Bures, P. Hnetynka, and F. Plasil. Sofa 2.0: Balancing advanced features in a hierarchical component model. In *Fourth International Conference on Software Engineering Research, Management and Applications*, pages 40–48, 2006. 36

[42] T. Bureš, P. Hnětynka, and M. Malohlava. Using a product line for creating component systems. In *Proceedings of the 2009 ACM symposium on Applied Computing (SAC'09)*, pages 501–508, 2009. 37

[43] C. Claus and B. Zhang and W. Stechele and L. Braun and M. Hubner and J. Becker. A Multi-platform controller allowing for maximum dynamic partial reconfiguration throughput. pages 1–7, 2008. 23, 26

[44] 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. 23, 26, 190

[45] C. Schuck and B. Haetzer and J. Becker. An Interfance for a Decentralized 2D-Reconfiguration on Xilinx Virtex-FPGAs for Organic Computing. In *ReCoSoC'08*, 2008. 26

[46] C. Schuck and M. Kuhnle and M. Hubner and J. Becker. A framework for dynamic 2D placement on FPGAs. In *IPDPS 2008*, 2008. 22

[47] L. Cai and D. Gajski. Transaction level modeling: an overview. In *CODES+ISSS'03*, 2003. 60

[48] F. Cancare, M. D. Santambrogio, and D. Sciuto. An application-centered design flow for self reconfigurable systems implementation. In *Asia and South Pacific Design Automation Conference (ASP-DAC 2009)*, pages 248–253, 2009. 27

[49] E. Canto, F. Fons, and M. Lopez. Reconfigurable opb coprocessors for a microblaze self-reconfigurable soc mapped on spartan-3 fpgas. In *32nd Annual Conference on IEEE Industrial Electronics (IECON 2006)*, pages 4940–4944, 2006. 26

[50] C. Choi and H. Lee. An reconfigurable fir filter design on a partial reconfiguration platform. In *First International Conference on Communications and Electronics (ICCE '06)*, pages 352–355, 2006. 183

[51] T. A. Claasen. System on a Chip: Changing IC design today and in the future. *IEEE Micro*, 23(3):20–26, May/Jun 2003. V, 11, 12

[52] C. Claus, F. H. Muller, and W. Stechele. Combitgen: A new approach for creating partial bitstreams in virtex-ii pro devices. In *Workshop on reconfigurable computing Proceedings (ARCS 06)*, pages 122–131, 2006. 27

[53] C. Claus, B. Zhang, M. Huebner, C. Schmutzler, J. Becker, and W. Stechele. An xdl-based busmacro generator for customizable communication interfaces for dynamically and partially reconfigurable systems. In *Workshop on Reconfigurable Computing Education at ISVLSI 2007*, 2007. 26

[54] CoFluent Design. System modeling and architecturing with cofluent studio. http://www.cofluentdesign.com/, 2009. 75

[55] K. Compton and S. Hauck. Reconfigurable computing: a survey of systems and software. *ACM Comput. Surv.*, 34(2):171–210, 2002. V, 14, 15

[56] G. Coulson, G. Blair, P. Grace, F. Taiani, A. Joolia, K. Lee, J. Ueyama, and T. Sivaharan. A generic component model for building systems software. *ACM Transactions on Computer Systems*, 26(1):1–42, 2008. 37

[57] P. Coussy, G. Corre, P. Bomel, E. Senn, and E. Martin. High-level synthesis under i/o timing and memory constraints. In *IEEE International Symposium on Circuits and Systems, (ISCAS 2005)*, pages 680–683 Vol. 1, 2005. 130

[58] F. P. Coyle and M. A. Thornton. From UML to HDL: a model driven architectural approach to hardware-software co-design. pages 88–93, 2005. 130

[59] M. C.Williamson. *Synthesis of Parallel Hardware Implementations from Synchronous Dataflow Graph SpecificationsR*. PhD thesis, EECS Department, University of California, 1998. 130

[60] K. Czarnecki and S. Helsen. Classification of model tranformation approaches. In *Proceeding of OOPSLA Workshop on Generative Techniques in the Context of Model Driven Architecture*, 2003. 47

[61] D. Björklund and J. Lilius. From UML behavioral descriptions to efficient synthesizable VHDL. In *Proceedings of 20th IEEE NORCHIP Conference*, pages 11–12, 2002. 131

[62] R. Damasevicius and V. Stuikys. Application of UML for hardware design based on design process model. In *ASP-DAC'04*, 2004. 130

[63] DaRT team. Graphical Array Specification for Parallel and Distributed Computing (GASPARD2), 2009. http://www.gaspard2.org/. 59, 63

[64] J. Davis, T. Zhangxi, F. Yu, and L. Zhang. A practical reconfigurable hardware accelerator for boolean satisfiability solvers. In *Proceedings of the 45th annual Design Automation Conference (DAC'08)*, pages 780–785. ACM, 2008. 130

[65] D. de Niz, G. Bhatia, and R. Rajkumar. Model-based development of embedded systems: The sysweaver approach. In *Proceedings of the 12th IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 231–242, 2006. 37

[66] J.-L. Dekeyser, A. Gamatié, S. Meftali, and I. R. Quadri. *Heterogeneous Embedded Systems - Design Theory and Practice*, chapter High-level modeling of dynamically reconfigurable heterogeneous systems. Springer, 2010. 89

[67] A. Demeure and Y. Del Gallo. An array approach for signal processing design. In *Sophia-Antipolis conference on Micro-Electronics (SAME'98), System-on-Chip Session, France*, October 1998. 65

[68] A. Demeure, A. Lafage, E. Boutillon, D. Rozzonelli, J. Dufourd, and J. Marro. Array-OL: Proposition d'un formalisme tableau pour le traitement de signal multi-dimensionnel. In *Colloque GRETSI sur le Traitement du Signal et de l'Image, Juan-Les-Pins, France*, September 1995. 65

[69] U. B. E. Department. Ptolemy project: Heterogeneous modeling and design. http://ptolemy.eecs.berkeley.edu/, 2009. 37

[70] C. Dezan. *Géneration automatique de circuits avec ALPHA du CENTAUR*. PhD thesis, IRISA, 1993. 129

[71] A. Donlin. Transaction level : flows and use models. In *CODES+ISSS'04*, 2004. 60

[72] L. Douadi, P. Deloof, and Y. Elhillali. Real time implementation of reconfigurable correlation radar for road anticollision system. In *IEEE International Conference on Industrial Technology (ICIT 2008)*, pages 1–7, 2008. 180, 181, 212

[73] J. Dowling and V. Cahill. The k-component architecture meta-model for self-adaptive software. In *REFLECTION '01: Proceedings of the Third International Conference on Metalevel Architectures and Separation of Crosscutting Concerns*, pages 81–88. Springer-Verlag, 2001. 34, 36

[74] 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. 33

[75] P. Dumont. *Spécification multidimensionnelle pour le traitement du signal systématique*. PhD thesis, Université des Sciences et Technologies de Lille, France, December 2005. Dumont Philippe. 85

[76] P. Dumont and P. Boulet. Another multidimensional synchronous dataflow: Simulating Array-Ol in Ptolemy II. Technical Report 5516, INRIA, France, March 2005. available at www.inria.fr/rrrt/rr-5516.html. 66

[77] E. Brinksma and G. Coulson and I. Crnkovic and A. Evans and S. Gérard and S. Graf and H. Hermanns and J. Jézéquel 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. 29, 31, 32, 33, 38

[78] E. Canto and F. Fons and M. Lopez. Self-recofigurable embedded systems on Spartan-3. In *International Conference on Field Programmable Logic and Applications, FPL 2008*, pages 571–574, 2008. 26

[79] C. Ebeling, D. Cronquist, and P. Franklin. Rapid - reconfigurable pipelined datapath. In *Proceedings of the 6th International Workshop on Field-Programmable Logic, Smart Applications, New Paradigms and Compilers (FPL' 96)*, pages 126–135. Springer-Verlag, 1996. 14

[80] Eclipse. Eclipse Modeling Framework. http://www.eclipse.org/emf. 45

[81] Eclipse. Eclipse Modeling Framework Technology, 2009. http://www.eclipse.org/emft. 74

[82] Eclipse. EMFT JET, 2009. http://www.eclipse.org/emft/projects/jet. 49, 74, 170

[83] H. Espinoza. *An Integrated Model-Driven Framework for Specifying and Analyzing Non-Functional Properties of Real-Time Systems*. PhD thesis, University of Evry, FRANCE, 2007. 56

[84] H. Espinoza, D. Cancila, B. Selic, and S. Gérard. Challenges in combining sysml and marte for model-based design of embedded systems. In *ECMDA-FA '09: Proceedings of the 5th European Conference on Model Driven Architecture - Foundations and Applications*, pages 98–113. Springer-Verlag, 2009. 55

[85] M. Eva. *SSADM Version 4: A User's Guide*. McGraw-Hill Publishing Co, april 1994. 44

[86] 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. 76

[87] M. Faugere, T. Bourbeau, R. Simone, and G. Sebastien. Marte: Also an uml profile for modeling aadl applications. In *ICECCS '07: Proceedings of the 12th IEEE International Conference on Engineering Complex Computer Systems*, pages 359–364. IEEE Computer Society, 2007. 88

[88] M. Faugere, T. Madeleine, R. Simone, and S. Gerard. Marte: Also an uml profile for modeling aadl applications. In *ICECCS '07: Proceedings of the 12th IEEE International Conference on Engineering Complex Computer Systems*, pages 359–364. IEEE Computer Society, 2007. 56

[89] J.-M. Favre, J. Estublier, and M. Blay-Fornarino, editors. *L'ingénierie dirigée par les modèles, au-delà du MDA*. Hermès Science, Lavoisier, Jan. 2006. 42, 44

[90] H. Fecher, J. Schönborn, M. Kyas, and W. P. de Roever. 29 new unclarities in the semantics of UML 2.0 state machines. In *ICFEM*, volume 3785 of *Lecture Notes in Computer Science*, pages 52–65. Springer, 2005. 50

[91] S. Fritsch, A. Senart, D. Schmidt, and S. Clarke. Time-bounded adaptation for automotive system software. In *Proceedings of the 30th international conference on Software engineering (ICSE '08)*, pages 571–580. ACM, 2008. 34

[92] G. Gailliard and E. Nicollet and M. Sarlotte and F. Verdier. Transaction level modelling of SCA compliant software defined radio waveforms and platforms PIM/PSM. In *Design, Automation & Test in Europe, DATE'07*, 2007. 76

[93] G. Perrouin and F. Chauvel and J. DeAntoni and J. Jzquel. Modeling the Variability Space of Self-Adaptive Applications. In *Dynamic Software Product Lines Workshop*, pages 15–22, 2008. 36

[94] D. D. Gajski and R. Kuhn. Guest editor introduction : New VLSI-tools. *IEEE Computer*, 16(12):11–14, Dec. 1983. 11

[95] A. Gamatié, E. 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. 88, 89, 90, 94, 97

[96] A. Gamatié, H. Yu, G. Delaval, and E. Rutten. A case study on controller synthesis for data-intensive embedded systems. In *6th IEEE Int. Conference on Embedded Systems and Software (ICESS'09)*. © IEEE Press, May 2009. 88, 89, 97

[97] C. Glitia. *Optimisation des applications de traitement systematique intensives sur Systems-on-Chips*. PhD thesis, USTL, 2009. 62

[98] C. Glitia and P. Boulet. High Level Loop Transformations for Multidimensional Signal Processing Embedded Applications. In *International Symposium on Systems, Architectures, MOdeling, and Simulation (SAMOS VIII)*, 2008. 85

[99] GRACE++. System level design environment for network-on-chip (noc) and multi-processor platform (mp-soc) exploration. www.iss.rwth-aachen.de/Projekte/grace/index.html, 2009. 76

[100] S. Graf. Omega – Correct Development of Real Time Embedded Systems. *SoSyM, int. Journal on Software & Systems Modelling*, 7(2):127–130, 2008. 51, 88

[101] T. Grandpierre, C. Lavarenne, and Y. Sorel. Optimized rapid prototyping for real-time embedded heterogeneous multiprocessors. In *Proceedings of 7th International Workshop on Hardware/Software Co-Design (CODES'99)*, 1999. 75

[102] Z. Guo, B. Buyukkurt, W. Najjar, and K. Vissers. Optimized generation of data-path from c codes for fpgas. In *Proceedings of the conference on Design, Automation and Test in Europe (DATE '05)*, pages 112–117. IEEE Computer Society, 2005. 45

[103] S. Gupta, N. Dutt, R. Gupta, and A. Nicolau. Spark : A high-lev l synthesis framework for applying parallelizing compiler transformations. *VLSI Design, International Conference on*, 0:461, 2003. 130

[104] H. Yu. *A MARTE based reactive model for data-parallel intensive processing: Transformation toward the synchronous model*. PhD thesis, USTL, 2008. 73, 77, 81, 88, 89, 90, 91, 92, 94, 96, 97, 122

[105] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous dataflow programming language Lustre. *Proceedings of the IEEE*, 79(9), September 1991. 88

[106] N. Halbwachs and D. Pilaud. Use of a real-time declarative language for systolic array design and simulation. In *International Workshop on Systolic Arrays*, Oxford, July 1986. article not found. 62

[107] D. Harel. Statecharts: A Visual Formalism for Complex Systems. *Science of Computer Programming*, 8(3):231–274, June 1987. article note found. 88, 89, 90, 92, 103

[108] D. Harel and A. Naamad. The STATEMATE Semantics of Statecharts. *ACM Transactions on Software Engineering and Methodology*, 5(4):293–333, 1996. 92

[109] R. Hartenstein. A decade of reconfigurable computing: a visionary retrospective. In *Proceedings of the conference on Design, automation and test in Europe (DATE 01)*. IEEE Press, 2001. 13

[110] J. Hauser and J. Wawrzynek. Garp: A mips processor with a reconfigurable coprocessor. In *In IEEE Symposium on FPGAs for Custom Computing Machines*, pages 12–21, 1997. 15

[111] High Performance Fortran Forum. High performance fortran language specification, January 1997. http://hpff.rice.edu/versions/hpf2/index.htm. 62

[112] Y. E. Hillali. *Etude et réalisation d'un système de communication et de localisation, basé sur les techniques d'étalement de spectre aux transports guidé*. PhD thesis, University of Valenciennes, France, 2005. 110

[113] HoneyWell. Metah. http://www.htc.honeywell.com/metah/index.html, 2009. 37

[114] E. L. Horta and J. W. Lockwood. Parbit: A tool to transform bitfiles to implement partial reconfiguration of field programmable gate arrays (fpgas). Technical report, Washington University, Department of Computer Science, 2001. 26

[115] E. L. Horta, J. W. Lockwood, D. E. Taylor, and D. Parlour. Dynamic hardware plugins in an fpga with partial run-time reconfiguration. volume 0, page 343. IEEE Computer Society, 2002. 26

[116] Huafeng Yu and Abdoulaye Gamatié and Éric Rutten and Jean-Luc Dekeyser. Safe Design of High-Performance Embedded Systems in a MDE framework. *Innovations in Systems and Software Engineering, ISSE*, 4(3), 2008. 88, 89, 97

[117] IBM Corporation. The Coreconnect Bus Architecture, white paper. In *International Business Machines Corporation*, 2004. 191, 197

[118] C. Im, H. Kim, and S. Ha. Dynamic voltage scheduling technique for low-power multimedia applications using buffers, 2001. 95

[119] INRIA Atlas Project. ATL. http://modelware.inria.fr/rubrique12.html. 48

[120] INRIA Triskell Project. Kermeta. http://www.kermeta.org/. 36, 48

[121] ITRS. International technology roadmap for semiconductors. design, 2005 edition. http://www.xilinx.com/prs_rls/design_win/0412_marsrover.htm, 2005. 27

[122] 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. 19, 22, 24, 26

[123] J. Carver and R. Neil Pittman and A. Forin. Relocation and Automatic Floor-planning of FPGA Partial Configuration Bit-Streams. Technical report, Microsoft Research, 2008. 27

[124] J. Delahaye and G. Gogniat and C. Roland. Software Radio ad Dynamic Reconfiguration on a DSP/FPGA Platform. In *3rd Karlsruhe Workshop on Software Radios*, pages 143–151, 2004. 19

[125] J. Polakovic and J. Stefani. Architecting reconfigurabe component based operating systems. In *Journal of Systems Architecture*, volume 54, pages 562–5755, 2008. 36, 37

[126] J. Taillard. *Une approche orientée modÌle pour la parallélisation d'un code de calcul éléments finis*. PhD thesis, USTL, 2009. 73

[127] J. Vidal and F. De Lamotte and G. Gogniat. A co-design approach for embedded system modeling and code generation with UML and MARTE. In *Design, Automation and Test in Europe (DATE'09)*, 2009. 25, 58, 75, 82

[128] A. Jerraya and W. Wolf, editors. *Multiprocessor Systems-on-Chip*. Elsevier Morgan Kaufmann, San Francisco, California, 2005. 11

[129] J.P. Diguet and G. Gogniat and J.-L. Philippe and Y. Moullec and S. Bilavarn and C. Gamrat and K. Chehida and M. Auguin and X. Fornari and P. Kajfasz. EPICURE: A partitioning and co-design framework for reconfigurable computing. *Journal of Microprocessors and Microsystems*, 30(6):367–387, 2006. 75

[130] 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. 25, 26

[131] H. Kalte, G. Lee, M. Porrmann, and U. Ruckert. Replica: A bitstream manipulation filter for module relocation in partial reconfigurable systems. *International Parallel and Distributed Processing Symposium*, 4:151b, 2005. 26

[132] A. Khan, C. A. F. Mallet, and R. D. Simone. Marte timing requirement and spirit ip-xact. Research Report RR-6647, INRIA, http://hal.archives-ouvertes.fr/inria-00321953/en/, 2009. 198

[133] D. Koch, C. Beckhoff, and J. Teich. Recobus-builder: A novel tool and technique to build statically and dynamically reconfigurable systems for fpgas. In *International Conference on Field Programmable Logic and Applications (FPL'08)*, pages 119–124, 2008. 26

[134] H. Kopetz and N. Suri. Compositional design of rt systems: a conceptual basis for specification of linking interfaces. In *Object-Oriented Real-Time Distributed Computing, 2003. Sixth IEEE International Symposium on*, pages 51–60, 2003. 34

[135] K. Kronlof, S. Kontinen, I. Oliver, and T. Eriksson. A method for terminal platform architecture development. In *Proceedings of the Forum on Specification and Design Languages (FDL'06)*, pages 337–343, 2006. 75

[136] O. Kwon, S. Yoon, and G. Shin. Component-based development environment: An integrated model of object-oriented techniques and other technologies. In *Proceedings of the 2nd International Workshop on CBSE*, 1999. 43

[137] L. Bass and P. Clements and R. Kazman. Software Architecture In Practice. *Addison Wesley*, 1998. 33

[138] L. Ober and L. Ober and S. Graf and D. Lesens. Projet Omega : Un profil UML et un outil pour la modelisation et la validation de systemes temps reel. pages 73: 33–38, 2005. 76

[139] O. Labbani. *Modélisation à haut niveau du contrôle dans des applications de traitement systématique à parallélisme massif*. PhD thesis, USTL, 2006. 77, 88, 89, 99

[140] M. Lajolo and M. Prevostini. Uml in an electronic system level design methodology. In *In DAC'04 UML for SoC workshop*, 2004. 75

[141] 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. 88

[142] C. Lavarenne, O. Seghrouchni, Y. Sorel, and M. Sorine. The SynDEx software environment for real-time distributed systems, design and implementation. In *Proceedings of European Control Conference, ECC'91*, Grenoble, France, July 1991. 75

[143] S. Le Beux. *Un flot de conception pour applications de traitement du signal systématique implémentées sur FPGA à base d'Ingénierie Dirigée par les Modèles*. PhD thesis, Université des Sciences et Technologies de Lille (USTL), 2007. 18, 73, 81, 128, 130, 131, 132, 133, 136, 138, 167, 168, 197

[144] E. Lee and D. Messerschmitt. Static scheduling of synchronous data flow programs for digital signal processing. *IEEE Transactions on Computers*, 36(1):24–35, 1987. 62, 130

[145] M. Léger, T. Ledoux, and T. Coupaye. Reliable dynamic reconfigurations in the fractal component model. In *Proceedings of the 6th international workshop on Adaptive and reflective middleware (ARM '07)*, pages 1–6. ACM, 2007. 35, 36

[146] M. Liu, W. Kuehn, Z. Lu, and A. Jantsch. Run-time partial reconfiguration speed investigation and architectural design space exploration. In *Proceedings of the Field-Programmable Logic and Applications (FPL'09)*, 2009. 23, 207

[147] M. Huebner and C. Schuck and J. Becker. Elementary Block Based 2-Dimensional Dynamic and Partial Reconfiguration for Virtex-II FPGAs. In *IPDPS 2006*, 2006. 23

[148] 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. 22, 23, 26

[149] M. Murgida and A. Panella and V. Rana and M.D. Santambrogio and D. Sciuto. Fast IP-Core generation in a Partial Dynamic Reconfigurable workflow. In *VLSI-SoC 2006*, 2006. 198

[150] R. Manduchi, G. M.Cortelazzo, and G. A. Mian. Multistage sampling structure conversion of video signals. *IEEE Transactions on circuits and systems for video technology*, 3:325–340, 1993. 62

[151] F. Maraninchi and Y. Rémond. Mode-automata: About modes and states for reactive systems. In *European Symposium On Programming*, Lisbon (Portugal), Mar. 1998. Springer verlag. 81, 88, 230

[152] F. Maraninchi and Y. Rémond. Mode-automata: a new domain-specific construct for the development of safe critical systems. *Sci. Comput. Program.*, 46(3):219–254, 2003. 87, 88

[153] A. Marshall, T. Stansfield, I. Kostarnov, J. Vuillemin, and B. Hutchings. A reconfigurable arithmetic array for multimedia applications. In *Proceedings of the 1999 ACM/SIGDA seventh international symposium on Field programmable gate arrays (FPGA '99)*, pages 135–143. ACM, 1999. 15

[154] MARTES. The MARTES Project, 2009. http://www.martes-itea.org/public/info.php. 75

[155] C. Mauras. *Alpha : un langage équationnel pour la conception et la programmation d'architectures parallèles synchrones*. PhD thesis, Université de Rennes I, France, December 1989. 62, 130

[156] S. McMillan and S. Guccione. Partial run-time reconfiguration using jrtr. In *Proceedings of the The Roadmap to Reconfigurable Computing, 10th International Workshop on Field-Programmable Logic and Applications (FPL'00)*, pages 352–360. Springer-Verlag, 2000. 26

[157] P. MDE. Model-Driven Engineering. http://planetmde.org. 41, 44

[158] C. Mead and L. Conway. *Introduction to VLSI Systems*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1979. 43

[159] T. Mens and P. V. Gorp. A taxonomy of model transformation. In *Proceedings of the International Workshop on Graph and Model Transformation (GraMoT 2005)*, volume 152 of *Electronic Notes in Theoretical Computer Science*, pages 125–142, March 2006. 47

[160] M. Mernik, J. Heering, and A. M. Sloane. When and how to develop domain-specific languages. *ACM Comput. Surv.*, 37(4):316–344, 2005. 44

[161] J. Miller and J. Mukerji. Model Driven Architecture (MDA). Technical report, OMG, 2001. 40, 49

[162] P. Moenner, L. Perraudeau, P. Quinton, S. Rajopadhye, and T. Risset. Generating regular arithmetic circuits with ALPHARD. Technical report, IRISA, 1996. 129

[163] G. Moore. Cramming more components into integrated circuits. *Electronics*, 38(8):114–117, 1965. 10

[164] L. Morel. Array iterators in Lustre: From a language extension to its exploitation in validation. *EURASIP Journal on Embedded Systems*, 2007, 2007. 62

[165] A. E. Mrabti, F. Petrot, and A. Bouchhima. Extending ip-xact to support an mde based approach for soc design. In *Design, Automation & Test in Europe Conference & Exhibition (DATE '09)*, pages 586–589, 2009. 198

[166] M. Mura, L. Murillo, and M. Prevostini. Model-based design space exploration for rtes with sysml and marte. In *Forum on Specification, Verification and Design Languages (FDL 2008)*, pages 203–208, 2008. 76

[167] P. K. Murthy and E. A. Lee. Multidimensional synchronous dataflow. *IEEE Transactions on Signal Processing*, 50:3306–3309, 2002. 62

[168] N. Bergmann and J. Williams and P. Waldeck. Egret: A flexible platform for real-time reconfigurable system-on-chip. In *Proceedings of International Conference on Engineering Reconfigurable Systems and Algorithms*, pages 300–303, 2003. 25

[169] N. Dorairaj and E. Shiflet and M. Goosman. PlanAhead Software as a Platform for Partial Reconfiguration. *Xcell Journal*, 55:68–71, 2005. 84, 201

[170] K. G. Nezami, P. W. Stephens, and S. D. Walker. Handel-c implementation of early-access partial-reconfiguration for software defined radio. In *IEEE Wireless Communications and Networking Conference (WCNC'08)*, pages 1103–1108, 2008. 25

[171] O. Nierstrasz, G. Arévalo, S. Ducasse, R. Wuyts, A. Black, P. Muller, C. Zeidler, T. Genssler, and R. Born. A component model for field devices. In *Proceedings of the IFIP/ACM Working Conference on Component Deployment (CD '02)*, pages 200–209. Springer-Verlag, 2002. 37

[172] 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. 88, 89, 200

[173] Object Management Group Inc. http://www.omg.org. 49

[174] Object Management Group Inc. Model-driven architecture (mda). http://www.omg.org/mda. 40, 49

[175] Object Management Group Inc. MOF 2.0 core final adopted specification. http://www.omg.org/cgi-bin/doc?ptc/03-10-04, Oct. 2003. 45

[176] Object Management Group Inc. MOF Query / Views / Transformations. http://www.omg.org/cgi-bin/doc?ptc/2005-11-01, Nov. 2005. 48, 50, 157

[177] Object Management Group Inc. Final adopted omg sysml specification. http://www.omg.org/cgi-bin/doc?ptc/06-0504, mai 2006. 52

[178] Object Management Group Inc. Omg unified modeling language (OMG UML), superstructure, v2.1.2. http://www.omg.org/spec/UML/2.1.2/Superstructure/PDF/, Nov. 2007. VI, 51, 64, 102, 103, 104, 105, 107, 145

[179] Object Management Group Inc. Uml 2 infrastructure (final adopted specifcation). http://www.omg.org/spec/UML/2.1.2/Infrastructure/PDF/, Nov. 2007. 40, 44, 50, 102

[180] Y. Oh, H. Lee, and C. Lee. Dynamic partial reconfigurable fir filter design. In *Applied Reconfigurable Computing (ARC 2006)*, pages 30–35, 2006. 183

[181] OMG. Modeling and analysis of real-time and embedded systems (MARTE). http://www.omgmarte.org/. 53, 64, 95

[182] OMG. UML profile for Schedulability, Performance and Time (SPT), Version 1.1, 2005. 52

[183] OMG. UML profile for System on Chip (SoC), Version 1.0.1, 2006. 52

[184] OMG. M2M/Operational QVT Language, 2007. http://tiny.cc/tFGGx. 49, 74

[185] OMG. OMG Unified Modeling Language (OMG UML), Superstructure, V2.1.2, 2007. http://www.omg.org/spec/UML/2.1.2/Superstructure/PDF/. 122

[186] OMG. Portal of the Model Driven Engineering Community, 2007. http://www.planetmde.org. 29

[187] R. Ommering, F. Linden, J. Kramer, and J. Magee. The koala component model for consumer electronics software. *IEEE Computer*, 33(3):78–85, 2000. 37

[188] OpenMP Architecture Review Board. OpenMP application programme interface. http://www.openmp.org/drupal/mp-documents/spec25.pdf, May 2005. 73

[189] 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. 17, 21

[190] 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. 22, 24

[191] J. Pan, T. Mitra, and W. Wong. Configuration bitstream compression for dynamically reconfigurable fpgas. In *IEEE/ACM International Conference on Computer Aided Design (ICCAD-2004)*, pages 766–773, 2004. 27

[192] K. Paulsson, M. Hubner, and J. Becker. On-line optimization of fpga power-dissipation by exploiting run-time adaption of communication primitives. In *Proceedings of the 19th annual symposium on Integrated circuits and systems design (SBCCI'06)*, pages 173–178. ACM, 2006. 19

[193] K. Paulsson, M. Hubner, and J. Becker. Dynamic power optimization by exploiting self-reconfiguration in xilinx spartan 3-based systems. *Microprocessors & Microsystems*, 33(1):46–52, 2009. 19

[194] E. Piel. *Ordonnancement de systèmes parallèles temps-réel*. PhD thesis, Université des Sciences et Technologies de Lille (USTL), 2007. 71, 73, 111, 113, 114

[195] A. D. Pimentel. The artemis workbench for system-level performance evaluation of embedded systems. *Journal of Embedded Systems*, 1(7), 2005. 76

[196] M. Porrmann, U. Witkowski, H. Kalte, and U. Rückert. Dynamically reconfigurable hardware - a new perspective for neural network implementations. In *Proceedings of the Reconfigurable Computing Is Going Mainstream, 12th International Conference on Field-Programmable Logic and Applications (FPL'02)*, pages 1048–1057. Springer-Verlag, 2002. 130

[197] M. Prochazka, R. Ward, P. Tuma, P. Hnetynka, and J. Adamek. A uml2 profile for reusable and verifiable software components for real-time applications. In *Proceedings of DAta Systems In Aerospace (DASIA 2008), European Space Agency Report Nr. SP-665*, 2008. 37

[198] D. Project-Team. Dataparallelism for Real-Time. Activity report, INRIA, http://www.lifl.fr/DaRT/pub/public/dart.pdf, 2008. 73

[199] I. R. Quadri, P. Boulet, 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. 55, 66, 94

[200] I. R. Quadri, Y. 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. 180

[201] I. R. Quadri, A. Gamatié, P. Boulet, and J.-L. Dekeyser. Modeling of configurations for embedded system implementation in marte. In *1st workshop on Model Based Engineering for Embedded Systems Design (MBESD 2010), at DATE 2010*, 2010. 231

[202] I. R. Quadri, S. Meftali, and J.-L. Dekeyser. High level modeling of partially dynamically reconfigurable fpgas based on mde and marte. In *Reconfigurable Communication-centric SoCs (ReCoSoC'08)*, 2008. 66

[203] I. R. Quadri, S. Meftali, and J.-L. Dekeyser. Marte based modeling approach for partial dynamic reconfigurable fpgas. In *6th IEEE Workshop on Embedded Systems for Real-Time Multimedia*, 2008. 25, 58, 227

[204] 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. 91

[205] I. R. Quadri, S. Meftali, and J.-L. Dekeyser. A model based design flow for dynamic reconfigurable fpgas. *International Journal of Reconfigurable Computing*, 2009. 23, 25, 58, 227

[206] I. R. Quadri, S. Meftali, and J.-L. Dekeyser. *Dynamic Reconfigurable Network-on-Chip Design: Innovations for Computational Processing and Communication*, chapter From MARTE to Reconfigurable NoCs: A model driven design methodology. IGI-Global, 2010. 55

[207] 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. 190

[208] R. Koch and T. Pionteck and C. Albrecht and E. Maehle. An adaptive system-on-chip for network applications. In *IPDPS 2006*, 2006. 26

[209] S. Raaijmakers and S. Wong. Run-time partial reconfiguration for removal, placement and routing on the virtex-ii pro. In *International Conference on Field Programmable Logic and Applications (FPL'07)*, pages 679–683, 2007. 27

[210] W. Raslan and A. Sameh. Mapping sysml to systemc. In *Proceedings of the Field-Programmable Logic and Applications (FPL'07)*, 2007. 75

[211] E. Riccobene, P. Scandurra, A. Rosti, and S. Bocchio. A model-driven design environment for embedded systems. In *Proceedings of the 43rd annual Design Automation Conference (DAC '06)*, pages 915–918. ACM, 2006. 52, 75

[212] E. Riccobene, P. Scandurra, A. Rosti, and S. Bocchio. Model-based methodologies for pervasive and embedded software (mompes'07). In *Designing a Unified Process for Embedded Systems*, pages 77–90, 2007. 75

[213] M. Rieder, R. Steiner, C. Berthouzoz, F. Corthay, and T. Sterren. Synthesized uml, a practical approach to map uml to vhdl. pages 203–217. Springer, 2006. 131

[214] F. Rocheteau. *Extension du langage Lustre et application la conception de circuits: le langage Lustre-V4 et le système Pollux,*. PhD thesis, Institut National Polytechnique de Grenoble, Grenoble, France, 1992. 62

[215] O. Rohlik, A. Pasetti, T. Vardanega, V. Cechticky, and M. Egli. A uml2 profile for reusable and verifiable software components for real-time applications. In *International conference on Software Reuse*, 2006. 37

[216] S. Gérard. *Modélisation UML exécutable pour les systèmes embarqués de l'automobile*. PhD thesis, Université d'Evry, 2000. 76

[217] S. Mohanty and V.K. Prasanna and S. Neema and J. Davis. Rapid design space exploration of heterogeneous embedded systems using symbolic search and multi-granular simulation. In *LCTES/Scopes 2002*, 2002. 76

[218] S. Pillement and D. Chillet. High-level model of dynamically reconfigurable architectures. pages 1–7, 2009. 25, 76

[219] S. Sendall and W. Kozaczynski. Model Transformation: The Heart and Soul of Model-Driven Software Development. *IEEE Software*, 20(5):42–45, 2003. 46

[220] B. Salefski and L. Caglar. Reconfigurable computing in wireless. In *Proceedings of the 38th annual Design Automation Conference (DAC'01)*, pages 178–183. ACM, 2001. 14

[221] 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. 88

[222] D.-C. Schmidt. Model-Driven Engineering. *IEEE Computer*, 39(2):25–31, 2006. 41, 44

[223] R. Scholz. Adapting and Automating XILINX's Partial Reconfiguration Flow for Multiple Module Implementations. In *Book Chapter in Reconfigurable Computing: Architectures, Tools and Applications*. Springer verlag, 2007. 27

[224] L. Seinturier, N. Pessemier, L. Duchien, and T. Coupaye. A component model engineered with components and aspects. In *Proceedings of the 9th International SIGSOFT Symposium on Component-Based Software Engineering (CBSE'06)*, volume 4063 of Lecture Notes in Computer Science, pages 139–153. Springer, 2008. 35

[225] B. V. Selic. On the semantic foundations of standard UML 2.0. In M. Bernardo and F. Corradini, editors, *Formal Methods for the Design of Real-Time Systems, International School on Formal Methods for the Design of Computer, Communication and Software Systems, SFM 2004*, volume 3185 of *Lecture Notes in Computer Science*, Bertinoro, Italy, septembre 2004. Springer-Verlag. 51

[226] E. Senn, J. Laurent, E. Juin, and J. Diguet. Refining power consumption estimations in the component based aadl design flow. In *Forum on Specification, Verification and Design Languages (FDL 2008)*, pages 173–178, 2008. 76

[227] S. Sentilles, A. Vulgarakis, T. Bureš, J. Carlson, and I. Crnković. A component model for control-intensive distributed embedded systems. In *Proceedings of the 11th International Symposium on Component-Based Software Engineering (CBSE'08)*, pages 310–317. Springer-Verlag, 2008. 37

[228] M. Shaw, R. DeLine, and G. Zelesnik. Abstractions and implementations forarchitectural connections. In *ICCDS '96: Proceedings of the 3rd International Conference on Configurable Distributed Systems*, page 2. IEEE Computer Society, 1996. 35

[229] M. Simonot and V. Aponte. A declarative formal approach to dynamic reconfiguration. In *Proceedings of the 1st international workshop on Open component ecosystems (IWOCE'09)*, pages 1–10. ACM, 2009. 38

[230] SoCLib. SoCLib: An open platform for modelling and simulation of multi-processors system on chip, 2009. https://www.soclib.fr/trac/dev/wiki. 73

[231] J. Soula. *Principe de Compilation d'un Langage de Traitement de Signal*. PhD thesis, Université de Lille 1, December 2001. In French. 152 pages Julien Soula. 85

[232] SPEEDS! SPEculative and Exploratory Design in Systems Engineering, 2009. http://www.speeds.eu.com/. 36, 75

[233] P. Stevens. A landscape of bidirectional model transformations. In *Summer school on Generative and Transformational Techniques in Software Engineering 2007 (GTTSE'07)*, 2007. 47

[234] G. Stitt, F. Vahid, and S. Nematbakhsh. Energy savings and speedups from partitioning critical software loops to hardware in embedded systems. *ACM Transactions on Embedded Computing Systems*, 3(1):218–232, 2004. 17

[235] G. Systems. Metropolis: Design environment for heterogeneous systems. http://gigascale.org/metropolis/, 2009. 37, 76

[236] C. Szyperski. ACM Press and Addison-Wesley, New York, N.Y, 1998. 32, 38

[237] 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. 29

[238] V. Tadigotla, L. Sliger, and S. Commuri. Fpga implementation of dynamic run-time behavior reconfiguration in robots. In *Proceedings of the 2006 IEEE International Symposium on Intelligent Control*. IEEE Computer Society, 2006. 130

[239] S. Taha, A. Radermacher, S. Gerard, and J.-L. Dekeyser. Marte: Uml-based hardware design from modeling to simulation. In *Forum on Specification and Design Languages (FDL 2007)*, 2007. 76

[240] S. Taha, A. Radermacher, S. Gerard, and J.-L. Dekeyser. An open framework for detailed hardware modeling. In *IEEE proceedings SIES'2007*, 2007. 76

[241] J. Taillard, F. Guyomarc'h, and J.-L. Dekeyser. OpenMP code generation based on an Model-Driven Engineering approach. In *The 2008 High Performance Computing & Simulation Conference (HPCS 2008)*, Nicosia, Cyprus, June 2008. 66

[242] H. Tan, R. DeMara, A. Thakkar, A. Ejnioui, and J. Sattler. Complexity and performance evaluation of two partial reconfiguration interfaces on fpgas: A case study. In *Engineering of Reconfigurable Systems and Algorithms*, pages 253–256, 2006. 22

[243] H. Tardieu, A. Rochfeld, and R. Colletti. *La Méthode Merise : Principes et outils*. Editions d'Organisation, 1991. 44

[244] Tata Research Development and Design Centre. Modelmorf ¿ a model transformer. http://www.tcs-trddc.com/ModelMorf, 2009. 48

[245] I. S. Technologies). OMEGA: Correct Development of Real-Time Embedded Systems , 2009. http://www-omega.imag.fr/. 76

[246] F. Telecom. Smartqvt documentation, 2007. http://smartqvt.elibel.tm.fr/doc/index.html. 49, 74

[247] R. Tessier and W. Burleson. Reconfigurable computing for digital signal processing: A survey. *Journal of VLSI Signal Processing Systems*, 28(1/2):7–27, 2001. 129, 183

[248] W. Thies, M. Karczmarek, and S. P. Amarasinghe. Streamit: A language for streaming applications. In *CC '02: Proceedings of the 11th International Conference on Compiler Construction*, pages 179–196, London, UK, 2002. Springer-Verlag. 62

[249] T. Todman, G. Constantinides, S. Wilton, O. Mencer, W. Luk, and P. Cheung. Reconfigurable computing: architectures and design methods. *IEE Proceedings on Computers and Digital Techniques*, 152(2):193–207, 2005. 14

[250] J. Tugnait. Nonparametric bispectrum-based time delay estimation for bandlimited signals. *Electronics Letters*, 31(19):1634–1635, 1995. 182

[251] W. Vanderperren and D. Suvée. Jascoap: Adaptive programming for component based software engineering. In *3rd International Conference on Aspect-Oriented Software Development (AOSD-2004)*. ACM, 2004. 34

[252] M. von der Beeck. A comparison of statecharts variants. In *ProCoS: Proceedings of the Third International Symposium Organized Jointly with the Working Group Provably Correct Systems on Formal Techniques in Real-Time and Fault-Tolerant Systems*, pages 128–148, London, UK, 1994. Springer-Verlag. 230

[253] W. Cesario et al. Component-Based Design Approach for Multicore SoCs. *Design Automatic Conference, DAC'02*, 00:789, 2002. 76

[254] W. McUmber and B. Cheng. UML-based analysis of embedded systems using a mapping to VHDL. In *IEEE International Symposium on High Assurance Software Engineering, HASE'99*, pages 56–63, 1999. 131

[255] G. Wang and H. MacLean. Architectural Components and Object-Oriented Implementations. In *Proceedings of the 1st International Workshop on CBSE,*, 1998. 43

[256] S. Wood, D. Akehurst, W. Howells, and K. McDonald-Maier. Mapping the design of repetitive structures onto vhdl. In *International ModEasy¿07 Workshop*, 2007. 128, 130

[257] Xilinx. Two Flows for Partial Reconfiguration: Module Based or Difference Based. In *Xilinx Application Note XAPP290, Version 1.1*, 2003. 20

[258] Xilinx. Two Flows for Partial Reconfiguration: Module Based or Difference Based. In *Xilinx Application Note XAPP290, Version 1.2*, 2004. 20

[259] Xilinx. Spartan-3 advanced configuration architecture, xapp452 (version 1.0), 2005. 18

[260] Xilinx. Early Access Partial Reconfigurable Flow. 2006. http://www.xilinx.com/support/prealounge/protected/index.htm. 20, 21, 23, 196

[261] Xilinx. Early Access Partial Reconfigurable User Guide for ISE 9.2.04i. 2008. http://www.xilinx.com/support/prealounge/protected/index.htm. 21

[262] Xilinx. OPB hwicap product specification. Technical report, 2009. 23

[263] Xilinx. Virtex-VI product page. 2009. http://www.xilinx.com/products/v6s6.htm. 28

[264] L. Yan, G. Wang, and T. Chen. The input-aware dynamic adaptation of area and performance for reconfigurable accelerator. In *Proceeding of the ACM/SIGDA international symposium on Field programmable gate arrays (FPGA'09)*, pages 281–281. ACM, 2009. 130

[265] H. Yu, A. Gamatié, E. Rutten, P. Boulet, and J.-L. Dekeyser. Vers des transformations d'applications à parallélisme de données en équations synchrones. In *9ème édition de SYMPosium en Architectures nouvelles de machines(SympA'2006)*, Perpignan, France, Octobre 2006. 66

[266] L. Yung-Hsiang, L. Benini, and G. D. 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. 95

Appendix A

High level FPGA modeling in MARTE

One of the current limitations of our design flow is the absence of high level MARTE models for specifying the targeted reconfigurable architectures. As discussed during the course of this dissertation, this limitation is due to the absence of intermediate levels between the high design abstraction levels offered by Model-Driven Engineering; and the low level RTL tools responsible for simulation and final implementation of dynamically reconfigurable SoCs. However, we have proposed an initial contribution in [203, 205], as illustrated in Figure A.1, but corresponding model transformations have not been developed, due to the limitations described before. The contributions extended the MARTE profile and related metamodel, for expressing dynamic aspects at the modeling phase. Figure A.1 shows the modeling of a Xilinx Virtex II-Pro XC2VP30 FPGA on the XUP board, used in our case study. The modeling is similar to that illustrated in Figure 9.15, albeit some minute differences such as the introduction of an SRAM controller.

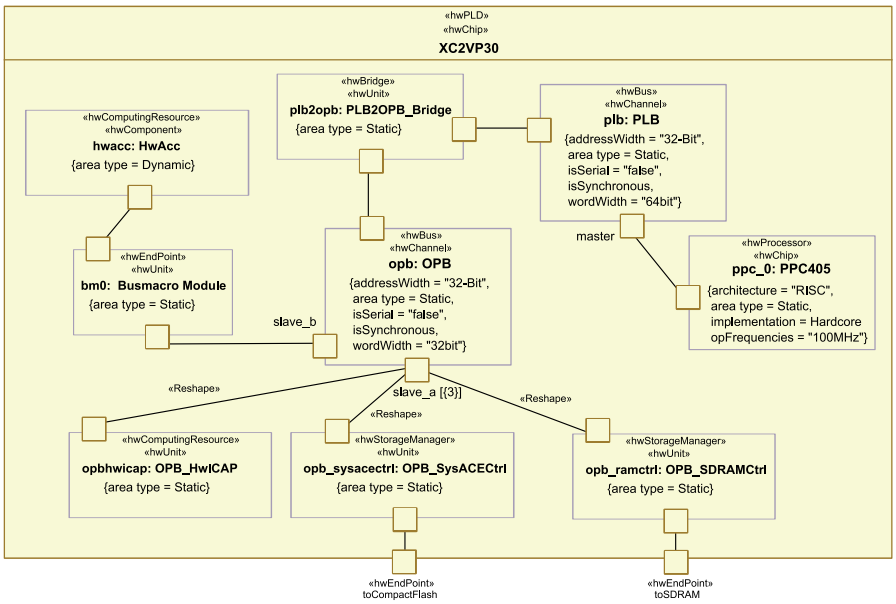


Figure A.1: Modeling of an FPGA with the MARTE profile

The high level modeling helps in creating a complete system (application and architecture), while the allocation, as illustrated in Figure A.2, permits to allocate the application onto the targeted architecture. This aspect can allow the designer different allocation choices: for example, the complete application can be placed onto a hardware accelerator, or either available hard/soft core processors present in the targeted architecture. Additionally, it is possible to separate the execution of the application: some key kernels can be executed onto hardware accelerator permitting a parallel execution, while others can be executed sequentially. The control semantics modeled in our design flow can also be allocated onto a specific processor, eliminating some of the current present ambiguities.

An intermediate solution for bridging the gap between MARTE based UML models and low level tools can be the introduction of novel model transformations. The transformations should be capable of interpreting the high level architectural models, and in turn can produce a part of input files required by the RTL tools. These files can be either in the form of some architectural description documents (such as generated by Xilinx EDK tool); for determining the structure of the architecture, i.e., instantiation of submodules and their external interfaces/ports. The RTL tools can thus take these files and generate the corresponding HDL code and software drivers for the creation of the specified architecture.

Finally, our modeling methodology can also be extended by integrating the MARTE *hwPhysical* arrangement notation that provides rectangular grid-based placement mechanisms for bridging the gap between UML diagrams and actual physical layout and topology of the targeted architecture. Unfortunately, due to the current functional limitations of the UML modeling tools, it is not possible to express this view. However, this view could be a potential additional aid to commercial PDR tools such as PlanAhead. Designers can specify the FPGA layout at the MARTE specification level, helping the RTL designer in specifying an initial layout in MARTE. At the Register Transfer Level, designers can accurately estimate if the layout is feasible and determine the number of consumed FPGA resources. Finally using these simulation/synthesis results, the high-level models can be modified resulting in an effective DSE strategy for PDR-based FPGA implementation.

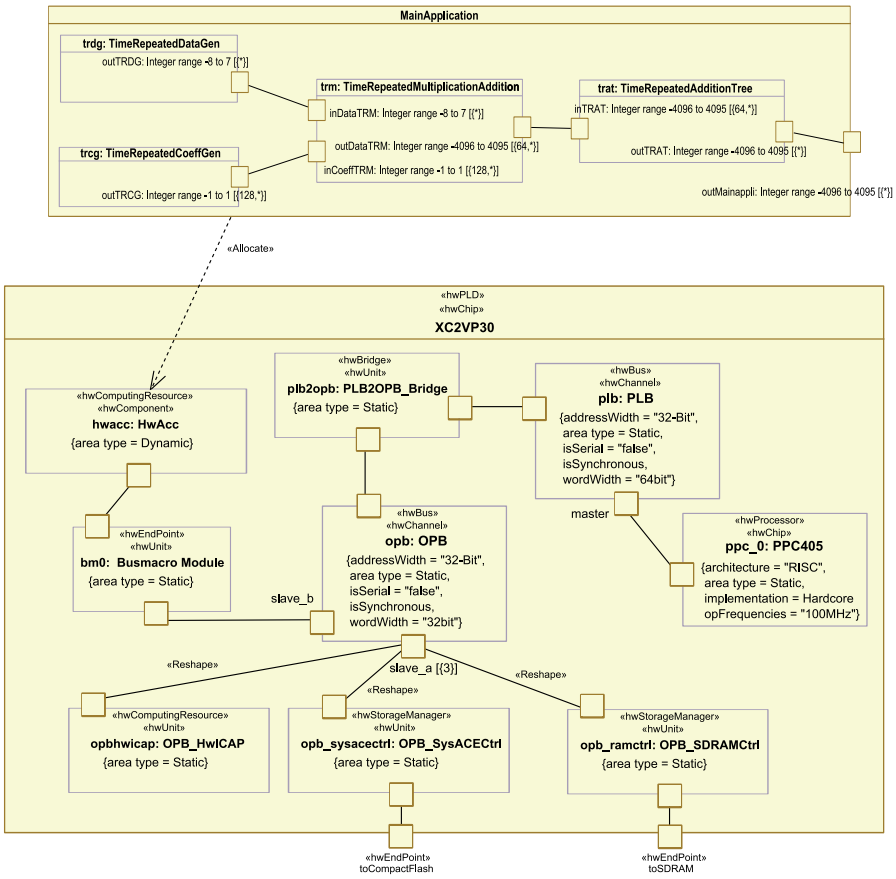


Figure A.2: Allocating the application onto the hardware accelerator present in the FPGA

Appendix B

MARTE proposal for configurations

During the writing of this dissertation, we observed that the current version of the MARTE specifications has also proposed the notion of configurations[1]. The current MARTE proposition for the modeling of embedded system configurations and their associated controllers is inspired from AADL; and also relies heavily on the usage of components and finite state machines (FSMs), as illustrated in the following examples. Figure B.1 illustrates the MARTE profile concepts related to system modes and configurations. It is evident that there is a one to one correspondence between these concepts and pure UML state machine semantics. A *mode* is related to a system *configuration*, and *mode transitions* switch between the different available configurations.

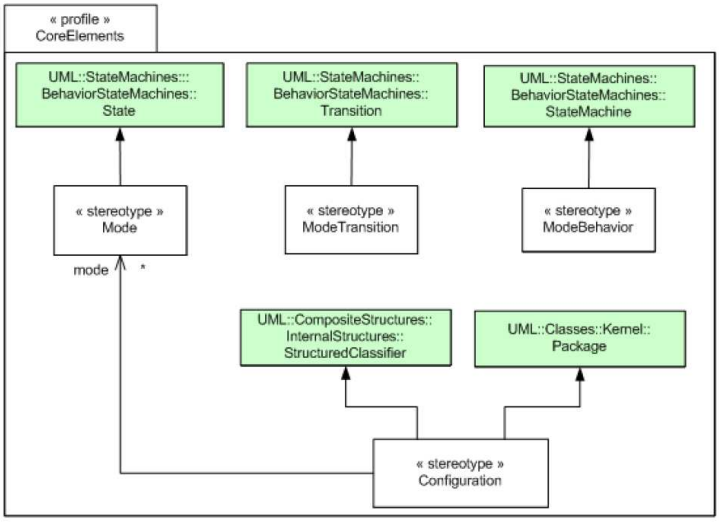


Figure B.1: Modes and Configurations in MARTE profile

Example of system configuration modeling in MARTE

Figure B.2 illustrates a component[2], with the stereotype «`configuration`» that represents a MARTE configuration for an allocation of an application functionality onto a hardware architecture. More generally, this component encapsulates a model in the form of a classifier or a package. The mode value `FullProcessorMode` indicating an active configuration is noted on the top right of the `Mode_SystemConfiguration1` component.

More concretely, the figure illustrates the mapping of the intra-part of an H.263 encoder dedicated to video processing. The software application part is composed of three main components: a DCT, a quantizer and a Huffman coding function. The application is to be allocated onto a hardware architecture:

[1] The concepts were introduced in MARTE Beta 3.0 specifications and are available in version 1.0 as well

[2] As the current specifications of the 1.0 version of the MARTE profile are not available for integration in existing UML modeling tools, we have presented an abstract proximitive illustration

composed of processors, a hardware accelerator, and memory devices. Here, the allocation expresses that all functional components are executed on the processors.

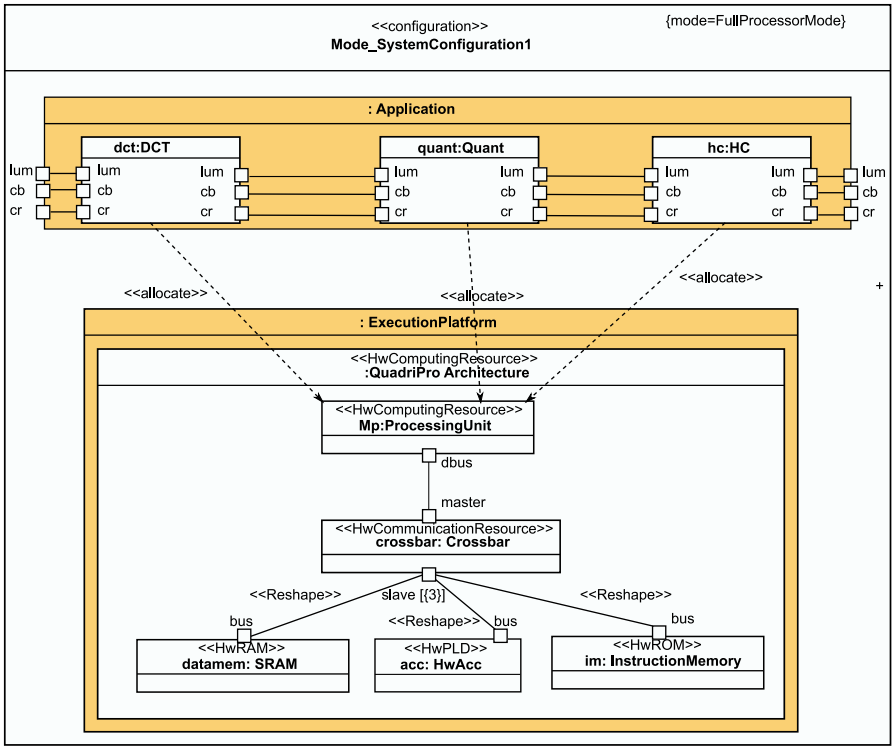


Figure B.2: Processor-based homogeneous allocation

Now, Figure B.3 depicts another software/hardware allocation scenario, according to which only the quantizer and the Huffman coding function are executed on the processors. The DCT is now executed on a dedicated hardware accelerator offering better execution performances than a processor, due to its parallel regular execution. Indeed, it is well-known that the DCT is one of the most resource-demanding parts of video encoding algorithms, and particularly of the H.263 encoder.

The way mode values are produced for selecting configurations is specified via an FSM, modeled in Figure B.4. The FSM contains two states corresponding to the illustrated configurations. Each state of the FSM is associated with some mode value specifications. An active configuration is therefore the one associated with a mode value, corresponding to the current state of the controller FSM. The transition from a configuration to another is captured via the transitions of the states present in the FSM.

Open questions

The current proposal in MARTE profile raises several interesting questions. Beyond the already identified unclarities in the semantics of UML state machines on which the configuration controller definition relies upon, there are further concerns during the interactions between a controller and its associated configurations, as discussed onwards.

Controllers can be combined at different SoC Co-Design levels to describe more complex configuration switches, as discussed in the dissertation. Typically, these controllers can also be composed in parallel. An expected behavior is therefore that all controllers make their transitions in parallel, within the same global transition. These parallel controllers may synchronize through their event occurrences, the output from one controller being an input of the other. Another interesting scenario for composing controllers is hierarchy, where a state of a controller may itself consist of another controller.

In the current proposition of MARTE, the semantics of such multi-level compositions of configuration controllers is not clearly discussed. For instance, in a hierarchical controller, how does one synchronize the states at a given level with states at the sub-levels? When a configuration gets suspended from a controller state at a sub-level, how does one manages to resume this configuration? These questions can be answered only if the semantics aspects are precised. In the literature, there are already several proposal candidates for defining such a semantics [12, 151, 252]. Also the question related to the transitions of the configuration

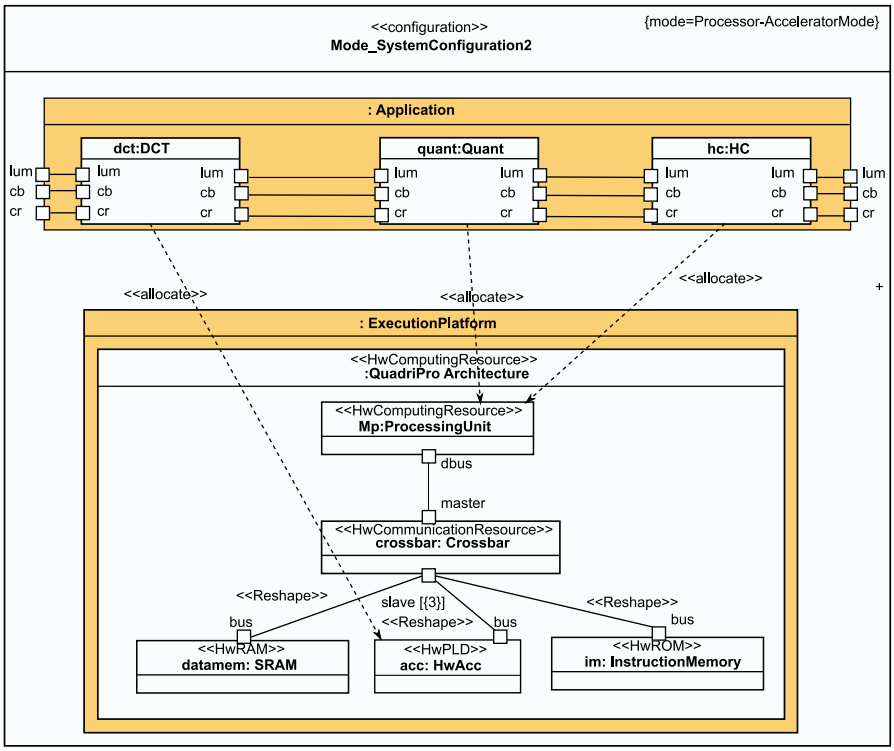


Figure B.3: Mixed processor/hardware accelerator allocation

switches becomes important, when focusing on an execution platform. In some cases, transitions may not be immediate in nature, but a change from one state to another needs a stabilization phase, for example, to preserve data flow before/after an effective switch. These issues need to be addressed via some specific control semantics.

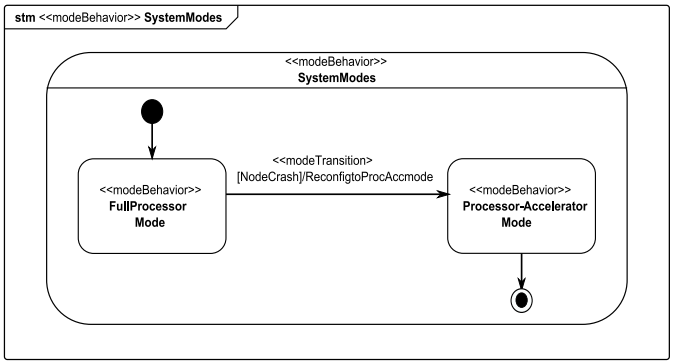


Figure B.4: Mode specification with an FSM

The mode automata based control semantics introduced in this thesis address some of these problems, but do not provide a complete solution. However, an interesting approach would be the merge of the MARTE proposal of configurations, and our introduced contributions. In [201], we have illustrated an initial effort related to this aspect, with the goal of providing future extensions.

Appendix C

Code examples

IP Code for elementary components of the DECM functionality

CoeffGen IP

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
use IEEE.numeric_bit.all;
use IEEE.numeric_std.all;
use IEEE.STD_LOGIC_ARITH.ALL;

library imran;
use imran.userlibrary.all;

ENTITY myCoeffGenVHDL IS
PORT
(
clk : IN STD_LOGIC;
reset : IN STD_LOGIC;
IPOutCoeffGen : OUT TABLE_TYPE_128_Integerrange1to1);
END myCoeffGenVHDL;

ARCHITECTURE archimyCoeffGenVHDL OF myCoeffGenVHDL IS

signal reference   : std_logic_vector(127 downto 0);

BEGIN

--the PRBS
        reference <= "01111111111000111000100111011001010111011111010100011110
10010101000001011111111010101010111110100001110100100011001011010110011110";

process(clk)
begin
  if clk'event and clk='1' then
     IPOutCoeffGen(128)<=0;
      for i in 1 to 127 loop
        if reference(128-i)='0' then
        IPOutCoeffGen(i)<=-1;
        else
        IPOutCoeffGen(i)<=1;
        end if;
      end loop;
  end if;
```

```
end process;

END archimyCoeffGenVHDL;
```

DataGen IP

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
use IEEE.numeric_bit.all;
use IEEE.numeric_std.all;
use IEEE.STD_LOGIC_ARITH.ALL;


library imran;
use imran.userlibrary.all;

ENTITY myDataGenVHDL IS
PORT
(
clk : IN STD_LOGIC;
reset : IN STD_LOGIC;
OutDataGen : OUT Integer range -8 to 7);
END myDataGenVHDL;

ARCHITECTURE archimyDataGenVHDL OF myDataGenVHDL IS

signal CounterEmission : INTEGER:=126;
signal Signal_Received : INTEGER range -8 to 7 :=0;
signal reference : std_logic_vector(127 downto 0);

begin

reference <= "01111111111000111000100111011001010111011111010100011110
10010101000001011111111010101010111101000011101001000110010110101100111110";

OutDataGen <=   Signal_Received;

--process for input signal without noise
process(reset, clk)

begin
  if reset='0' then
  Signal_Received <= 0;
  CounterEmission <= 126;

        elsif   clk'event and clk='0'   then
    if reference(CounterEmission)= '0'  then
                  Signal_Received <= 7;
                else    Signal_Received <= -8;
    end if;

          if CounterEmission > 0 then CounterEmission<= CounterEmission-1;
          else CounterEmission <= 126;
          end if;

        end if;
end process;

END archimyDataGenVHDL;
```

MultiplicationAddition IP code for DSP configuration

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
use IEEE.numeric_bit.all;
use IEEE.numeric_std.all;
use IEEE.STD_LOGIC_ARITH.ALL;


ENTITY myMultiplicationVHDL IS
PORT
(
clk : IN STD_LOGIC;
reset : IN STD_LOGIC;
IPInData1 : IN Integer range -8 to 7;
IPInData2 : IN Integer range -8 to 7;
IPInCoeff1 : IN Integer range -1 to 1;
IPInCoeff2 : IN Integer range -1 to 1;
IPOutData : OUT Integer range -4096 to 4095);
END myMultiplicationVHDL;

ARCHITECTURE archimyMultiplicationVHDL of myMultiplicationVHDL IS


signal Tempresult1     : INTEGER range  -32 to 31:=0;
signal Tempresult2     : INTEGER range  -32 to 31:=0;
signal TempFinalresult : INTEGER range -64 to 63:=0;
signal SignalConcat    : STD_LOGIC_VECTOR (6 downto 0):="0000000";
signal SignalConcat1   : STD_LOGIC_VECTOR (12 downto 0):="0000000000000";
signal minusvalue      : Integer range -4096 to 4095;

BEGIN

rt : process(reset,clk)
begin
if reset = '0' then
Tempresult1 <= 0;
Tempresult2 <= 0;
TempFinalresult <= 0;
elsif   clk'event and clk='1'   then

Tempresult1 <= IPInData1*IPInCoeff1;
Tempresult2 <= IPInData2*IPInCoeff2;
TempFinalresult <= Tempresult1 + Tempresult2;

end if;

minusvalue <= conv_integer(TempFinalresult);
IPOutData <= -minusvalue;

end process;

END archimyMultiplicationVHDL;
```

MultiplicationAddition IP code for If-then-else configuration

```

library IEEE;
use IEEE.STD_LOGIC_1164.all;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
```

```
use IEEE.numeric_bit.all;
use IEEE.numeric_std.all;
use IEEE.STD_LOGIC_ARITH.ALL;


ENTITY myMultiplicationVHDL IS
PORT
(
clk : IN STD_LOGIC;
reset : IN STD_LOGIC;
IPInData1 : IN Integer range -8 to 7;
IPInData2 : IN Integer range -8 to 7;
IPInCoeff1 : IN Integer range -1 to 1;
IPInCoeff2 : IN Integer range -1 to 1;
IPOutData : OUT Integer range -4096 to 4095);
END myMultiplicationVHDL;

ARCHITECTURE archimyMultiplicationVHDL of myMultiplicationVHDL IS

signal Tempresult1     : INTEGER range  -32 to 31:=0;
signal Tempresult2     : INTEGER range  -32 to 31:=0;
signal TempFinalresult : INTEGER range -64 to 63:=0;
signal SignalConcat    : STD_LOGIC_VECTOR (5 downto 0):="000000";
signal SignalConcat1   : STD_LOGIC_VECTOR (12 downto 0):="0000000000000";
signal minusvalue      : Integer range -4096 to 4095;

BEGIN

rt : process(reset,clk)
begin
if reset = '0' then
  Tempresult1 <= 0;
  Tempresult2 <= 0;
  TempFinalresult <= 0;
elsif   clk'event and clk='1'   then
                if IPInCoeff1 = 1 then
                  Tempresult1 <= IPInData1;
                else
                  Tempresult1 <= -IPInData1;
                end if;
                if IPInCoeff2 = 1 then
                  Tempresult2 <= IPInData2;
                else
                  Tempresult2 <= -IPInData2;
                end if;
TempFinalresult <= Tempresult1 + Tempresult2;

end if;

minusvalue <= conv_integer(TempFinalresult);
IPOutData <= -minusvalue;

end process;

END archimyMultiplicationVHDL;
```

Addition IP

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
```

```
use IEEE.numeric_bit.all;
use IEEE.numeric_std.all;
use IEEE.STD_LOGIC_ARITH.ALL;

library imran;
use imran.userlibrary.all;

ENTITY myAdditionVHDL IS
PORT
(
clk : IN STD_LOGIC;
reset : IN STD_LOGIC;
IPInData1 : IN Integer range -4096 to 4095;
IPInData2 : IN Integer range -4096 to 4095;
IPOutData : OUT Integer range -4096 to 4095);
END myAdditionVHDL;

ARCHITECTURE archimyAdditionVHDL OF myAdditionVHDL IS

signal IPOutDataTEMP : INTEGER range -4096 to 4095 :=0;
signal IPInData1TEMP : INTEGER range -4096 to 4095 :=0;
signal IPInData2TEMP : INTEGER range -4096 to 4095 :=0;

BEGIN

rt : process(reset,clk)
begin
if reset = '0' then
IPOutDataTEMP <= 0;
IPInData1TEMP <= 0;
IPInData2TEMP <= 0;
elsif   clk'event and clk='1'   then
IPInData1TEMP<=IPInData1;
IPInData2TEMP<=IPInData2;
IPOutDataTEMP <= IPInData1+IPInData2;

end if;

IPOutData<=IPOutDataTEMP;

end process;

END archimyAdditionVHDL;
```

QVTO rules for ADO Pre-computations in Input Tiler

```

mapping GCM::AssemblyConnector::toInputTilerSubConnector() : Sequence(mmHardwareacc::HW_SubConnector)
{
	init
	{

		var tiler := self.topology.oclAsType(RSM::linkTopology::Tiler);
		var repetition : RSM::shape::ShapeSpecification := null;
		var port : AssemblyConnectorEnd := null;
		var portInstance : AssemblyConnectorEnd := null;


		var paving : datatypes::IntegerMatrix := tiler.paving;
	var fitting : datatypes::IntegerMatrix := tiler.fitting;
	    var origin : datatypes::IntegerVector := tiler.origin;
	    var rep : Sequence(Integer) := Sequence{};

		var i := 0;
```

```
            result := Sequence{};
            var portEnd : Extension::AssemblyConnectorEnd := self._end[assemblyPart =
null]->asSequence()->first();
            var portInstEnd : Extension::AssemblyConnectorEnd := self._end[assemblyPart !=
null]->asSequence()->first();
            var PartShape := portInstEnd.assemblyPart.shape->asSequence();
        var ShapePort := portInstEnd.endPort.shape->asSequence();
        -- ports of the tiler
        var inputPort := portEnd.resolveone(HW_InputPort);
        var outputPort := portInstEnd.resolveone(HW_OutputPort);

        var ShapeArray := inputPort.dim.value->asSequence();

        var productAll := 0;
        var productPart := 1;
        var productPort := 1;

        var index_all : Sequence(Integer) := Sequence{};
         i := 0;
        --Product is the number of repeated instances

        while (i < PartShape.size->size())
        {
                rep += 0;
                productPart := productPart * PartShape.size->at(i+1);
                        i := i + 1;
                        index_all += 0;
        };
        i := 0;

        var shiftregister := -1;
        var length_shiftregister := 0;

        while(i < ShapePort.size->size())
        {
                productPort := productPort * ShapePort.size->at(i+1);
                i := i + 1;
                index_all += 0;
        };
        productAll := productPart*productPort;
        -- modification related to infinite dimension = -1 is converted to 1
        --for subconnector generation
        productAll :=
                if (productAll < 0)
                        then productAll * -1
                        else productAll
                endif;
                --
        i := 0;

        while (i < productAll) {
                var val := i;

                var repIndex : Sequence(Integer) := Sequence{};
                        var k := PartShape.size->size();
                        while (k>0) {
                                repIndex += index_all->at(k+ShapePort.size->size());
                                k := k - 1;
                        };

                var pattIndex : Sequence(Integer) := Sequence{};
                        k := ShapePort.size->size();
                        while (k>0) {
                                pattIndex += index_all->at(k);
                                k := k - 1;
                        };



                -- calculate tiler s
                k := 1;
                var ref : Sequence(Integer):= Sequence{};
                        while (k <= origin.vectorElem->size()) {
                                var va := origin.vectorElem->at(k);
                                var j := 1;
                                while (j <= repIndex->size()) {
                                        va := va + repIndex->at(j)*paving.matrixElem->at(j)
                                                .oclAsType(datatypes::IntegerVector).vectorElem->at(k);
                                        j := j + 1;
                                };
                                j := 1;
                                while (j <= pattIndex->size()) {
                                        va := va + pattIndex->at(j)*fitting.matrixElem->at(j)
```

```
                                                .oclAsType(datatypes::IntegerVector).vectorElem->at(k);
                                        j := j + 1;
                                };

--for the condition that the input port has an infinite dimension and that the
--subconnector source index dimension is no--t 0, meaning that data is not consumed
--at the same tick of clock and a shift register is necessary to accomodate the shifting
--window

                                if (va != 0 and ShapeArray->at(k) = -1) then {
                                        shiftregister := k;
                                        length_shiftregister := ShapePort.size->at(1);
                                } endif;
                                ref += va;
                                k := k + 1;
                        };

                var subConnectors := object mmHardwareacc::HW_SubConnector
                {
                        sourcePort := inputPort;
                        targetPort := outputPort;

                        -- compute repetition from counter
                        sourceIndex :=  object mmHardwareacc::HW_Shape
                                {
                                value += ref;
                                };
                        targetIndex := object mmHardwareacc::HW_Shape
                        {
                                value += repIndex;
                                value += pattIndex;
                        }


                --source and target port for each subconnector

                } ;

                i := i + 1;
                result += subConnectors;

                var newindex : Sequence(Integer) := Sequence{};
                        var position := 1;
                        var value := 1;

                        while (position<=ShapePort.size->size())
                                {
                                var int := index_all->at(position) + value;
                                if (int < ShapePort.size->at(position))
                                        then {
                                                newindex += int;
                                                value := 0;
                                                }
                                        else {
                                                newindex += 0;
                                                value := 1;
                                                }
                                endif;
                                position := position + 1;
                                };

                        while (position<=ShapePort.size->size()+PartShape.size->size())
                                {
                                var int := index_all->at(position) + value;
                                if (int < PartShape.size->at(position-ShapePort.size->size()))
                                        then {
                                                newindex += int;
                                                value := 0;
                                                }
                                        else {
                                                newindex += 0;
                                                value := 1;
                                                }
                                endif;
                                position := position + 1;
                                };

                        index_all := newindex;

        };

                if (shiftregister != -1) then {
                        var hw_tiler := self.resolveone(HW_InputTiler);
```

```
                        var hw_signal := object HW_Signal {
                                name := 'Signal' + inputPort.name ;

                                dim := object HW_Shape {
                                value += length_shiftregister;
                                };
                                owner := hw_tiler;
                                type := portInstEnd.endPort.resolveType();

                        };
                        var hw_delayed := object HW_delayedSubConnector {
                                name := 'DelayedSubconnector' + inputPort.name;

                                sourcePort := inputPort;
                                targetPort := hw_signal;
                                dim := object HW_Shape {
                          value += length_shiftregister;
                                };
                                sourceIndex := object HW_Shape {
                                        value += 0;
                                };
                                delay := length_shiftregister - 1;
                                tilerOwner := hw_tiler;
                                type := portInstEnd.endPort.resolveType();

                        };

                        var subconnectors := result;
                        var c := 1;
                        while (c <= subconnectors->size()) {
                                var subconn := subconnectors->at(c);
                                subconn.sourcePort := hw_signal;
                                c := c + 1;
                        };


                        hw_tiler.signal += hw_signal;

                        result += hw_delayed;
                }endif;

        }

}
```

MARTE based model driven design methodology for targeting dynamically reconfigurable FPGA based SoCs

Abstract:

The works presented in this dissertation are carried out in the context of System-on-Chip (SoC) and embedded system design, particularly dedicated to the domain of dynamic reconfiguration related to these complex systems. We present a design flow based on Model Driven Engineering (MDE) and the MARTE SoC Co-Design profile, to specify and implement these SoCs; in order to elevate the abstraction levels and to decrease system complexity.

The first contribution related to this thesis is identifying parts of dynamically reconfigurable SoCs that can be modeled at the high abstraction levels. This thesis targets the high level application models to be treated as dynamically swapple regions of a reconfigurable SoC, and proposes generic control models for managing these regions during runtime execution. While these semantics can be introduced at several high abstraction levels of a SoC Co-Design framework, we specially emphasis on fusion at the deployment level, that links intellectual properties to the modeled high level design components. Additionally, these concepts have been integrated into the MARTE metamodel to provide a suitable extension for expressing reconfigurability features at the high level modeling.

The second contribution is the proposal of an intermediate metamodel, that isolates the concepts present at the RTL. This metamodel integrates concepts responsible for the hardware execution of the modeled applications, and enriches the control semantics, resulting in creation of a dynamically reconfigurable hardware accelerator with several available implementations. Finally, using the MDE model transformations, we are able to generate HDL code equivalent to the different implementations of the reconfigurable accelerator as well as C langauge source code related to the reconfiguration controller responsible for the switching between the different implementations.

Finally, our design flow was verified successfully in a case study related to an anti-collision radar detection system. A key integral component of this system was modeled using the extended MARTE specifications and the generated code was used in the conception and implementation of a dynamically reconfigurable FPGA based SoC.

Keywords: Model-Driven Engineering, MARTE, UML, SoC Co-Design, Gaspard2, Parallel DIP applications, FPGAs, Partial Dynamic Reconfiguration, Control semantics, Mode automata.

Une méthodologie de conception dirigée par les modèles en MARTE pour cibler les systèmes sur puce basés sur les FPGA dynamiquement reconfigurables

Résumé:

Les travaux présentés dans cette thèse sont effectuées dans le cadre des Systèmes sur puce (SoC, System on Chip) et la conception de systèmes embarqués en temps réel, notamment dédiés au domaine de la reconfiguration dynamique, liés à ces systèmes complexes. Dans ce travail, nous présentons un nouveau flot de conception basé sur l'Ingénierie Dirigée par les Modèles (IDM/MDE) et le profilMARTE pour la conception conjointe du SoC, la spécification et la mise en œuvre de ces systèmes sur puce reconfigurables, afin d'élever les niveaux d'abstraction et de réduire la complexité du système.

La première contribution relative à cette thèse est l'identification des parties de systèmes sur puce reconfigurable dynamiquement qui peuvent être modélisées au niveau d'abstraction élevé. Cette thèse adapte une approche dirigée par l'application et cible les modèles d'application de haut niveau pour être traités comme des régions dynamiques des SoCs reconfigurables. Nous proposons aussi des modèles de contrôle générique pour la gestion de ces régions au cours de l'exécution en temps réel. Bien que cette sémantique puisse être introduite à différents niveaux d'abstraction d'un environnent pour la conception conjointe du SoC, nous insistons tout particulièrement sur sa fusion au niveau du déploiement, qui relie la propriété intellectuelle avec les éléments modélisés à haut niveau de conception. En outre, ces concepts ont été intégrés dans le méta-modèleMARTE et le profil correspondant afin de fournir une extension adéquate pour exprimer les caractéristiques de reconfiguration à la modélisation de haut niveau.

La seconde contribution est la proposition d'un méta-modèle intermédiaire, qui isole les concepts présents au niveau transfert de registre (RTL-Register Transfer Level). Ce méta-modèle intègre les concepts chargés de l'exécution matérielle des applications modélisées, tout en enrichissant la sémantique de contrôle, provoquant la création d'un accélérateur matériel reconfigurable dynamiquement avec plusieurs implémentations disponibles. Enfin, en utilisant les transformations de modèlesMDE et les principes correspondants, nous sommes en mesure de générer des codeHDL équivalents à différentes implémentations de l'accélérateur reconfigurable ainsi que différents codes source en langage C/C++ liés au contrôleur de reconfiguration, qui est finalement responsable de la commutation entre les différentes mplémentations.

Enfin, notre flot de conception a été vérifié avec succès dans une étude de cas liée à un système anti-radar de détection de collision. Une composante clé intégrante de ce système a été modélisée en utilisant les spécifications MARTE étendu et le code généré a été utilisé dans la conception et la mise en oeuvre d'un SoC sur un FPGA reconfigurable dynamiquement.

Mots clés: Ingénierie Dirigée par les Modèles, MARTE, UML, Conception conjointe du SoC, Gaspard2, les applications DIP parallel, FPGAs, Reconfiguration dynamique partielle, Sémantique du contrôle, Automates de modes.