# A high-level methodology for automatically generating dynamically reconfigurable systems using IP-XACT and the UML MARTE profile

Gilberto Ochoa Ruiz

SPIM

Thèse de Doctorat

école doctorale sciences pour l'ingénieur et microtechniques

UNIVERSITÉ DE BOURGOGNE

# A High-level Methodology for Automatically Generating Dynamically Reconfigurable Systems using IP-XACT and the UML MARTE Profile

Gilberto OCHOA RUIZ

SPIM

Thèse de Doctorat

école doctorale **sciences pour l'ingénieur et microtechniques**

UNIVERSITÉ DE BOURGOGNE

N° X X X

THÈSE présentée par

Gilberto OCHOA RUIZ

pour obtenir le

Grade de Docteur de

l'Université de Bourgogne

Spécialité : **Instrumentation et Informatique de l'Image**

# A High-level Methodology for Automatically Generating Dynamically Reconfigurable Systems using IP-XACT and the UML MARTE Profile

Soutenue le 14 novembre 2013 devant le Jury :

| | | |
|---|---|---|
| Frédéric ROUSSEAU | Rapporteur | Professeur à l'Université Joseph Fourier |
| Serge WEBER | Rapporteur | Professeur à l'Université de Lorraine |
| Hassan RABAH | Examinateur | Professeur à l'Université de Lorraine |
| Virginie FRESSE | Examinateur | MdC à l'Université de Saint -Etienne |
| Samy MEFTALI | Examinateur | MdC à l'Université de Lille 1 - INRIA |
| Ouassila LABBANI | Co-encadrant | MdC à l'Université de Bourgogne |
| El-Bay BOURENNANE | Directeur de thèse | Professeur à l'Université de Bourgogne |

# Acknowledgments

First, I wish to express my sincere gratitude to my advisors, Prof. E-Bay Bourennane and Dr. Ouassila Labbani-Narsis, for their guidance, ecouragement, and patience during my the last three and half years. Moreover, I would like to thank the Agence Nationale de la Recherche Scientifique for the financial support for the FAMOUS project, under which my research was carried out. I owe thanks to all people involved in this endeavour, both professors and colleague PhD students, for their constructive critiques and stimulating discussions during many of the project meetings.

Furthermore, I wish to express my deepest gratitute to the reviewers of the thesis, Prof. Serge Weber and Prof. Frédéric Rousseau for their insightful comments and constructive critiques that have led to and improved verson of thesis manuscript, and to the members of the jury Dr. Samy Meftali, Prof. Hassan Rabah and Dr. Virginie Fresse, for their remarks, questions and recommentations during my thesis defence.

I want to express my gratitude to other colleagues in the LE2I laboratory, especially to Elhillali Kerkouche, Kamel Messaoudi and Touiza Maamar, for their collaborative spirit and friendship. Moreover, I wish to express my sincere gratitude to the ensemble of the LE2I, which have supported me in many ways during my studies in France.

Furthermore, I want thank to many former colleagues and guiding lights in my professional life. First of all, I extend my gratitude to Dr. Miguel Arias-Estrada, for giving me the opportunity to create my first FPGA-based smart cameras, first at the FPGA Lab (INAOE), then in Prefixa Vision Systems. His constant inspiration has led me to pursue a scientific career in reconfigurable computing. In the same vein, I owe thanks to Prof. Fabrice Meriaudeau for his continous support during my master and PhD studies, and for putting me in contact with the RaPToM team at Le2i.

Moreover, I would like to express my utmost gratitute to the many friends that I have met during this entire time, without any particular order: Pamela Wattebled, Guillaume Spalla, Bruno Cisneros, Hayyan Alia, Sergio Martinez, Christian Mata, Fitrin Syharul, and Marcos Salazar, for providing me with an environment which was the closest that I have had to a family.

Last but not the least, I would like to thank my family: my parents and sisters, for their love and support.

# Contents

# General Introduction

# INTRODUCTION

## CONTEXT AND PROBLEMATIC

In a changing world there is an ever increasing requirement for embedded systems to be able to adapt to their environment or to meet new application demands. This adaptability is not always limited to software running on processors: many embedded applications also require that the hardware supporting them also adapts. Reconfigurable hardware is the key enabler for these systems. Hardware supported adaptation mechanisms provide a cost effective way of coping with changing environmental requirements, improvements in system features, changing protocol and data-coding standards, etc. This is in addition to providing the required flexibility to allow functionality to be defined after a system has been manufactured.

Run-time reconfiguration (RTR) has been introduced in recent years as a means of virtualizing hardware tasks in FPGA systems. However, it was not was not until the introduction of Dynamic Partial Reconfiguration (DPR) technologies by Xilinx that these systems became a reality. In DPR systems, parts of the system can be reconfigured at run-time while the other functionalities in the FPGA remain operational. This capability can provide many benefits to the systems designers, such as power and resources reduction, amongst others. However, despite the efforts by Xilinx and many industrial and academic endeavours, using DPR in very complex systems remains a daunting task. This is due, in the first place, to the complexity of the design flow, which requires an in-depth knowledge of many low level aspects of the FPGA technology. Secondly, efforts in the academia to extend the capabilities of DPR design flow have further increased the complexity of DPR SoC designs. Furthermore, the creation of SoC DPR-based systems has very specific requirements, in particular, IP reuse capabilities in which the parameterization and integration of IP cores (both DPR and non-DPR components) is performed in such a way that facilitates the design process.

The key to being able to create ever more complex systems has been the ever-increasing level of abstraction at which designers capture and reason about these systems. As an example over a decade ago, the EDA industry moved from gate level to a register-transfer level abstraction. This has been the single key factor in being able to create complex silicon systems. However in the area of system level design there have been significant efforts in developing system level languages and to define new design methodologies at this abstraction level. The reason for all this activity is simple. Register-transfer level

(RTL) hardware design is too low an abstraction level to start designing multimillion-gate systems. What is needed is a way to describe an entire system, including embedded software and analog functions, and formalize a set of constraints and requirements - all far beyond the capabilities of existing HDL-based RTL design practices.

System level languages proposals can be classified into three main classes. First, by reusing classical hardware languages such as extending Verilog to SystemVerilog. Second, by creating new languages specified for system level design. Third, by adapting software languages and methodologies as ADA or behavioural VHDL, C/C++, JAVA and UML. Actually, this third class has been of special interest to the hardware design community for the last few years. Here additional hardware related concepts are added to an existing software language in the hope that it would allow hardware to become accessible to software designers.

Among the emergent proposals in recent years, Model-Driven Engineering (MDE) has been used in co-design methodologies with relatively success in embedded systems modeling. Many of them make use of the UML profile for Modelling and Analysis of Real Time and Embedded Systems (MARTE). UML/MARTE models are used not only for communication purposes but, using model transformations, to produce concrete results such as a source code. For this purpose, MDE methodologies for SoC make use of a deployment phase in which the building blocks of the high-level models are linked to the low-level implementations that embody the related behaviour. This is basically an IP reuse problem, and in this way the components can be configured, and a synthesizable top-level implementation can be obtained. The main issue is that most MDE methodologies found in the literature make use of non-standardized IP and platform intermediate representations to move from the high-level models to the code generation phase. This fact has several implications, the first one being that UML MARTE platform models cannot easily exchangeable, limiting the interaction of MDE tools with other industrial ongoing efforts. Furthermore, the IP reuse tactics fostered by several methodologies based on customized metamodels are highly methodology-dependent, making it more difficult to adapt to different back-ends or evolve to the changing demands in the SoC industry.

However, in recent years, the SPIRIT consortium has developed the IP-XACT specification that describes a standard way documenting IP meta-data for SoC integration, and which has culminated by its adoption as an IEEE standard. Several industrial case studies have demonstrated that the adoption of IP-XACT facilitates the configuration, integration, and verification in multi-vendor SoC design flows. Additionally to the IP packaging and integration, IP-XACT can also provide an effective means for IP reuse by linking the low level implementation to their high-level counterparts in an MDE approach. IP-XACT has gathered great interest during the last recent years, with more and more tools being developed around the standard; in parallel, the MDE community have recognized the importance of being compliant with other hardware description environments, and a great deal of research has been carried out for integrating both efforts.

We believe that the combination of UML MARTE and IP-XACT can improve the applicability of the model-driven approaches to the development of FPGA-based SoC platforms, and in particular, facilitate the conception of DPR systems. This can be achieved by combining a component-based approach (for which IP-XACT was conceived) and a well fixed IP taxonomy for easily associating different blocks in the UML MARTE models to their IP-XACT counterparts that in fact abstract the HDL low-level IP implementations. However, targeting pure VHDL generation might be counter-intuitive, since it does not lend itself to further IP reuse; it is thus preferable to exploit the capabilities of IP-XACT as a standard intermediate representation (for both IP blocks and the platforms they compose) for generating the desired back-end representations. We have proceeded in this manner: we use IP-XACT for promoting IP reuse (IP reflection from the Xilinx XPS library) and design by reuse by generating the platform representation from IP-XACT. In both cases, the model transformations enable this passage seamlessly, without compromising the necessary abstraction in UML MARTE and the flow agnostic philosophy of IP-XACT regarding the target back-end.

Nevertheless, the IP-XACT standard, in its current version, does not support all the modeling aspects that approaches such as the MDE look after. Examples of these modeling endeavours are the conception of the application through Models of Computation, Schedulability or Design Space Exploration. However, the inclusion of IP-XACT in an MDE-based design flow can alleviate the interoperability issues of many current methodologies, greatly benefiting from the standard nature of the intermediate representation of the IPs and the SoC platform. However, in order to integrate IP-XACT in an MDE methodology, two aspects must be ensured. First an IP reuse mechanism through IP reflection that permits visualizing IP components in high-levels of abstraction, so a platform can be composed from these IP templates, must be set. Secondly, a mechanism for transforming the UML MARTE platform specification (stored as an XMI-IR) into its IP-XACT counterpart, and design object description has to be developed. The model transformations are the central part of any MDE methodology. A Model Transformation Tool (MTT) would permit the creation of the XMLized component descriptions, and on the other hand, the transformation from the XMI platform representation into the IP-XACT corresponding object. Furthermore, the MTT is also exploited for transforming the IP-XACT design description into the target implementation model, effectively decoupling the high-level models from the intended back-end.

The contributions of this thesis relate then to the study and implementation of a MDE-based flow based on IP-XACT that enables the modeling of the hardware artifacts of the DPR design flow (the composition of the SoC platform, along the specification of the modules to be dynamically reconfigured), and its subsequent transformation into synthesizable code, which can be used for the final implementation on FPGA devices. More details will be provided in the next section, when discussing each chapter.

## PLAN

This thesis manuscript is divided into three main parts, the first one dealing with theoretical aspects necessary to understand the contributions of this work, which fall in four main research topics, as depicted on Figure 1.

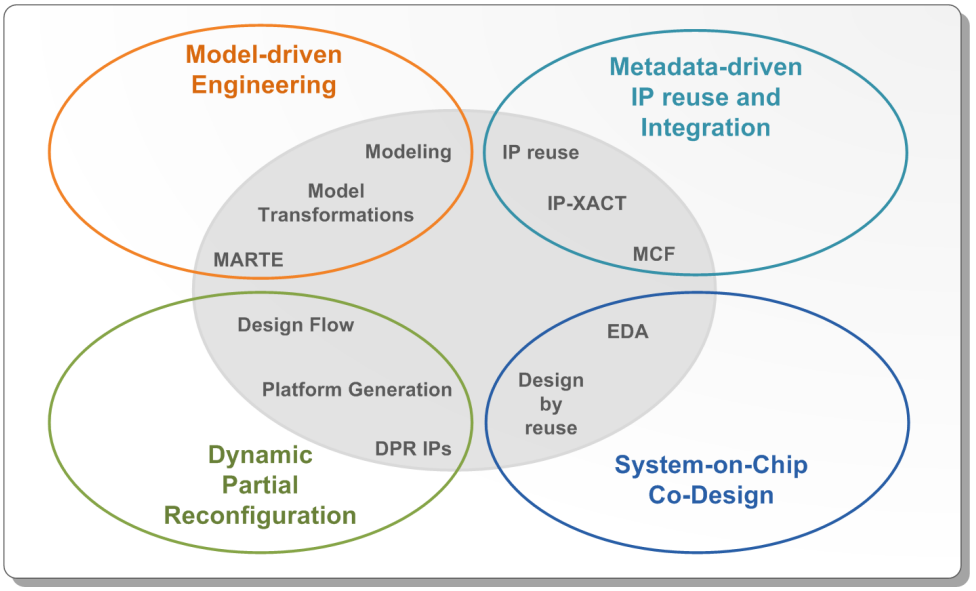


Figure 1: Main topics of this thesis work

The subjects treated in each of the chapters is described as follows:

**Chapter 1: Reconfigurable Architectures and Dynamic Partial Reconfiguration:** The first chapter of this thesis introduces the concept of reconfigurable computing and the use of reconfigurable architectures in the creation of complex System-on-Chip Systems. The architecture of the FPGA devices is described with sufficient detail so that the reader can have enough elements to understand the complexity of the subject matter. Then, the limitations of FPGA-based SoCs, which are mainly due to the design flow employed for the conception of such systems, are explored, and the concepts of Run-Time Reconfiguration and Dynamic Partial Reconfiguration are introduced, as well as the advantages of such approaches. Afterwards, the Dynamic Partial Reconfiguration design flow is analyzed in detail, as well as its inherent complexities; this analysis is used to introduce the FAMOUS framework, in which this work is circumscribed, as well as the main contributions of this thesis.

**Chapter 2: IP Reuse in MDE-based Co-design Methodologies:** This chapter explores in detail the Model-Driven Engineering paradigm, as well as the basic concepts required for understanding the main contributions of the thesis. Subsequently, one of the contributions of the thesis, in terms of providing IP reuse capabilities to the FAMOUS framework are delineated. We embark then into an in-depth description of the problems faced by the

SoC industry for promoting IP reuse and design for reuse, two ingredients that must be provided by any methodology aiming at the automatic generation of executable platform descriptions. With these basic concepts, we introduce the metadata-driven component composition framework (MCF) paradigm, and its limitations in terms of the intermediate systems representations (IR) used to pass from high-level models to the creation of synthesizable platform descriptions. We introduce then the IP-XACT standard, the chosen IR in the FAMOUS approach, and how this standard can alleviate the tool interoperability problems of many current MDE methodologies.

**Chapter 3: MDE-based methodologies for the creation of SoC**: Using the MCF paradigm as a launching pad, in this chapter we compare MDE methodologies that make use of MDE techniques for the modeling of SoC. We do not limit the comparison to FPGA targeted approaches, but to SoC design in general, dividing the comparisons into two categories: approaches that make use of UML and/or UML MARTE extensions for the modeling of complex SoC platforms (making use of custom metadata IR representations), and those who integrate IP-XACT as an intermediate system description. We situate the contributions of this work with respect to these approaches and make emphasis on the originality of this work in the closing discussion section.

In the second part of this manuscript we embark in a detailed description of the proposed MDE-based framework for the generation of DPR platforms from UML MARTE. Chapter 4 deals with the use of Xilinx Platform Studio (XPS) for creating FPGA targeted SoC platforms, and how this tool can be used for composing complex DPR systems. We describe the Xilinx-specific IP and system description models, and how they are used for storing information about different components in the tool suite. The first part of Chapter 5 presents how the IP reuse library is obtained from the XPS IPs descriptions, using IP-XACT as an intermediate representation, via model transformations. Afterwards, we deal with the composition of complex SoC platforms from the previously obtained IP library, making emphasis on the MDE paradigm, and how an executable description can be created through a compilation chain consisting on a set of model transformations.

The last chapter of this thesis manuscript presents implementation results of two different DPR platforms. The contributions introduced in the previous three chapters are used here to present full-fledged examples of how the presented methodology is used for creating such platforms in UML MARTE. Some perspectives and conclusions are presented in the final chapter.

**Chapter 4: The role of the proposed approach in the DPR Design Flow**: In this chapter we discuss how the proposed IP reuse and system composition methodology interacts with the Xilinx Dynamic Partial Reconfiguration design flow. Since we aim at a component-based approach, we make use of IPs with CoreConnect interfaces using the

Xilinx Platform Studio suite, a software tool the enable us to generate SoC platforms in a straightforward manner. We describe how the IPs are wrapped by the CoreConnect interfaces and we define an IP taxonomy for the MCF, dividing IPs into static and partially reconfigurable IPs containing or not Reconfiguration Services. Then, we discuss how XPS stores the IP and system representations, and how we plug our methodology into this back-end for promoting IP reuse and design by reuse of these components.

**Chapter 5: Proposed Methodology**: Using the IP taxonomy proposed in the previous chapter, and the knowledge of how the IPs are wrapped and used in the XPS framework. We introduce how the IP-XACT standard is exploited in our methodology for creating a vendor-agnostic IP library through model transformations from the Xilinx XPS intermediate representation. We present the main IP-XACT concepts related to interfaces and components, as well as necessary vendor extensions for making this transformation possible. Then, we introduce an IP deployment metamodel that enables us to obtain a MARTE representation of the IPs contained in the IP-XACT library. In this manner, we obtain a multi-level library through metadata reflection, which can be used for composing a platform at high levels of abstraction.

Once the multi-level component IP library has been created, it can be used for composing a SoC DPR platform. The deployment IP extensions introduced in the first part of the chapter are used in the second part of the chapter to describe how an UML MARTE platform description is transformed into an IP-XACT dessign, through model transformations. The concept of IP-XACT design description is analyzed in detail, as well as its purpose in a system generation flow. This object is employed as an introspective description, which is used for parsing the instantiated components for propagating constraints, and through a second model transformation, to automatically generate the XPS platform description

**Chapter 6: Case Study**: In the last chapter, we present two case studies to demonstrate how the proposed IP reuse and system composition framework are used for creating a DPR platforms in UML MARTE. And then how through model transformations, a synthesizable platform description can be obtained along the Partially Reconfigurable Modules to be mapped into the platform at run-time. We present synthesis results and more importantly, a discussion on how the proposed approach eases the conception of DPR systems at the entry level phase of the Xilinx design flow. Some metrics on the required design effort are presented, along a discussion on how MDE methodologies can automate the many complex tool interactions in the flow and furthermore, abstract the technologic dependent aspects of the flow to users working at a high-level of abstraction.

# I

# Context and Problematic

# 1

# Reconfigurable Architectures and Dynamic Partial Reconfiguration

## Contents

## 1.1/ INTRODUCTION

Reconfigurable computing systems use FPGAs or other programmable hardware to implement complex algorithms execution by mapping computationally-intensive calculations to the reconfigurable substrate [1].Traditionally, there have been two primary methods in conventional computing for the implementation of algorithms into hardware. The first is to use Application Specific Integrated Circuits (ASIC). They are designed to specifically perform a given task, and thus they are very fast and efficient; however, the circuit cannot be altered or updated after fabrication, forcing to redesign and fabricate a new platform if a change or upgrade is necessary, a very expensive process (presenting a high Non-Recursive Engineering cost, NRE), as depicted on Figure 1.1 a). Then, the use of ASICs is only justified in consumer electronics that are characterized by very short time to market, and high volumes [2].

The second approach is to use software-programmed microprocessors (and by extension, DSPs), a far more flexible solution. Processors execute a set of instructions to perform a computation. By changing the software instructions, the functionality of the system is altered without changing the hardware. However, the downside of this flexibility is that the performance can suffer, if not in clock speed then in work rate, and is far below that of an ASIC [3] and they suffer from the sequential execution constraint, making them not suitable for many applications in which real-time performances are necessary, as depicted on Figure 1.1 b).

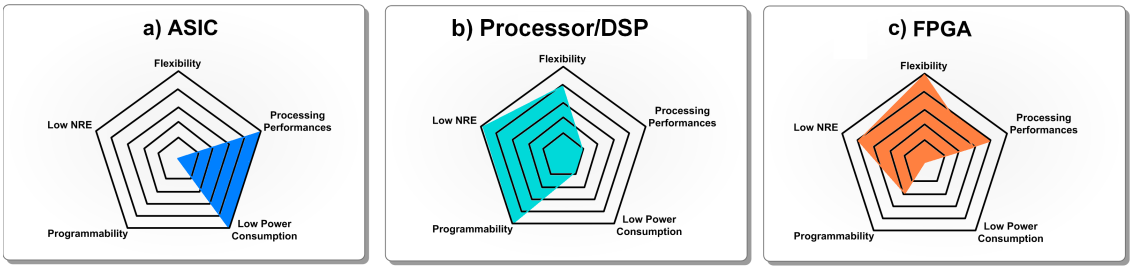


Figure 1.1: Comparison between different technologies used in SoC

Reconfigurable computing, making use of FPGA devices, is intended to fill the gap between hardware and software approaches, achieving potentially much higher performance than microprocessors, while maintaining a higher level of flexibility than ASICs [4], as depicted on Figure 1.1 c). Reconfigurable devices, including Field-Programmable Gate Arrays (FPGAs), contain an array of computational elements whose functionality is determined through multiple programmable configuration bits. These elements, sometimes known as logic blocks, are connected using a set of routing resources that are also programmable. In this way, custom digital circuits can be mapped to the reconfigurable hardware by computing the logic functions of the algorithm within the logic blocks, and using the configurable routing to interconnect the blocks to create the necessary function [5].

However, FPGA systems suffer the handicap of having a high-power consumption and a very low programmability, a factor that has limited their adoption, as depicted on the parametric graph.

In recent years, the research community has carried many endeavours in the adaptivity of electronic systems to environmental conditions and inner system status, which in theory can enable the processing of different applications in time slots on a single FPGA. This adaptivity on-demand, related to non-predictable requirements from the user or the environment implies the optimization of power dissipation and performance at run-time by providing on-chip computation capacities [6]. Traditional FPGA systems cannot provide these capacities due the nature of the design flows in which they inserted; most FPGAs are currently used as drop-in ASIC replacements or in many cases, as a platform for developing Systems-on-Chip. In order to provide devices at a competitive cost, most FPGA vendors forgo innovative configuration architectures in favour of simpler single-context designs, which the community has found somewhat limiting for implementing full-fledged run-time reconfigurable systems. However, in recent years, Xilinx and other FPGA vendors have introduced the concept of Dynamic Partial Reconfiguration (DPR), which enables to adapt small sections of the hardware platform on run-time, without halting the system functionality. Using DPR, systems behaviour can be tuned for improving its power consumption, ameliorating one of the handicaps mentioned above.

However, and despite the efforts by Xilinx and many industrial and academic endeavours, using DPR in very complex systems remains a daunting task. This is due, in the first place, to the complexity of the design flow, which requires an in-depth knowledge of many low level aspects of the FPGA technology. Secondly, efforts in the academia to extend the capabilities of DPR design flow have further increased the complexity of such systems. Nevertheless, the creation of SoC DPR-based systems could alleviate the programmability issues traditionally associated with FPGA-based systems, a trend that is becoming widespread, with the introduction of Extensible Processor Platforms. Such systems have very specific requirements, in particular, IP reuse and design by reuse capabilities must be provided, in which the parameterization and integration of IP cores (both DPR and non-DPR components) can performed in such a way that facilitates the design process.

The purpose of this chapter is to provide the reader with an in-depth knowledge of FPGAs architectures and how they are used for implementing complex embedded systems. The adaptivity limitations discussed above are then analyzed in detail, and the concept of run-time reconfiguration, as well as its advantages, is then described. Since one of the main goals of the FAMOUS project, and of this thesis work in particular, is facilitating the conception of DPR systems, as well as abstracting the burden of the design flow and the associated tools, these subjects are both described thoroughly. At the end of the chapter, the FAMOUS framework for facilitating the modeling and generation of DPR systems is introduced, as well as the main goals of this work.

## 1.2/ Reconfigurable Architectures

Before discussing the detailed aspects of dynamic partial reconfiguration, we will describe the architecture of FPGA devices, since DPR cannot be understood without a minimum of knowledge of the underlying FPGA resources (Configurable Logic Blocks, programmable interconnection and routing, and finally, heterogeneous blocks). More detailed surveys of FPGA architectures can be found elsewhere [7, 8] whereas schemes for configuration can be found in the manuals of the manufacturers.

### 1.2.1/ Logic Resources, Routing and Interconnections

An FPGA device is an integrated circuit with a central array of logic blocks that can be connected through a configurable interconnect routing matrix. Around the periphery of the logic array is a ring of I/O blocks that can be configured to support different interface standards. This flexible architecture can be used to implement a wide range of synchronous and combinatorial digital logic functions. It is common to categorize FPGA platforms as either fine grained or coarse grained. This is due to the fact that their underlying fabric predominantly consists of large numbers of relatively simple programmable logic block islands embedded in a sea of programmable interconnect, as depicted on Figure 1.2 a)

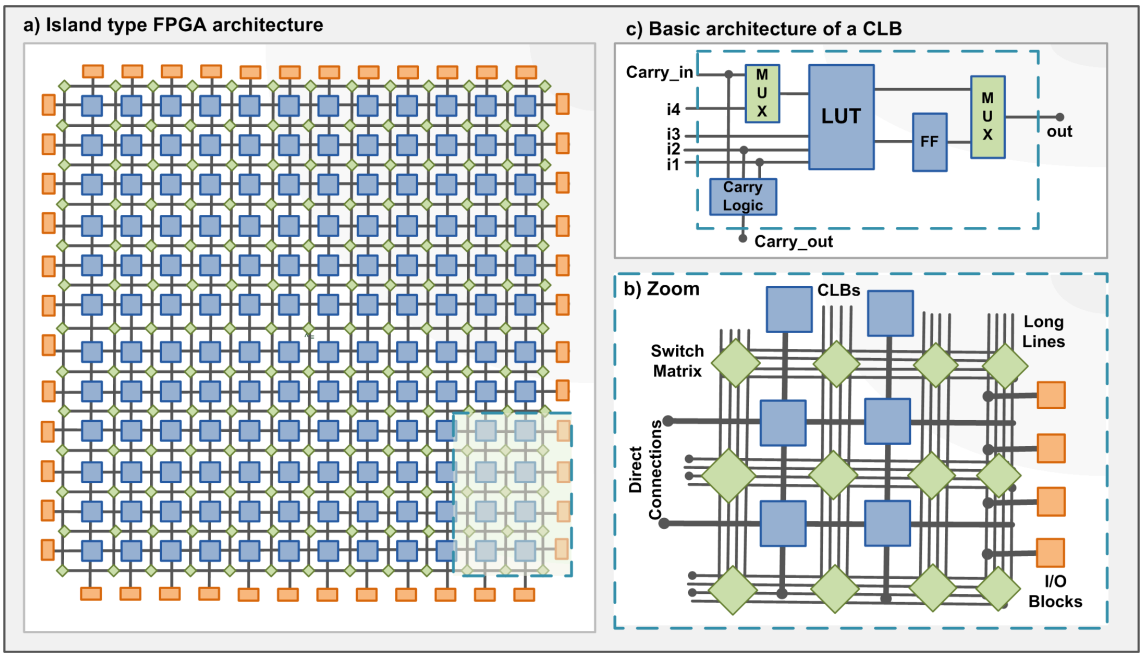


Figure 1.2: Generic FPGA hardware architecture

After several decades of research, it has been established that the best function block for a standard FPGA is a Logic Block built around a lookup table (LUT). An N-input LUT is basically a memory that, when programmed appropriately, can compute any function

of up to N inputs. This flexibility, with relatively simple routing requirements (each input needs only to be routed to a single multiplexer control input) turns out to be very powerful for logic implementation. The typical FPGA has a logic block with one or more N-input LUT(s), optional D flip-flops (FF), and some form of fast carry logic, as depicted in Figure 1.2 b). For instance, Xilinx has introduced 6-input LUT(s) in their Virtex 6, Virtex 7 and Spartan 6 FPGA families. The LUTs allow any function to be implemented, providing generic logic. The flip-flop can be used for pipelining, implementing registers, state-holding functions for finite state machines, or any other situation where clocking is required. The fast carry logic is used to speed up carry-based computations, such as addition, parity, wide AND operations, and other functions. In the case of a fine-grained architecture, each logic block can be used to implement only a very simple function. For example, it might be possible to configure he block to act as any 3-input function, such as a primitive logic gate (AND, OR, NAND, etc.) or a storage element (D-type flip-flop, D-type latch, etc.). In the case of a coarse-grained architecture, each logic block contains a relatively large amount of logic compared to their fine-grained counterparts.

Just as there has been a great deal of research in FPGA logic block architectures, there has been equally as much investigation into interconnect structures. As logic blocks have basically been standardized on LUT-based structures, routing resources have become primarily island-style, with logic surrounded by general routing channels. Most FPGA architectures organize their routing structures as a relatively smooth sea of routing resources, allowing fast and efficient communication along the rows and columns of logic blocks. As shown on Figure 1.2 c), the logic blocks are embedded in a general routing structure, with input and output signals attaching to the routing fabric through connection blocks. The connection blocks provide programmable multiplexers, selecting which of the signals in a given routing channel will be connected to the logic block IO terminals. The Switch Matrices (or Routing Switches) are also programmable, and allow connections between the horizontal and vertical routing resources to permit the signals to change their routing direction. In this manner, relatively arbitrary interconnections can be achieved between logic blocks in the system.

The ring of I/O banks surrounding the array of CLBs is used to interface the FPGA device to external components. Traditionally, the ring of I/O banks is either staggered or in-line around the FPGA device. The difference between staggered and in-line I/O is just as the names describe. A trade-off must be made architecturally between the number of available signal pins and the amount of resources implemented within the device. I/O block (IOB) is a common term used to describe an I/O structure, although other names may also be used. An IOB includes input and output registers, control signals, multiplexers and clock signals. The signals routed through the I/O block can be registered or unregistered. In order to interface to different types of logic, an FPGA device IOB must support multiple IO interface standards. Both single-ended (e.g. PCI, LVTTL) and differential operational modes (i.e. LVDS) are typically supported.

### 1.2.2/ HETEROGENEOUS PLATFORMS

In order to provide greater performance or flexibility in computation, FPGA vendors have gradually incorporated more functionalities into their programmable chips [9]. These functionalities are embedded into the circuit logic resources, producing a heterogeneous structure, where the capabilities of the logic cells are not the same throughout the system. This approach has become a necessity due to the widespread adoption of FPGAs for implementing complex platforms, such as Systems on Chip, where multiple processors, communication standards and memory controllers are required.

Furthermore, DSP applications (such as image and video processing) often require of complex functions such adders and multipliers, along complex and highly parametrisable DSP functions (DSP Blocks). In both scenarios, such modules are difficult to implement in an optimal manner using generic FPGA logic resources; therefore, FPGA vendors have introduced such complex blocks in the form of hardwired functions. These blocks are designed to be as efficient as possible in terms of power consumption, silicon real estate, and performance. Each FPGA family features different combinations of such blocks, together with various quantities of programmable logic blocks, as depicted on Figure 1.3

Another type of heterogeneous logic structures are memory blocks, known as Block-RAMs, which are scattered throughout the reconfigurable hardware. These blocks enable the storage of frequently used data and variables, and permit a quick access to these values due to the proximity of the memory inside the FPGA to the logic blocks that access it [3]. These blocks can also be used for a variety of purposes, such as implementing standard single- or dual-port RAMs, FIFO functions, state machines, and so forth.

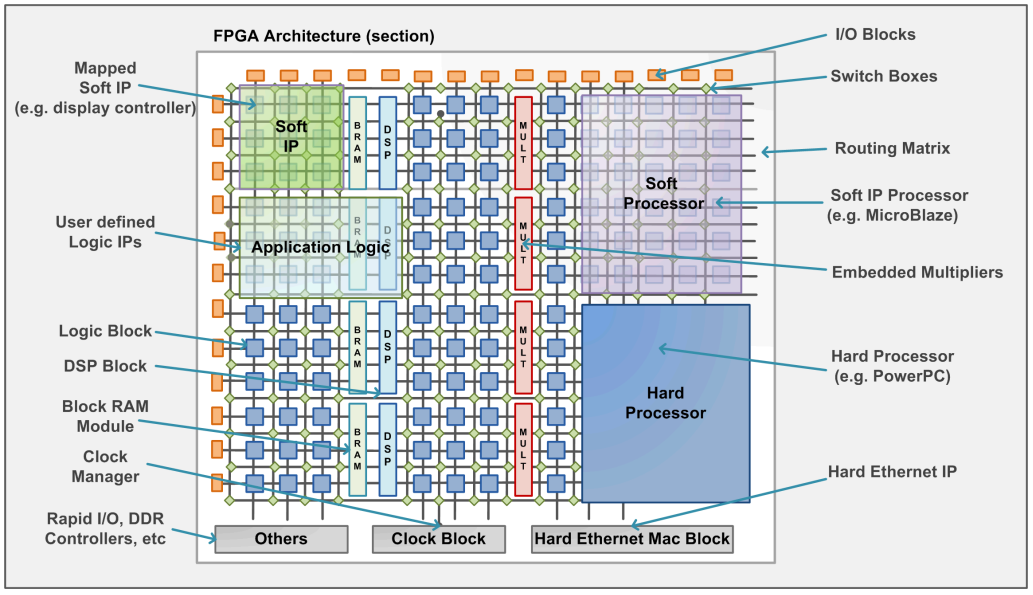


Figure 1.3: Heterogeneous FPGA platform, depicting general configurable resources, hard blocks and eventual soft IPs

Another major trend, briefly introduced in the previous section, its the introduction of complete microprocessor solutions into the FPGA fabric, known as hard processors [10]. Similarly, the so called soft core processors (i.e. components that made use of the FPGA logic resources for their implementation) have been designed and employed in a variety of applications, particularly in tasks that cannot be parallelized or in situations where the control of the system is better suited for a processor solution. Using embedded processors brings many advantages, particularly the reduction in chip count, the number of tracks, signal integrity issues, and a decrease on the board complexity.

At the other end of the spectrum, soft IP refers to a source-level library of high-level functions that can be included in the design. These functions are typically represented using a hardware description language, or HDL, such as Verilog or VHDL at the register transfer level (RTL) of abstraction. Any soft IP functions the designers decide to use are incorporated into the main body of the design which is also specified in RTL and subsequently synthesized down into a group of programmable logic blocks (possibly combined with some hard IP blocks like multipliers, DPR and BRAMs) and then mapped onto the FPGA. For instance, Figure 1.3 depicts two scenarios for the use of Soft IPs: in the first case, the function can be an functionality provided by FPGA vendors, such as the TFT interface; a second scenario is that a custom implementation of an IP is created by the user for a given application, and which can be implemented using a set of CLBs, DSP and Block RAM modules.

As performance and throughput become increasingly important, modern FPGA devices offer support for various high-speed communication standards. This is due to the high-performance of FPGA in accelerating complex computations: data needs to be transmitted rapidly and efficiently to other nodes of the system, while preventing the FPGA to become the bottleneck of the platform. Examples of these standards are Infiniband, PCI Express, RapidIO, and 10-gigabit internet, among others. In a similar manner, vendors offer optimized hardwired cores for a variety of solutions; examples of these cores are DDR, DDR2 and DDR3 controllers, MAC controllers, and LVDS Transmitters. For a more detailed description in these technologies the reader is directed to the vendors websites [11] or to excellent compendiums in the literature [5].

Finally, the primary FPGA element for handling, managing and adjusting FPGA local and system clocks is the Clock block. Input clocks, from external sources and connected to specialized FPGA pins can then be used to drive a special hard-wired function (block) known as clock managers, which are programmed to generate a number of daughter clocks, then fed to different components on the platform. In this manner, subsystems running at different frequencies can be created, either to work independently or in conjunction, adding for the heterogeneity of the target applications compared to both ASICs and processor-based solutions.

## 1.3/ USE OF FPGAS IN SOC SYSTEMS AND THEIR LIMITATIONS

### 1.3.1/ TRADITIONAL FPGA DESIGN FLOW

The standard FPGA design flow (in this case, Xilinx), as depicted on Figure 1.4 transforms an FPGA design description into a configurable bitstream. Typically, the FPGA design description is written in Hardware Description Language, like Verilog, VHDL, System C or Handel C. More complex designs often make use of system integration tools to construct the initial design description from libraries of intellectual property (IP) components, which are used along user defined functions to create System-on-Chip applications. Each pass through the standard design flow is independent of subsequent passes. The bitstream produced at the end of the flow is dependent only of the initial design specification and it is used to configure the entire device.

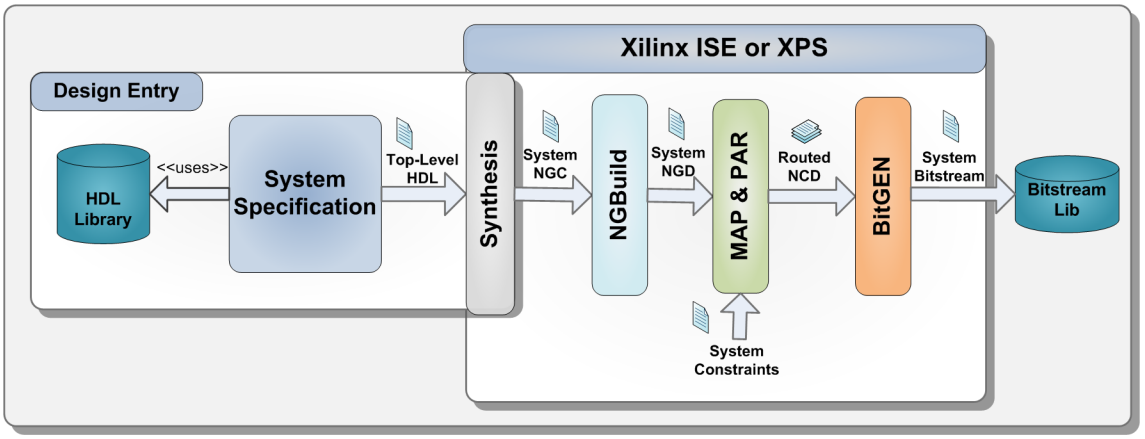


Figure 1.4: Traditional FPGA design flow

An in-depth discussion of each of these steps is outside of this scope of this work given their complexity, but the reader is directed to the manufacturer IO Command Line Tools User Guide(formerly known as the Development System Reference Guide) [12] for more details. Nevertheless, it is important to understand the basic principles in order to provide some insight into the methodology and thus each of the steps is briefly discussed as follows.

- **1.Design Entry:** In this stage of the design flow, the design is created using a FPGA-vendor supported schematic editor, text-based entry (HDL), or both (i.e. Xilinx ISE). In a similar way, the platform can be based in a combination of embedded processor, peripherals and user created logic; in such scenario, the design can be created using a specialized tool, such as Xilinx Platform Studio (XPS) [13].

  These tools enable the designer to create a design in a user-friendly environment, while the design effort to obtain the VHDL top-level description is greatly reduced.

- **2.Design Implementation:** Usually, the design specification is done regardless of the target implementation FPGA family. However, if certain features from a specific FPGA are necessary, then these constrains are taken into account from the very beginning; nonetheless, at the synthesis phase, the tool support targets a particular FPGA device. Then, a conversion of the logical design file format, such as EDIF, that was created in the design entry stage is translated to physical file format. The physical information is contained in the NGC file for FPGAs.

- **3.NGDBuild:** This stage performs all the steps necessary to read a netlist file in NGC or EDIF format and creates a NGD file describing the logical design (a logical design in terms of logic elements such as AND gates, OR gates, decoders, flip-flops, and RAMs). The NGD file resulting from an NGDBuild run contains both a logical description of the design reduced to Xilinx NGD (Native Generic Database) primitives and a description in terms of the original hierarchy expressed in the input netlist. The output NGD file can be mapped to the desired device family.

- **4. Map:** The mapping algorithms translate a logical design into a description that can be implemented into a specific FPGA device. The input to mapping is an NGD file, which contains a logical description of the design in terms of both the hierarchical components used to develop the design and the lower level Xilinx primitives, and any number of NMC (macro library) files, each of which contains the definition of a physical macro.

  MAP first performs a logical DRC (Design Rule Check) on the design in the NGD file. MAP then maps the logic to the components (logic cells, I/O cells, and other components) in the target Xilinx FPGA. The output design is an NCD (Native Circuit Description) file, a physical representation of the design mapped to the components in the Xilinx FPGA. The NCD file can then be placed and routed.

- **5.PAR:** After a design has undergone the necessary translation to bring it into the NCD (Circuit Description) format, it is ready for placement and routing. This phase is done by PAR (Xilinx's Place and Route program). PAR takes an NCD file, places and routes the design, and outputs an NCD file which is used by the bitstream generator (BitGen).

- **6.Bitstream Generation:** Produces a bitstream for Xilinx device configuration. After the design has been completely routed, it is necessary to configure the device so that it can execute the desired function. This is done with BitGen, Xilinx bitstream generation program. BitGen takes a fully routed NCD file and produces a configuration bitstream. The bit file contains all of the configuration information from the NCD defining the internal logic and interconnections of the FPGA. The binary data in the BIT file can then be downloaded into the FPGA memory.

### 1.3.2/ LIMITATIONS OF TRADITIONAL FPGA DESIGNS FLOW

Despite of the success of FPGAs in several domains, many system architects have found the traditional FPGA implementations too limiting [14], since once the design has been implemented in the circuit, any change would require undergoing the entire specification stages on Figure 1.4. This fact poses a problem in applications where a higher adaptivity on the application would be desirable, due to the manner in which the FPGA is programmed. In fact, the bitstream obtained from the design flow is charged onto the FPGA at power-up; this is due to technological reasons: the FPGAs are based on SRAM cells; thus, once the device is switched off, the totality of its functionality is lost. As mentioned previously, FPGAs are the preferred solution in applications where fabricating millions of ASICs is not an option; they also provide a bigger flexibility, since they can be re-programmed on field (hence the Field Programmability in the name), allowing for updates to be made, just as with processor-based applications, which usually only require a change in the firmware to perform a new mission.

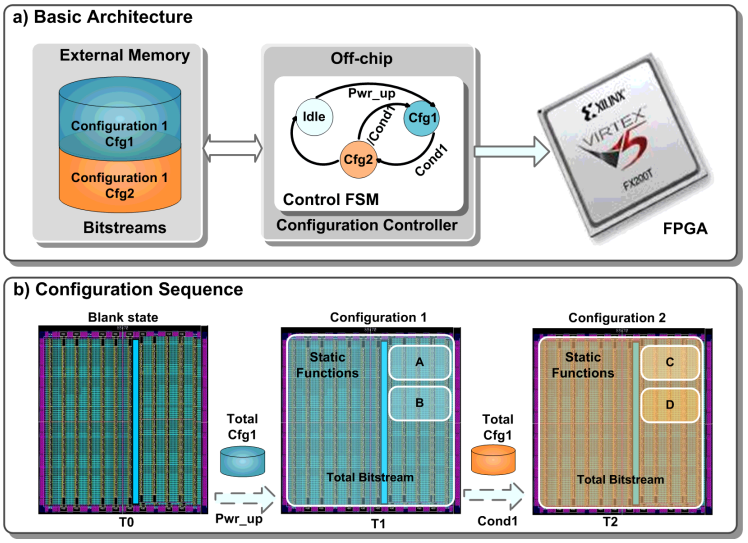


Figure 1.5: Run-time reconfiguration in a non-DPR design

Nevertheless, if the application needs to be changed at-run time [3] (some of these scenarios will be described as follows), the application would require to stop the FPGA and reconfigure it in its totality, as depicted on Figure 1.5. Furthermore, this process traditionally requires the use of an external processor, which is in charge of retrieving the configuration bitstreams from non-volatile memory using a relatively simple state machine [15]. Using an external processor increases the chip count and thus the cost of such a system, but a second problem arises: storing the totality of the bitstream for each configuration (even if the differences between each implementation is very small, as depicted on Figure 1.5 b), where only functions A and C are replaced by modules D and E) leads to increased memory storage requirements, further augmenting the cost of the system. Last, but not less important, is the time required for the system to be reconfigured; as the totality of

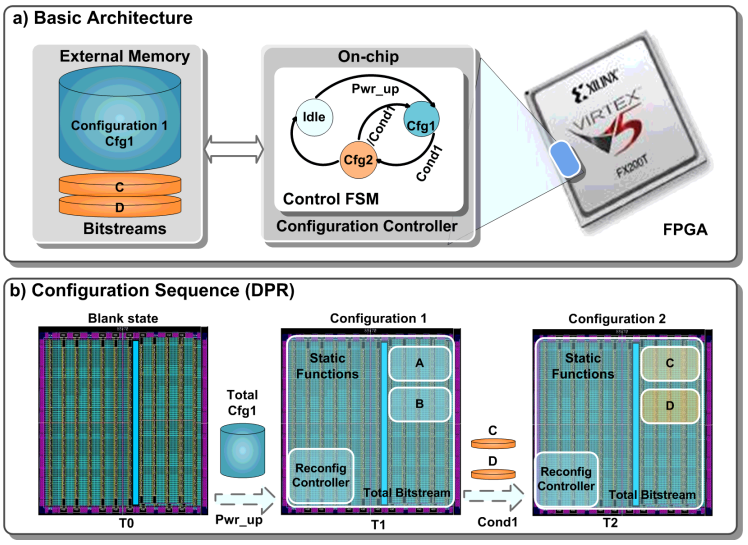


Figure 1.6: Run-time dynamic partial reconfiguration

the FPGA needs to be modified, a non-negligible amount of time (in the order of several seconds for large FPGAs) is lost to perform the reconfiguration process [16] [17], without even taking into account the access to the bitstreams from the non-volatile memory [2].

#### 1.3.2.1/ Run-time partial reconfiguration

In the last two decades, the concept of runtime reconfiguration (RTR) was introduced to describe systems capable of swapping in and out different configurations of the reconfigurable hardware as they are needed during the application execution [18] . However, the nature of the traditional design flows and other technological underpinnings had prevented the adoption of RTR techniques, since slow-downs in system performance or its viability in critical systems had made their implementation downright prohibitive. In recent years, Dynamic Partial Reconfiguration (DPR) has been introduced as a method to add flexibility to FPGA-based solutions. Through the use of DPR, a module of the system can be substituted by other functionality or simply taken out the reconfigurable fabric on demand, as depicted on Figure 1.6, without halting the device to perform the desired modification, and thus without impacting the execution of the system.

This is achieved by separating the logic into static and dynamically reconfigurable areas; this means that a section of the FPGA remains operational during the totality of the applications execution, whilst small areas of the FPGA can be reconfigured on the fly. For instance, our example shows a system containing a microprocessor and some peripherals as part of the static logic, whereas two modules comprise the dynamic part of the system. Each of the dynamic areas is reconfigured independently, by charging solely the reconfiguration information (know an as partial bitstream) required to modify its functionality. Apart from providing an increased flexibility to the application, DPR has many intrinsic

advantages. For instance, the reconfiguration time is diminished, since only a fraction of the total configuration information is retrieved from external memory. Moreover, these partial bitstreams are much smaller than a complete bitstream, which leads to a reduced external memory footprint, as depicted on Figure 1.6 b). The use of a processor (as in the previous case) is also necessary, but as it can be embedded along the rest of the static logic (using and on-chip processor such as the MicroBlaze), reducing the chip count and the complexity of the system.

#### 1.3.2.2/ Benefits of Run-time Partial Reconfiguration

Partial Reconfiguration allows using more hardware than that physically present in the FPGA by virtualizing hardware tasks and mapping them on-demand. This capability can be exploited to reduce the size of the FPGA and its overall power consumption. This also permits to execute an algorithm with an optimized implementation depending on its parameters and data set. Furthermore, upon the usual speed and power research goals, DPR offers system-level advantages for professional electronics [19] . Some of the scenarios in which DPR has proven successful are listed as follows.

- **(a) Hardware virtualization.** Frequently, the areas of a program that can be accelerated through the use of reconfigurable hardware are too numerous or complex to be loaded simultaneously onto the available hardware. For these cases, it might be beneficial to swap different configurations in and out of the FPGA as they are needed. Since this mapping is performed on-demand, the system can adapt dynamically, effectively promoting software-like component virtualization.

- **(b) Logic Resources Utilization and Power Reduction** A system might need to contain multiple implementations of a module that performs a similar task, but which are mutually exclusive. These multiplicity reduces the amount of logic that can be accommodated into a single device; by being able to reduce the amount of logic (and the required static power consumption) that an application uses at any given time, more functionalities can be integrated into a smaller FPGA.

- **(c) Survivability.** There is a great deal of research in the areas of evolvable hardware and fault tolerant systems; in the second case, the system could operate in a degraded mode when a part of the system is damaged. This can be achieved by moving the function initially mapped to a damaged area of the FPGA to another.

- **(d) Mission change.** A system can be reconfigured for a new mission without halting its functionality. This capability enables for adding functionalities that were not specified at the beginning, or in many instances, updates that can ameliorate the application performance.

## 1.4/ XILINX PARTITION PARTIAL RECONFIGURATION DESIGN FLOW

In the previous sections we have discussed the limitations of traditional FPGA designs, which are constrained by the manner in which the original system specification is mapped, placed and routed on the FPGA. As discussed before, the obtained bitstream configures the totality of the FPGA, even if internally the FPGA resources can be addressed in units called frames. An in-depth discussion the architectural features limiting the adoption of partial reconfiguration in many Xilinx FPGA devices (mostly Virtex families) is outside of the scope of this work; for an in depth discussion of the evolution of these features, as well the evolution of the Partial Reconfiguration design flow, the reader is directed to previous work [20], or to excellent reviews in the literature [2, 21]. As stated above, the main difference between DPR and non-DPR systems using Virtex devices, lies not on architectural constraints, but in the nature of the design flow; the DPR design flow makes use of techniques and tools usually not required in traditional designs, making it more difficult to exploit [22]. However, significant advances have been made by Xilinx to improve the usability of the tools, automating many previously burdensome design flow-specific phases.

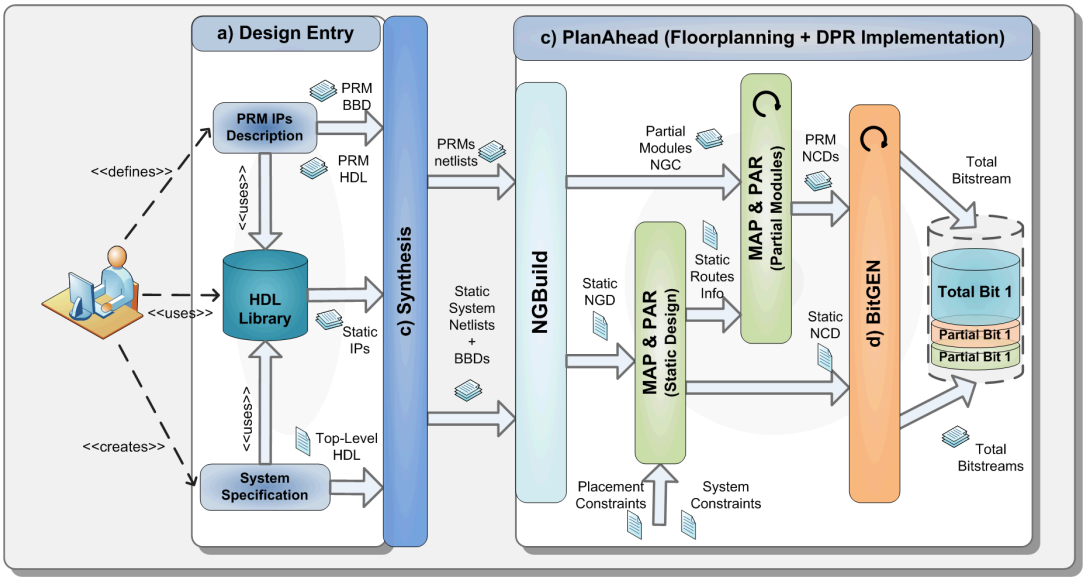


Figure 1.7: Schematic representation Partition Partial Reconfiguration design flow

In contrast to the traditional design approach, the DPR design flow, as depicted in Figure 1.7, requires more than one pass through the standard flow: each reconfigurable module passes through a reconfigurable aware version of the standard design flow [2] (specifically in the PAR and BITGEN phases). Furthermore, the results of each pass are dependent on the output of at least one of the previous passes. This is a necessary precaution to avoid introducing resource conflicts between successive configurations when they are loaded on the FPGA at run-time.

This figure presents the necessary phases required to accomplish the creation of a reconfigurable design, by using the recently introduced Partition DPR design flow [23]. In the subsections that follow, we intend to provide a glimpse of the complexity of the associated design flow, stage by step, and in a subsequent section, we further discuss the implications of using partial reconfiguration in terms of the required level of expertise. This is important in order justify the use of Model-driven Engineering approach in the context of the FAMOUS project.

### 1.4.1/ Design partitioning

A system architect starts by defining a system specification (in the form of a top-level HDL description). The system is created from an IP library containing the description of the different modules in the system. At this phase, the designer of a DPR application must already know which modules are susceptible of being dynamically reconfigured. These modules are known as Partially Reconfigurable Modules (PRM), and they are typically the netlists or HDL descriptions that are to be converted into partial bitstreams and used during the PR reconfiguration process, as explained in the previous section.

The Xilinx DPR design flow assumes that the technology is to be used to swap different configurations of the same module, and therefore, multiple PRMs need to be gathered from the IP library in order to define a Reconfigurable Partition (RP) [24]. A RP is an attribute set on an instantiation that defines the instance as partially reconfigurable, and its recognized by the subsequent design flow stages as a black-box in which PRMs netlists need to be mapped. The RP must be defined on a section of the design that belongs to the static logic, ad depicted on Figure 1.8. The partition is to be created in such a way that can accommodate the different implementations of PRMs; this means that the RP must contain a common interface that can be used for subsequent implementations of the PRMs, as depicted on the right side.

### 1.4.2/ Synthesis

For their latest DPR design flow, Xilinx has decided to perform the system implementation using PlanAhead [25]. This software tool is intended to be used for the design and analysis of circuits on Xilinx FPGAs, and is based on the principles of hierarchical floorplanning. It is deployed between synthesis and place-and-route and enables the designer to more rapidly analyze, modify, constrain and implement his designs. PlanAhead is an optional tool in the traditional FPGA design flow, which complements the mainstream ISE tool chain [26]. It is typically deployed for designs requiring the highest performance and consistently leads to faster and more compact solutions. It also promotes design reuse through the creation of reusable intellectual property blocks.

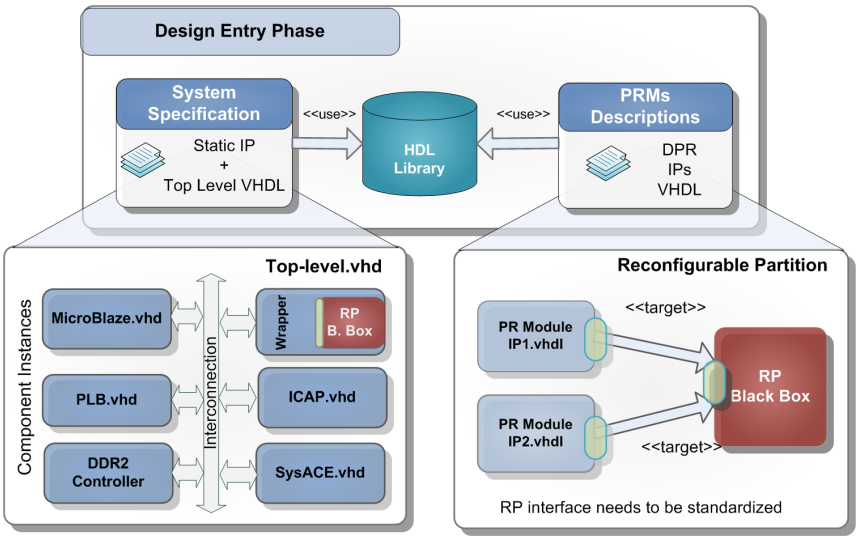


Figure 1.8: Design Entry Phase and Design Partitioning

PlanAhead allows importing the logical design hierarchy in the form of NGC or EDIF files, and then proceeding to the implementation of the static and partially reconfigurable modules. Nevertheless, the system description obtained in the design entry must be transformed into netlists through a synthesis phase, as depicted on Figure 1.7, before moving to the system implementation phase. In contrast with the traditional FPGA design flow, where the complete system specification is synthesized and then fed to the subsequent phases for implementation, the DPR design flows requires separated netlists for the static and reconfigurable modules in the design.

The synthesis for DPR systems proceeds as follows. First, each PRM is synthesized independently from the others in a bottom-up fashion. This can be done through the use of independent projects, either through a graphical interface or on the command line. For each module, IO insertion must be disabled, as the ports of these modules (in most cases) do not connect to package pins, but to the static logic above. Then, in parallel, the static modules are synthesized together to generate one netlist or individually, to generate multiple static netlists. The NGDBuild utility, in the following stage, merges the static and reconfigurable modules, and the Reconfigurable Partition definitions denote the interfaces between the static and reconfigurable logic, and the placeholders where the PRMs are to be mapped and implemented.

### 1.4.3/ PLANNING

Once the netlists for the static logic and PRMs have been obtained, they are imported into the PlanAhead tool. The most important steps of the partial reconfiguration design flow take place in this tool. The first stage is the planning (or floorplanning) of the reconfigurable areas, in which appropriate sections of the FPGA must be reserved for implement-

ing the different PRMs assigned to a particular RP in the partitioning phase, as depicted in the right side of Figure 1.9. These areas are known as Partially Reconfigurable Regions (PRRs) and have to be planned in such a way that they can accommodate the largest PRM defined for a reconfigurable partition (known as resource budgeting).

PlanAhead automates the detailed floorplanning of the design and the PRRs. The corresponding physical hierarchy is then created and mapped to the initial floorplan. PlanAhead uses the concept of a physical block (PBlock) as the basic unit of physical hierarchy; PBlocks partition a design physically and can be composed hierarchically for block nesting. Once created, the PBlocks can be placed and sized on a floorplan of the target FPGA. Various resources estimates are immediately available (prior to place and route) to guide the designer in defining the floorplan and iteratively improving the design. The PRRs are defined as PBlocks and the resources underneath (in terms of CLBs, DSP ,and BRAM blocks) can be compared with the requirements of the largest PRMs in order to decide the optimal size and dimensions of the PRR. This step is remarkably simple due to a highly intuitive interface and allows obtaining accurate resource utilization information for carrying out the logic budgeting in a straightforward manner.

PlanAhead generates customized User Constrains File (UCF) containing information about the area reserved for each partition. The static design components are mapped to a single PBlock. Each PRM is constrained to particular sections of the FPGA by using AREA_GROUP and RANGE constraints in the UCF file. An AREA_GROUP is a grouping constraint that associates logical design elements with a particular label or group. AREA_GROUP constraints and partition definitions are necessary to delineate the static (non-reconfigurable) logic from the reconfigurable logic, preventing logic in the static design from merging with logic in the PRMs, and vice versa. The RANGE constraints enumerate the resources (e.g. CLBs, BRAMs, DSP blocks) required by each of the PRMs. Then the UCF file is fed, along the NGD files for the static module, to the next phase, in

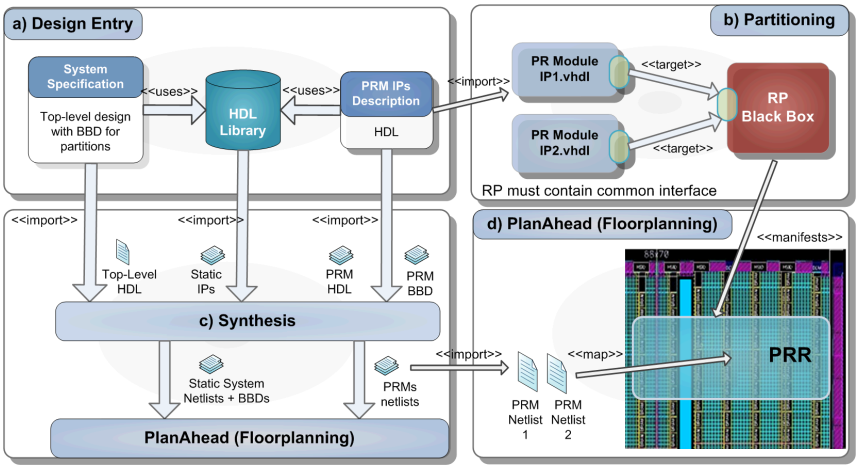


Figure 1.9: Design Planning and Floorplanning Phases

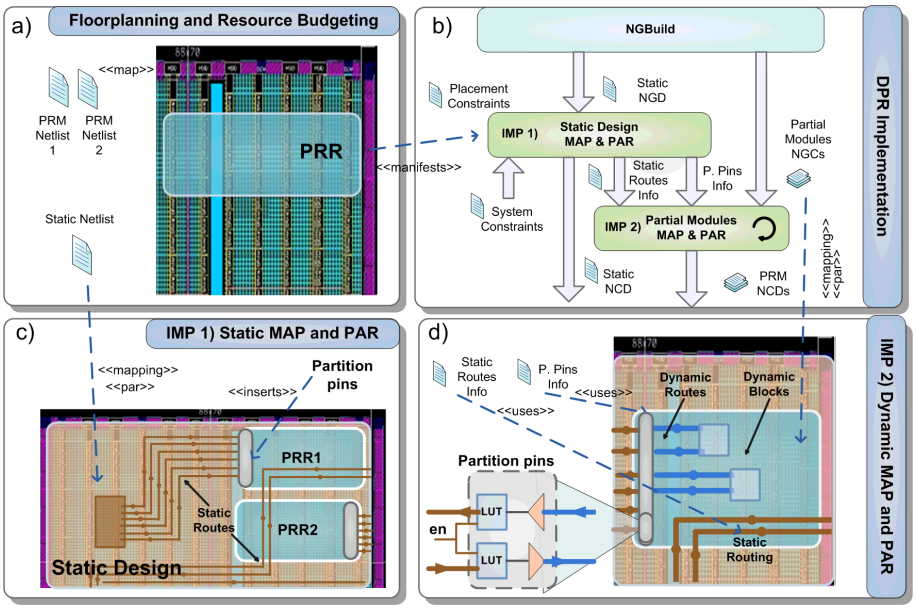


Figure 1.10: Floorplanning and Implementation Phases of the DPR design flow

which the static component of the platform is implemented.

## 1.4.4/ DPR System Implementation

### 1.4.4.1/ Static Design Implementation

After the netlists have been imported into PlanAhead, they are translated into the native Xilinx format (NGD) by running the NGDBuild tool. The resulting native netlists are then fed, along the placement constrains from the Floorplanning and Logic Budgeting Phase (Figure 1.10 a)) to the MAP and PAR tools for implementing the design. In contrast with the traditional FPGA approach, the creation of PDR designs requires more than one pass through the standard design flow - each reconfigurable module passes through a reconfigurable aware version of the static specification, as depicted on Figure 1.10 b).

From the figure, it can be seen that the results of the static implementation (labelled as Static Routes and Partition Pins info) are forwarded to the implementation of the PRMs, which are processed independently. This is a necessary precaution to avoid introducing resources conflicts between successive configuration bitstreams when they are loaded on the FPGA at runtime. In fact, static design routes may pass completely through a PRR. This alleviates routing network congestion around each PRR (often static routes need to reach I/O blocks close to reconfigurable areas); the information of which resources have been used by the static design has to be stored in order to prevent its usage in the PRMs and consulted subsequently during the PRMs implementation phase.

A graphical example is provided on Figures 1.10 c) and d), which depict the result of

the MAP and PAR of the static section of a DPR design containing two reconfigurable regions. As it can be observed on the figure, the static design contains the so-called Partition Pins at the boundary between static logic and reconfigurable logic. Partition Pins are necessary to guarantee that the circuit connections between the static logic and the different PRMs for each RP remain identical for each configuration (they act as proxy logic or anchors). The partition pins are also a convenient way for creating timing constraints on nets that pass to, from, or through the RP boundary, and are inserted automatically during the static implementation phase, in contrast with previous versions of the DPR design flow, which required the use of special pre-routed bus macros. Figure 1.10 d) shows a zoom of the partition pins implementation, which make use of LUTs at fixed locations to implement the different configurations of the PRMs, effectively providing a static interface for the reconfigurable modules.

#### 1.4.4.2/ PRM Implementation

The DPR design flow promotes the concept of dynamic configurations. This implies that each combination of PRMs in a DPR design constitutes a different configuration; for instance, if each of the PRRs can hold 2 different PRMs then, there are at least four different configurations and three more if one or both PRRs are not configured (blank state for reducing power consumption). Before proceeding with the implementation, the user must explicitly define all the PRMs that are to be mapped to a given PRR, as depicted on Figure 1.10 a).

The MAP and PAR tools then implement the static design with a combination of PRM modules, selected by the user (depending upon the needs of the intended application); as mentioned in the previous point, the implementation tools insert partition pins in the interface of the static logic and the RP. This information is stored in intermediate files (in particular, information of static routes and the location of the partition pins, as shown on Figure 1.10 c)) and used for subsequent passes, when implementing the remaining PRMs netlists (IMP2 in the second MAP and PAR block of Figure 1.10 b)). As depicted on Figure 1.10 d), the implementation of the subsequent configuration is done taking the partition pins as a reference point, and using the static routes information to avoid conflicts the reconfiguration process.

### 1.4.5/ Bitstream Generation

Just as the implementation phase, the generation of partial bitstreams is an iterative process. Each configuration (a combination of static and PRM netlists) must undergo the implementation phase separately. The bitstream generation phase follows a similar pattern: a static bitstream and partial bitstreams are created for each pass. Figure 1.11 a) continues the example used previously, but showing two configurations (two pairs of

PRMs to be mapped onto two PRRs). The first run of the DPR flow produces a set routed NCD files, both for the static and partially reconfigurable modules, what are fed to the Xilinx BitGen tool, as depicted on Figure 1.11 b).

As depicted on the first part of Figure 1.11 c), these files are used to produce the first configuration, consisting on a total bitstream and two partial bitstreams (corresponding to the Cfg 1, which uses modules PRMA and PRMB ). The static design constraints are then copied for a second iteration, in which the configuration Cfg2 is produced, this time using the modules PRMC and PRMD. Alongside the full bitstreams, the process can also generate blank bitstreams for each of the PRRs (not shown on the figure); when written onto the FPGA, a blank bitstream has the effect of removing the contents and connections of a configured PRM from a given PRR.

The rationale behind producing several total configurations is to enable the designer to choose the initial configuration of the FPGA (since the device needs nevertheless to be configured in it entirety at power-up) and then swapping the partial bitstreams as needed by the application.

### 1.4.6/ RECONFIGURATION MANAGEMENT

In the previous sub-sections, we have described how a system specification is used to produce a set of bitstreams that are used to adding flexibility to FPGA-based systems through run-time partial reconfiguration. The reconfiguration process can be achieved on-demand (i.e. using external user commands as an input) or through a remote connection [27]. However, most systems require to be controlled autonomously, meaning that the

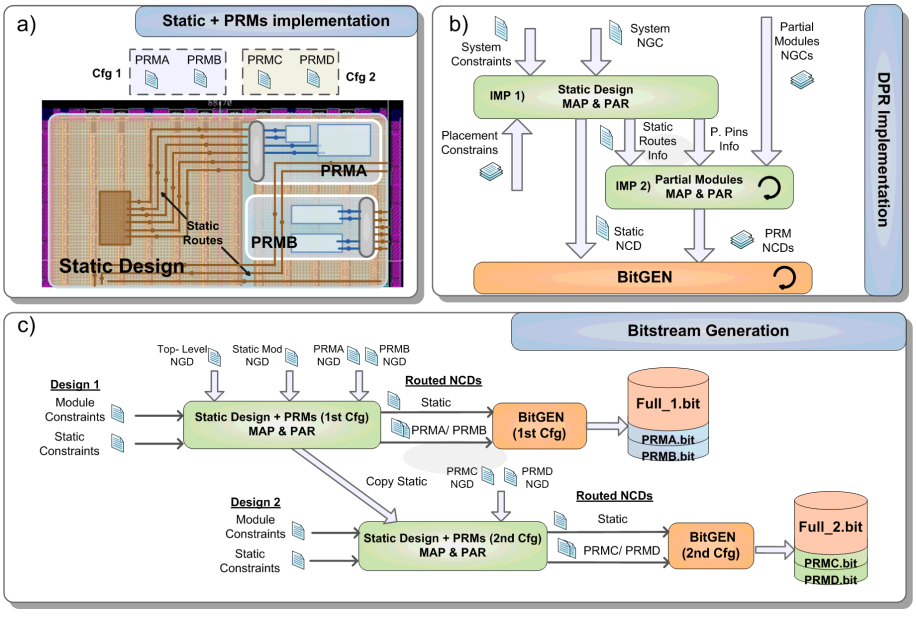


Figure 1.11: Bitstream generation phase

management of the partial bitstreams has to be done by a processor [28], either inside or outside of the FPGA fabric (using a hard processor such as the MicroBlaze, PowerPC or more recently, the Zynq EPP).

Using a processor facilitates the partial reconfiguration process, given that the application running on it can be easily adapted if changes are required during its lifetime (in contrast to a VHDL based controller, which would require a great deal of effort) [29]. Many works have been proposed in the literature for optimizing the reconfiguration process (for instance, in terms of reconfiguration time) or for providing Operating System Capabilities to DPR platforms [30, 31]; however, it is not the goal of this thesis to explore these works in-depth. Nevertheless, the discussion that follows focuses on the added complexity (application development, tool usage, expertise requirement) inherent to this phase of the DPR systems implementation.

The development of processor-based FPGA platforms, using Xilinx FPGAs, is better facilitated by the use of the Embedded Development Kit [32], which is composed by Xilinx Platform Studio and the Sofware development kit [13]; the first tool is used for creating the hardware platform, whilst the latter is employed for developing code for the target processor. Using the EDK flow, as depicted on Figure1.12, the system description can be obtained from Xilinx Platform Studio; for accelerating the development of the application (DPR or not) this description is fed to Xilinx SDK, in which the software development can start before all the details of the hardware platform are finalized. The software developer creates an application using C code and set of drivers for communicating with the hardware IPs from a library; then the code and drivers are compiled for creating an executable file. When the DPR system has been created using Xilinx PlanAhead, the configuration bitstream can then be combined with the application bitstream (the executable to run on the program memory of the chosen processor) using the Data2Mem tool, generating a complete bit file. This bitstream is to reside in non-volatile memory for configuring the FPGA at power-up, along the partial-bitstreams (although, in some cases, the partial files can be accessed remotely for minimizing the memory footprint [33]). Once the FPGA has

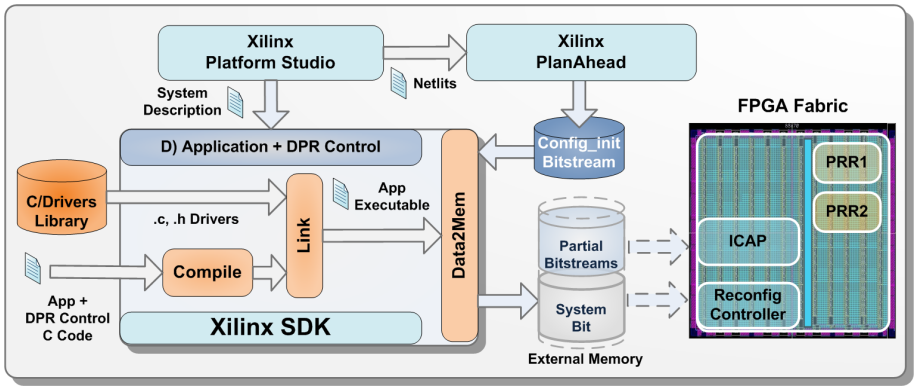


Figure 1.12: Generation of the DPR control application using Xilinx tools

been configured, the partial bitstreams can be loaded onto the reconfigurable region by writing the reconfiguration information into the Internal Control Access Port (ICAP) [34], a task controlled by the processor, and which must be taken into account during the application development.

## 1.5/ Challenges in the conception of Partially Reconfigurable Systems-on-Chip

In the previous section we have described the Partition-based Xilinx Partial Reconfiguration, step by step in the development process. In this section, we intend to provide more insight on the inherent complexity of the flow, both in terms of the tool ecosystem and the required expertise for implementing DPR systems, from the design entry phase to the reconfiguration management itself. Subsequently, we will briefly tackle past and current endeavours aiming at extending the capabilities of the traditional DPR design flow for creating more complex applications and architectures, but that ultimately further complicate the proceedings. Then, we introduce the FAMOUS (Fast Modeling and Design Flow for Dynamically Reconfigurable Systems) approach, which this thesis work is a part of, and we discuss how the methodology intends to alleviate the conception of complex DPR system through a Model-driven Engineering approach. Finally, we circumscribe the limits of this thesis work in the context of the FAMOUS methodology, and we introduce the main contributions of this work, which will be used as a launching pad for the rest of the second part of this manuscript.

### 1.5.1/ Dynamic Partial Reconfiguration Tool Flow

The DPR flow methodology differs significantly from the standard flow used to create FPGA- based designs, but the underlying processes used by Xilinx to take the design from its original entry description to the implementation remains largely unchanged in the sense that all the Xilinx tools are still used and its only in the implementation phase where there is a significant difference. The Figure 1.13 depicts the Partition Design Flow in terms of the necessary steps and Xilinx software tools required to implement a Dynamic Partial Reconfiguration design using the new design flow.

As discussed in the previous section, the creation of DPR systems is a complex methodology, that requires a thorough planning, from the design entry phase, in which the system architect, but also hardware developers must take special constrains into account for the development of the platform. Regardless of the chosen flow for the entry phase (purely VHDL or based on Xilinx EDK), the modules need to be designed following strict rules and furthermore, require the use of several tools with different directives. The knowledge required to develop a processor-based design adds to the complexity of the task,

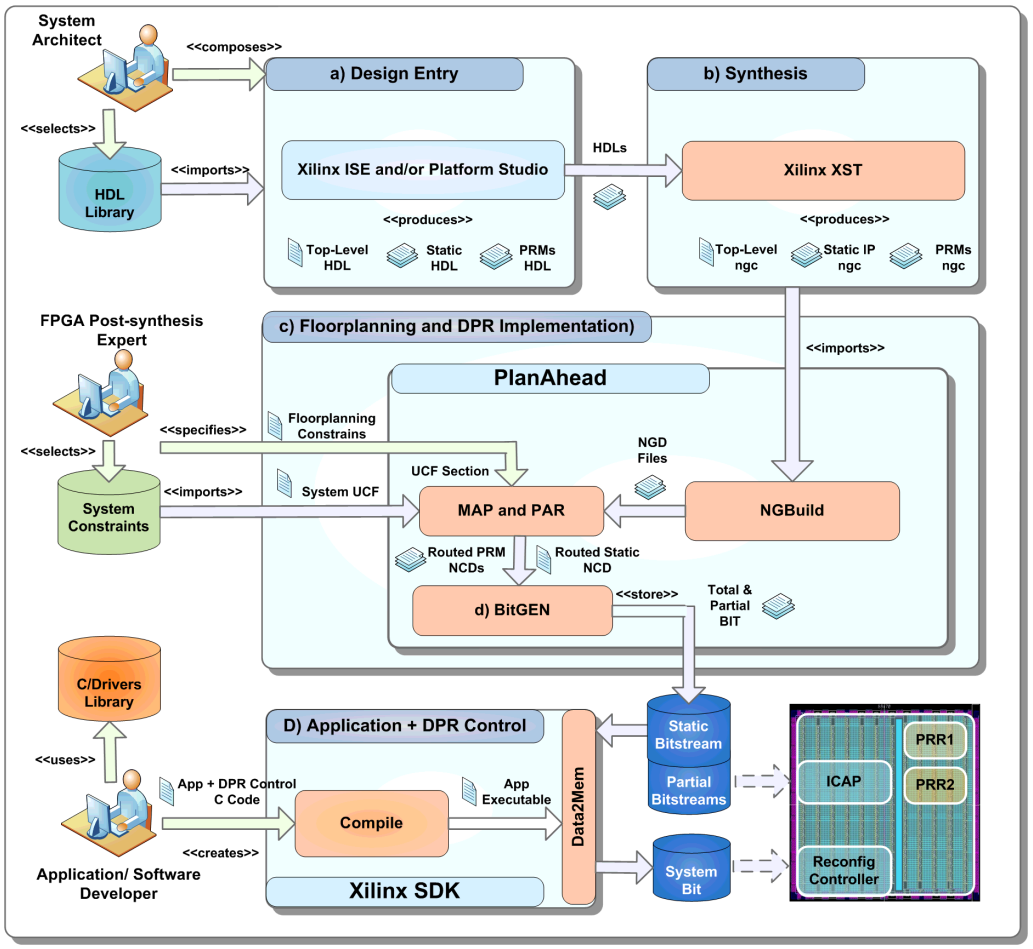


Figure 1.13: Sequence and tools usage flow for the Dynamic Partial Reconfiguration design flow

because several models apart from pure VHDL descriptions are used for managing the platform and the IPs; in particular, selecting IPs from a library, instantiating them (which implies their configuration and integration) them requires in some instances high levels of expertise, even if some plug-and-play capabilities are built into the platform. Moreover, the design entry process can be very time consuming if it is to be performed manually, with the addition burden of being very prone to errors.

The second phase of the DPR design process, in which the platform is floorplanned and generated using PlanAhead, requires as well a great deal of expertise. In the first place, an expert in FPGA post-synthesis must provide constraints for IOs in the design, and timing constraints for many of the signals, both internal and external. Furthermore, constraints for placing the static and partially reconfigurable regions have to be defined; the Xilinx DPR tools do not automate this process, given its complexity, and thus many of the steps have to be done manually and often in an iterative manner, until the optimal solution is met. For instance, selection and size of the reconfigurable region, as well as the availability of underlying resources for multiple reconfigurable modules is of the

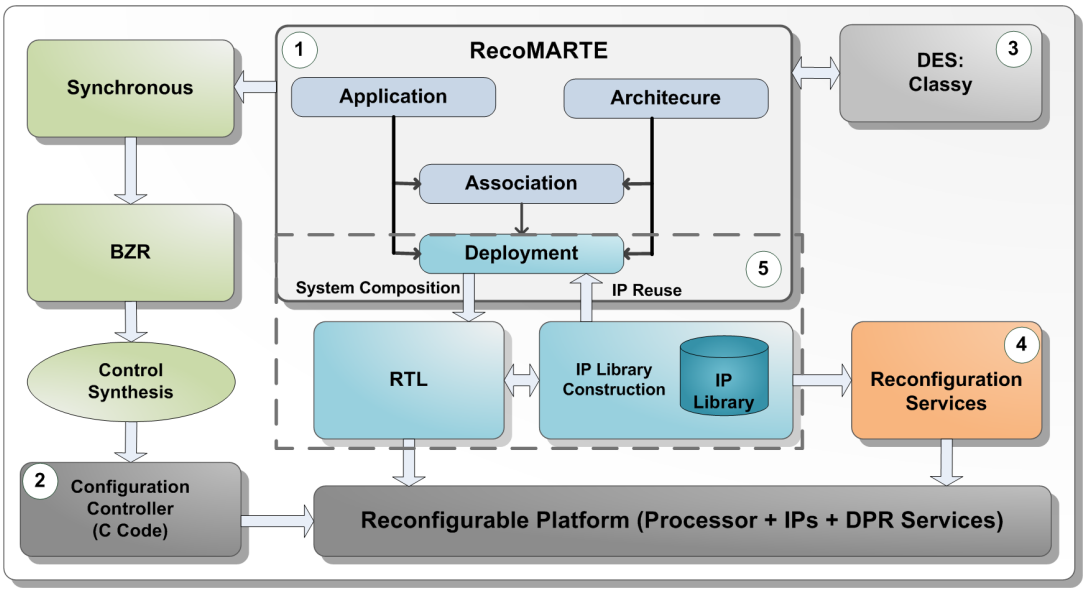


Figure 1.14: The FAMOUS Framework for the modeling and generation of DPR systems

utmost importance; given that one of the aims of partial reconfiguration is optimizing the hardware resources utilization, a poor floorplan can bring more harms than benefits. Last by not least, it must be noted that most of these constrains have to be entered manually in several User Constraints Files (UCF), which complicates that task for an user not familiarized with the low-level nuances of the FPGA design process.

Finally, the generation of the system and code for controlling the reconfiguration process requires knowledge of several tools (Xilinx XPS and SDK and the design philosophies behind their conception). As mentioned in the previous section, each tool in the Xilinx tool ecosystem is highly dependent on each other; the outputs from the hardware tools have to be imported into the software tools, in a not always transparent manner. As with any other SoC co-design methodology, there are hardware dependent software aspects that must be taken into account and, in the context of partial reconfiguration in particular, not enough has been done. This thesis is circumscribed in an academic endeavour whose aims are at facilitating the creation of DPR systems from high-levels of abstraction, while hiding as much as possible the technological and tool dependencies described in the last two sections. This framework will be introduced in the next section.

### 1.5.2/ The FAMOUS Framework for DPR Systems

One of the main objectives of FAMOUS framework (depicted on Figure1.14) is to provide a usable design flow for easy-of-use dynamic partial reconfiguration. Reconfiguration should enable the reduction of the size of FPGA and offer new opportunities for optimizations such as reduction of power consumption and more. These advances, coupled with new implementation capabilities provided by reconfigurability could offer new research

and industrial avenues.

The FAMOUS framework relies on a Model Driven Engineering(MDE) approach, using RecoMARTE, an UML Profile extension of MARTE for describing reconfigurable systems at a high-level of abstraction. The associated tools and methods therefore assist the designer throughout the whole design process and handle automatically transformations from one level to another. Such CAD environment, with the aid of analysis and verification tools has been envisioned to lead to less error prone implementations and therefore significantly reduce the development time of DPR systems.

Some of the efforts in the FAMOUS projects are described as follows:

- **1.** Identify concepts related to Partial Reconfigurable systems, and integrate them to the UML MARTE to obtain an extended and complete profile, RecoMARTE [35] which can then be used for modeling DPR systems. However, the project interests are beyond profiling and systems modeling [36], but several works aim explicitly at the generation of executable/synthesizable models to demonstrate the feasibility of the FAMOUS approach, depicted in Figure 1.14.
- **2.** The automatic generation of a correct by construction DPR controller [37], obtained by transforming the UML application model containing information about different modes and configurations of the system. The obtained controller can be either a C function to be integrated in a processor or a VHDL code to be mapped in the FPGA along other functionalities of the system.
- **3.** A validation approach, from RecoMARTE models, within which the system configurations are first analyzed modularly to ensure the correctness of each one. Each configuration is addressed by using static analysis techniques such as those implemented in the compilers of synchronous languages in order to check the absence of causality cycle, of behavioural determinism, of system reactivity. These validation techniques have been studied through the use of the CLASSY tool [38].
- **4.** The definition of a set of DPR services to be implemented by the system, such as context saving of the virtual tasks (reconfigurable modules), which improve the adaptivity of partially reconfigurable systems. Such DPR services are introduced in the form of a wrapper to be integrated in the DPR IPs [39] .
- **5.** The generation of the platform description by means of a meta-data driven composition framework using the IEEE standard IP-XACT [40]. This XML specification has been widely adopted by the industry and its used as an intermediate representation (IR) between the UML MARTE platform models and Xilinx DPR tools backend proprietary representations.

  The IP blocks used to build the platform were converted to IP-XACT descriptions and subsequently to UML MARTE templates through model transformations for pro-

vide our methodology with IP reuse capabilities. Then, the IP library is to provide MARTE with an effective and flow agnostic meta-data driven hardware generation approach [41].

## 1.6/ Discussion and Conclusions

This thesis addresses the point number 5 on Figure 1.14 (enclosed by the dashed grey square). The main goal of this work is to define a platform generation flow for facilitating the design entry phase of the Xilinx Dynamic Partial Reconfiguration design flow, as depicted on Figure 1.13. As discussed before, the creation of DPR systems is facilitated by using a component-based SoC approach, in which the system architect gathers IPs from a library and composes a top-level platform (a process traditionally consisting on instantiating components in VHDL, parameterizing and interconnecting them): a prone to errors procedure which, if automated, could greatly facilitate the use of the flow.

These automation capabilities are provided by the use of MDE techniques, which will be discussed in-depth in the next chapter; MDE makes use of models and models transformations for describing and generating executable SoC platforms, respectively. Nevertheless, most MDE approaches do not provide mechanisms for reusing low-level component descriptions, and therefore, a means for performing this mapping must be provided. This is achieved through the construction of an IP library, which can be used for composing the platform in a high-level language, such as UML MARTE (an IP reuse approach). The composed platform can then be used to obtain a synthesizable netlist, typically in RTL (design by reuse); in the FAMOUS framework, we make use of an intermediate representation, the IP-XACT standard, for passing from the high-level models to the targeted back-end (Xilinx Platform Studio), but also for describing the IP components in the library, significantly facilitating design automation.

Furthermore, this thesis work is closely related with other endeavours of the FAMOUS approach. First, we deal with some modeling aspects in UML, specifically at the Deployed Allocation level, as depicted on Figure 1.15 a), which depicts the hardware MDE branch of the FAMOUS methodology, in terms of model transformations. We propose the use of a double Y-schema (more details will be provided in the next chapter), in which in a first stage, through the ⟨⟨allocation⟩⟩ operation, elements in the application are mapped (⟨⟨allocated⟩⟩) to components in the platform.

Using the same modeling diagram, the deployment maps the elements in the application and architecture models to their implementation counterparts. This is the rational behind the use of the two libraries at the combined ⟨⟨Deployed Allocation⟩⟩ level; it must be noted that the Task IPs in the figure can denote software code to be run on processors, or hardware IPs to be mapped to the reconfigurable partitions (in the form of DPR Wrappers). The second interaction of this thesis with other FAMOUS works is related

to the reuse of reconfigurable services, that are integrated in the form of IP wrappers to chosen modules, for performing tasks such as context saving. This wrapper must be automatically integrated into the platform, undergoing the traditional parameterization and customization procedures, which are performed in the Deployed Model.

The Deployed Allocation phase produces a model in which all the information of the components (Static, DPR Partitions and PRMs) is readily available and linked to IP-XACT components to be configured. A ⟨⟨Deployed⟩⟩ model, containing this information, along their interconnections (in the case of Static and reconfigurable partitions) and parameterization data, is used at this phase for generating an executable model, effectively generating the outputs of the Xilinx DPR design entry phase, as depicted on Figure1.15 b). In order to make this possible, the ⟨⟨Deployed⟩⟩ model is fed to a Model Transformation engine, containing a set of meta-models and transformations rules.

The models described before, in tandem with the modeling environment can be seen as a Metadata-driven Composition Framework (MCF, concept to be discussed in the next chapter) targeting the generation of a variety of proprietary back-end models. As described in [40], we target the generation of the proprietary Xilinx Platform Studio [42] models; this approach facilitates the creation of a framework in which the hardware and software components of a DPR SoC can be jointly developed. Thus, through model transformations, we convert the ⟨⟨Deployed⟩⟩ model, first into an IP-XACT design description (providing a flow-agnostic intermediate representation to our approach) and then a Xilinx XPS model, used by this tool to create the VHDL top-level skeleton. In parallel, the PRMs are synthesized using information contained in their corresponding IP-XACT component descriptions. In this manner, we provide the inputs for the DPR design flow in the form of netlists after the synthesis phase.

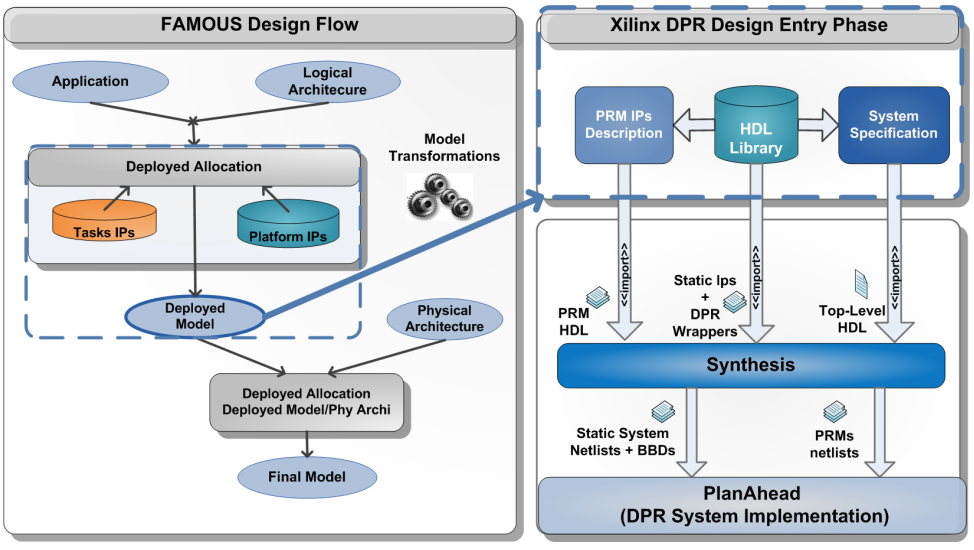


Figure 1.15: The proposed MDE approach (a) and its interactions with the PR design flow (b)

# 2

# IP Reuse in MDE-based Co-Design Methodologies

## Contents

## 2.1/ INTRODUCTION

The semiconductor industry has continued to make impressive improvements in the achievable density of very large-scale integrated (VLSI) circuits [43]. On the other hand, competition among its players demands a reduction in the design time and cost, while nevertheless improved design reliability and quality are also required. In order to keep pace with the levels of integration available, design engineers have developed new methodologies and techniques to manage the increased complexity inherent in these large chips [44]. One such methodologies is System-on-Chip (SoC) design, or more recently, Network-on-Chip, wherein pre-designed and pre-verified blocks often called intellectual property (IP) blocks are obtained from internal/external sources, combined on a single architecture, and then implemented in either ASIC or FPGA technologies to speedup the overall design process. These reusable IP cores may include embedded processors, memory blocks, interface blocks, and components that handle application specific processing functions (i.e. hardware accelerators) [45]. The same problems arise in FPGA-based SoC systems, and this is especially true in systems supporting partial reconfiguration, which make use of very heterogeneous sources of IPs (static and dynamic modules along the conventional blocks described above).

Many partial solutions to tackle the aforementioned issues have been proposed such as Electronic System Level, Hardware/Software Co-Design, Platform Based Design and UML for SoC [46]. However, none of these approaches have met widespread adoption by the industry or academia. Nonetheless, it has been recognized that an effective solution for the SoC Co-Design problem consists on raising the abstraction level in which such systems are created (a top-down approach). This approach increases productivity of software/hardware developers by reducing the amount of effort needed to develop and maintain complex systems [47], offering two main benefits. First, it provides an incremental or bottom-up system design approach permitting to create complex systems, while making system verification and maintenance more tractable. Secondly, this approach enables a reduction in the development efforts, as components can be reused, at least theoretically, across different tools and design flows.

Current SoC Co-Design practices coalesce many of these approaches with the aim of increasing design efficiency. For instance, a combination of IP re-use, Hardware/Software Co-Design and High-level modeling could potentially ameliorate the design process of complex embedded systems by bringing together the best of each domain [48]. First, IP reuse helps in separating concerns so that unacquainted work can be accomplished by certain experts of that domain, creating a library-of -plug and play components which can be automatically integrated onto the platform (making this approach orthogonal to component based design). Secondly, Hardware/Software Co-Design enables concurrent design methodologies, whereby hardware design is abstracted and becomes closer to software design [47]; for instance, software programming languages are extended for the

hardware domain (e.g., HandelC, SystemC), which reduces the complexity by using the same programming models. Finally, high-level modeling contributes in raising the level of abstraction, while at the same time, enables the partition of the system design into different aspects: it enables the specification of parallel independent models for both the system hardware and software, the allocation (or mapping) and the possibility of integrating heterogenous components into the system [35]. The use of UML and other graphical languages (which, in this context, can be seen as metadata-driven component composition frameworks) increases the comprehensibility of the system specification, as it enables designers to carry out high-level descriptions of the system, easily illustrating the internal concepts (hierarchy, connections, dependencies, parameterization). The graphical nature of these specifications equally facilitates reuse and refinements, depending upon the underlying back-ends, adding flexibility and adaptivity to the flows in which they are inserted.

## 2.2/ MODEL-DRIVEN ENGINEERING

In recent years, SoC designers and architects have begun to look at UML for possible improvements to the specification, design, implementation and verification processes, as UML provides several means for architectural as well as behavioural design, which have been well established in HW and HW/SW Codesign for some time [49] [50]. The different UML diagrams or variations of them already have found application in various areas such as: requirements specification, test-benches, IP integration, and architectural and behavioural modelling. The fundamental appeal of Model-based methods is that they let system designers use abstractions matching their primary design concerns rather than be constrained by properties of the implementation technology they target.

A model-based design process comprises a range of models representing different aspects of system behaviour. The automated design process proceeds by refining, integrating, and analyzing these models in a complex flow [51]. This complex, iterative model construction process is combined with model analysis to establish required properties and model transformations for integrating models, extracting information for analysis, or translating them into code. The richness of the model-based development process is formally encapsulated by the MDE paradigm.

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 [52]. Above all, MDE plays a very important role, which contributes to modeling, automatic code generation and bridging the gap between different technologies. The key concept in MDE is the model, in addition to a couple of core relations related to the former. The first is representation (a model is a representation of a system); the second, conformance (a

model conforms to a metamodel) [53]. These two relations are separately presented in the following sub-sections. Another key notion of MDE, model transformations, is also discussed. Furthermore, the advantages of MDE are illustrated subsequently following the introduction of the previous concepts, along how MDE is put in practice for the modeling and design of embedded systems.

## 2.2.1/ MDE Basics

### 2.2.1.1/ 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 perspectives (views in MDE). The abstraction mechanism avoids dealing with details and eases re-usability.

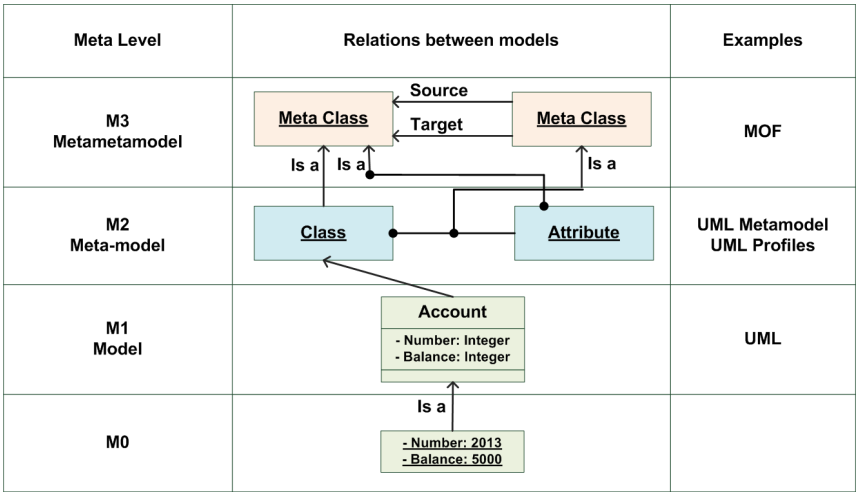


Figure 2.1: Different levels of modeling in MDE

### 2.2.1.2/ Meta-models

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 (such as UML [54]), and is used for defining the syntax of a model. A metamodel can be viewed as an internal representation of the different models 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 signify 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 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 [55] and MOF [56], are examples of such kind of metamodels or metametamodels.

Figure 2.1 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. For instance, 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.

#### 2.2.1.3/ Model Transformations

Models in MDE are not only used for communication and comprehension, but by using model transformations [57], to produce concrete results such as executable source code. With the help of metamodels, 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 2.2; it can be seen as 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.

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

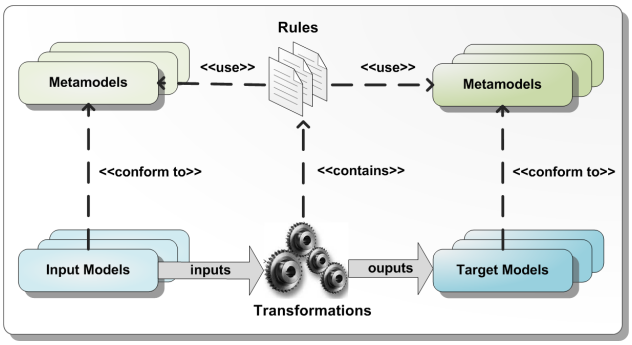


Figure 2.2: Relationship between meta-models, models, transformations, and rules

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 [58]), 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 reusability 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 [59]. The corresponding source/target patterns directly cover the whole source/target models. Transformations are in general mixed-style in nature (having both declarative and imperative rules) so that complex transformations can be implemented.

#### 2.2.1.4/ A MULTI-LEVEL APPROACH IN MODELING AND TRANSFORMATIONS

The Model Driven Architecture, or MDA, is the most widespread approach in Model-Driven Engineering. This approach has been introduced by the Object Management Group (OMG), an international consortium [60]. The MDA approach brings together multiple standards, wherein the Meta-Object Facility (MOF) [56] is a metametamodel, and the Unified Modeling Language (UML), in conformity to MOF is employed for the creation of models (and if extensions are required, it permits the creation of the so-called profiles). Furthermore, the XML Metadata Interchange (XMI) standard [61], which describes a format based on XML is used in the context of the serialization of objects conforming to MOF (typically, an UML model can be encoded into an XMI file, so it can be used by other tools).

The basic principle of MDA is the division of the problem specification in different mod-

els. Between the different abstraction levels of a modeled system and its resulting code, intermediate levels then 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.

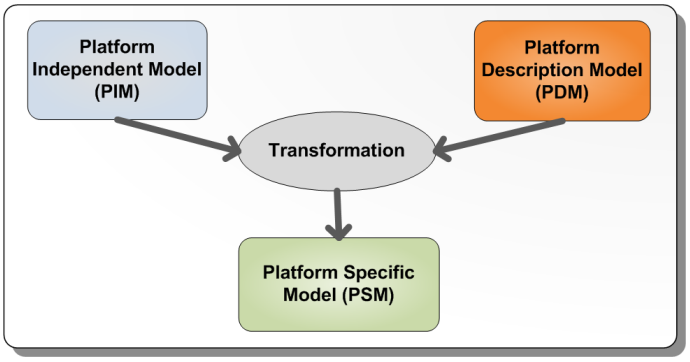


Figure 2.3: A global overview of the MDA approach

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 (Platform Specific Model). 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.

### 2.2.2/ The UML MARTE Profile for Embedded Systems Design

UML is considered as one of the main unified visual modeling languages in MDE. The UML metamodel [54] 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 [62].

It has been proposed to answer the requirements of modeling specification, communication, documentation, among other purposes. It integrates the advantages of component re-use, unified modeling of heterogeneous systems, different facets modeling of a system, amongst others. As UML is widely utilized in industry and academia for modeling purposes, a large number of tools have been developed for its support.

When UML is utilized in the framework of MDE, it is usually carried out through by its internal metamodeling mechanism termed as a profile [63]. 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 Domain Specification Language (DSL) [64].

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 [65]. 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), and making possible to carry out the intents of MDA [66].

The use of model based approaches for SoC co-design has been thoroughly discussed in [67]. UML/MDE has been adopted in co-design methodologies in recent years with relatively success. The extensions mechanisms capabilities of UML have stimulated its use in embedded systems modeling [68]. In particular, structural modeling has been the most prominent application of the UML in SoC design, for specification of requirements, behavioural and architectural modelling, and system integration. There are several approaches which use the UML profile and extensions to support embedded hardware resources modelling [69]. In this thesis, we do not engage in a thorough discussion of the different UML Profiles for SoC, since it is not the principal focus of our work; nevertheless, the reader is directed to works that deal with the advantages and shortcomings of the different approaches [35].

Among the UML profiles dedicated to the conception of complex real-time systems, MARTE [70], a standard first introduced by the OMG, and which stands for Modeling and Analysis of Real-Time Systems, has distinguished itself from the others for its capacity to provide precise concepts for the modeling of hardware architectures related to Systems-on-chip; the standard is in fact a replacement for a previous specification, the SPT 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 additional 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 2.4 presents the global architecture of the MARTE profile and its decomposition into 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.

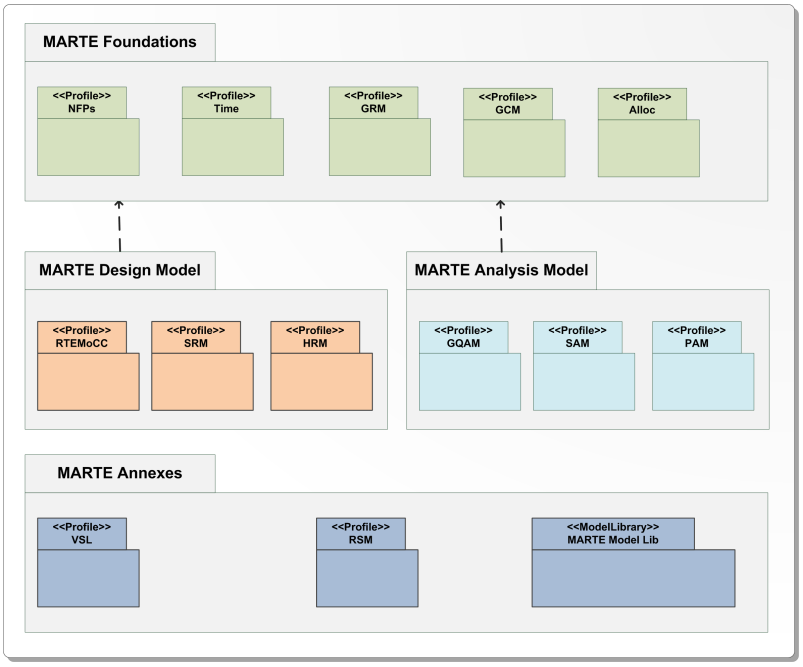


Figure 2.4: Global architecture of the MARTE profile

We briefly give an overall summary of the MARTE concepts present in both the MARTE metamodel and the MARTE profile. However, we concentrate on those packages that are closely related to the modelling of the platform components of a SoC System.

- **Core Elements:** Describes the basic core concepts in the MARTE specifications, such as Models, Model Elements (such as Classifiers) and their associated behaviours. The behaviours can be any thing such as state machines, activity diagrams, interactions (sequence diagrams/ timing diagrams), automatas, etc.
- **Non Functional Properties (NFPs):** They allow describing properties which are not related to functional aspects such as energy consumption, memory utilization,

consumed resources, etc. This is a critical aspect for a detail description of an RTES.

- **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.

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

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

  **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.

- **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.

### 2.2.3/ MDE APPLIED TO SOC CO-DESIGN METHODOLOGIES

The aforementioned MDA approach practices are well represented by the Model-driven Engineering (MDE) paradigm [71] that in which system modeling is also performed making use of 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 can be carried out to generate executable models from high-level models; furthermore, MDE is supported by several standards and

tools. Many co-design MDE methodologies have been proposed over the years (for instance [72]). Traditionally these methodologies made use of the Y schema presented originally in [73]; this approach has been adapted to represent how co-design methodologies face the system design process. Its three axes represent functional behaviour, hardware architecture and final implementation in specific technologies (e.g., circuit, programming languages). The central point of these three axes denotes the allocation of the application resources (data and instructions) onto 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 technology.

As depicted on Figure 2.5, the architectural (hardware) and behavioural (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 [74]. Moreover, concurrent and parallel development of hardware and software by different teams helps to shorten the overall design time. For instance, the software teams do not need to wait for the final configuration/netlist of the architecture conceived by the hardware team to start developing the software. The application behaviour is then mapped onto the hardware architecture, on which different analyses (such as simulation) can be performed. 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 [75]. If the mapping result is approved by the analysis, it can be utilized for implementation purposes.

The deployment phase of such a methodology is instrumental, since enables the generation of a complete and synthesizable SoC description from a high-level description in

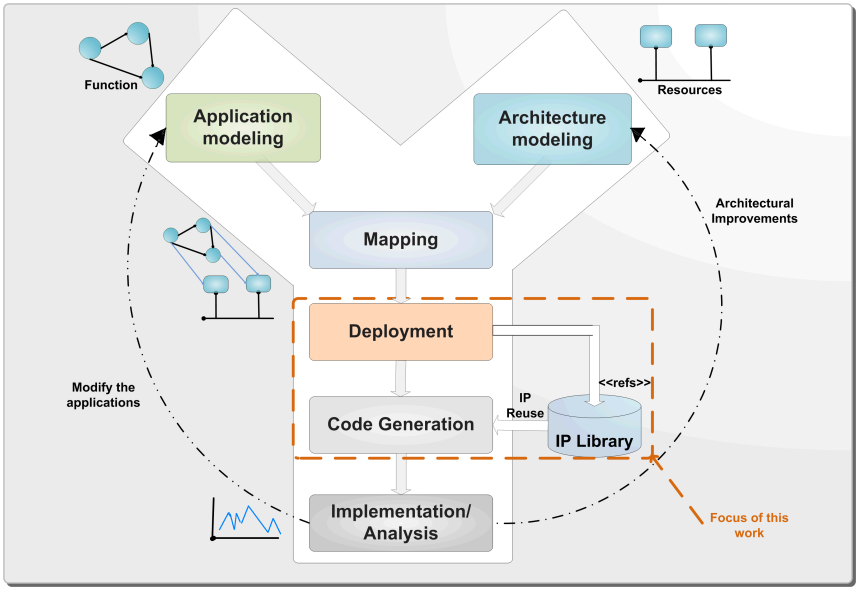


Figure 2.5: Typical SoC Co-design methodology, the deployment is the focus of this work

MARTE [35]. Designers must be able to precisely associate abstract components with their corresponding IP low-level implementations, while remaining at a high abstraction level. More precisely, sufficient information must be provided at this stage so that the code integration and parameterization on the IPs can be carried out automatically in the last step of the compilation chain, the system generation phase.

The deployment provides the designers a means to link the basic building blocks, which are the elementary components in the UML models to implementations that embody the related behaviour (i.e. IPs described using VHDL). Similarly, information related to the IP interfaces must be taken into account for facilitating IP integration and system composition. Thus, interface compatibility between an elementary component and associated IP must be ensured in order to determine the exact manner in which an IP must be used. The deployment metamodel has four main features, which are explained as follows

- **Associating IP with properties:** Traditionally, IPs need to be configured at the instantiation phase, for configuring internal functionalities or selecting certain features. Therefore, parameters information must be available so at least some functionalities can be controlled, propagating this information as constraints for the rest of the compilation chain. Furthermore, design flow specific information might be necessary, in order to make use of a particular component in a targeted back-end. For instance, in case of hardware IP, properties such as consumed resources could be provided, along the required compilation options.

- **Component re-utilization:** As mentioned previously, the reuse of hardware and software IPs is extremely important for design productivity. When developing a new SoC, the building blocks are not developed or created from scratch, but are off-the-shelf components internally developed or purchased from third parties.

  The deployment level should also enable the creation of IP libraries. The deployment level extension in the MARTE standard makes emphasis not only on how the IPs can be described independently from a specific SoC model, but also to ensure that the usage of components from a library is as simple and intuitive as possible.

- **Abstraction of associated functionality:** An elementary component linked to the implementation of a component is an abstraction of its functionality. However, this IP is specific to a target execution platform and technology (e.g. TLM, RTL). This choice must be made at the deployment phase in an easy and intuitive manner.

- **System Composition and Generation:** The deployment metamodel also provides a means of generating the executable back-end model. This is because the deployment model contains sufficient information of both the platform (component instances, their parameters, and interconnections). Then, a platform can be generated through model transformations from an intermediate representation, typically stored in an XMI file.

Despite the success and widespread adoption UML for SoC Co-design and in the modeling on real-time embedded systems, there are still several issues that must be addressed before it can be fully adopted by the industry. The main disadvantage of recent efforts in applying MDE to SoC design has been precisely in the passage from the high-level models to the code generation. This is due to the lack of standardized representation at the deployment level, in which typically metamodels of the architecture and of the IPs are used for performing the link between the high-level models and their underlying implementations (e.g. TLM or RTL). It must be noted that this is indeed an IP reuse/design by reuse problem [45], an issue that has prevented the SoC industry with promoting the much looked after design productivity gains, and which will be explored in detail in the next section.

## 2.3/ IP REUSE

This section has two purposes. The first one is to summarize the efforts carried out in the last years in the area of IP Reuse, specifically those efforts in the creation of Systems-on-Chip for reconfigurable platforms (i.e. FPGAs). The second goal of this work to delineate a framework for the specification of parametrizable IP cores which can be deployed not only in the traditional design flows for the creation of SoCs in reconfigurable logic, but also in the context of Reconfigurable Systems.

In order to propose the aforementioned framework, it is vital to understand the evolution of the efforts in the area of IP Reuse, which is one of the axes of this work. This methodology is focused on leveraging the complexity associated with the creation of Systems-on-Chip, where traditionally systems have been assembled by creating the cores as needed from scratch [73]. Through the reuse of IPs, a system can be created from a library of IP cores interconnected using commodity standard buses. The flow should produce a model which can be used through the entire process from design exploration to verification and hardware/software co-design. The final stage should allow the model to be imported in any ASIC or FPGA design flow for implementation. This section introduces basic concepts of IP reuse, and delimits the object of study of the subsequent subsections.

Afterwards, it explores the problems and main issues limiting the use of the IP Reuse design flows. Its widespread adoption has remained elusive for many reasons, being particularly important the absence of common methodologies for the description and parameterization of IP cores, the lack of standardization of interfaces, processes, flows, automation, and quality verification [76]. Nevertheless, the concept of IP Reuse as a whole is useful as a launching pad for the conceptualization of IP cores, since it provides a common ground for all the parties involved in the creation of Systems-On-Chip. Therefore, this section deals as well with the responsibilities of each of these parties for the success in adopting reuse methodologies.

### 2.3.1/ IP REUSE BASICS

A functional model of an IP component contains a lot of information, but it does not often contain the information about how it was meant to be used, what restrictions are placed on its usage or the way in which it is meant to be connected. This information is considered to be metadata about that block and was often the principal information that would have been found on a specification sheet for a device. Once that information is captured in a formalized manner it can be used by tools to help with things such as system construction, enable system consistency to be analyzed, or reduce the burden of things such as documentation. Metadata is especially relevant for the level of IP blocks and the associated models, whether hardware, software, or verification IP. As mentioned above, meta-models convert the unstructured data found on component datasheets or specifications into a more structured format that may be amenable to tool-based processing.

IP reuse and design reuse for co-design methodologies of System-On-Chip is not really new. IP is just a new name for what was formerly known as macro/mega cell, core or building block. What it is new is the growing complexity of the cells and more importantly, the increased heterogeneity of functions that can be used in a SoC. Even the pure, block-oriented instantiation of functional units in a SoC or NoC platform can be referred to as design by reuse. However, in order to actually promote IP reuse and design by reuse, the complex blocks must have been purportedly designed for this means, as depicted on Figure 2.6.

Design reuse is defined as the inclusion of previously designed components (design IPs) in software and hardware in a System-On-Chip architecture. Design reuse makes it faster and cheaper to design and build a new SoC, because the reused components will not only be already designed but also tested or verified for correctness. Hence, it is a necessity for efficient implementation of SoC designs. According to a recent study, design teams can

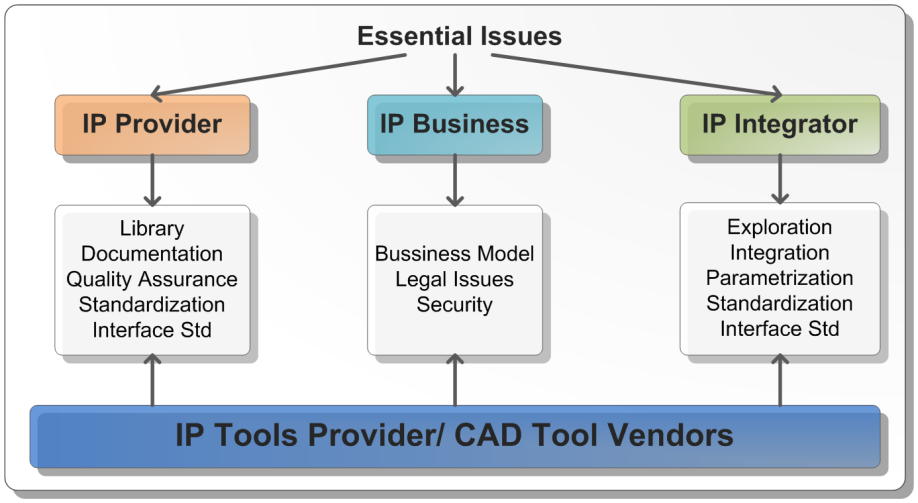


Figure 2.6: IP ecosystem and issues for widespread IP reuse adoption

save an average of 62% of the effort that they would otherwise expend in the absence of reusing previously designed blocks. Reuse of IPs span across the different abstractions, which requires extending the IP definition and reuse to different levels of abstraction

In recent years, several efforts and works have been carried out by the industry and the academia to leverage some of the problems related to IP reuse. Efforts for the standardization of IP interfaces have progressively appeared, such efforts can be globally classified as component-based design methodologies [77, 78]. The components on a SoC are connected together by an on-chip communication architecture backbone that supports all inter-component data communication, both within the chip as well as with external devices (e.g., external flash drives). These SoC communication architectures have been shown to have a significant impact on the performance, power consumption, cost, and design time of SoCs. Indeed, modern SoC design processes are increasingly becoming communication-centric, since reusable components (e.g., processors, memories, etc.), as well as custom hardware blocks and interfaces, need to be connected via a communication architecture fabric, with the goal of meeting various design constraints such as cost, performance, power/energy, and reliability.

Contemporary IP-based design flows are based on the premise of design reuse, with the system designer stitching together disparate IP blocks using the critical communication architecture fabric for data communication between the IP blocks [79]. Thus the communication architecture enables mixing and matching of different IP blocks to create a SoC. Furthermore, the communication architecture provides designers the ability to explore multiple communication schemes, topologies and protocols, thereby providing product differentiation to meet a diverse, multi-dimensional set of design constraints, including performance, power/energy/temperature, cost, reliability, and time-to-market [80]. Most component-based design approaches build SoC architectures from the bottom up, using pre-designed components with standard interfaces and/or a standard bus. For example, IBM defined a standard bus called CoreConnect [81], and ARM makes use of the AMBA AXI protocol [82]. When needed, wrappers adapt incompatible buses or component interfaces. Typically, internally developed components are tied to in-house (proprietary) standards, so adopting public standards implies significant effort to redesign interfaces or wrappers for legacy components.

This represents a dilemma: on one hand, major productivity gains are needed, leading to techniques such IP reuse. On the other hand, such approaches have made it more difficult for the SoC industry to carry out the verification and validation of systems comprised by components from multiple vendors. A possible way to reduce this time is to make further reuse of pre-existing cores in tandem with interface and parameterization standards. Thus, IP reuse and design by reuse have motivated the creation of industrial and academic consortiums with the aim of developing standards for enabling the exchange of IP cores among IP providers and SoC integrators [83]. Primarily, standards such as the Virtual Socket Interface (VSIA) [84] and subsequently the Open Core Protocol Interna-

tional Partnership (OCP-IP) [85] which aimed at creating standardized virtual sockets for seamless IP Core integration in SoC architectures, have been proposed.

Such reuse is best facilitated by model transformation techniques available with meta-modeling [86]. This kind of reuse allows coherence between the verification models at the various levels, and greater productivity by reducing the need to manually create verification IP for the design at each abstraction level. IP reuse requires that a library of IP blocks be available to the designer, along with a suitable CAD environment that lets the designer compose these IP components in a well-defined manner. Structural and behavioural constraints must be imposed on the way such composition can be carried out for the composed system to be correctly constructed. Meta-modeling helps in designing such a constrained environment; meta-information on IP components helps in checking that the constraints are met; and metaprogrammable APIs allow for adding various model analysis and transformation tools to the environment for further processing, test generation, and verification.

An orthogonal approach has been carried out by the Spirit Consortium through the IP-XACT standard [87], which proposes an IP-reuse approach to enable IP configuration and integration by utilizing metadata exchanged in XML format between tools and SoC design flows. The use of standards such as IP-XACT simplifies the description of IP cores and its integration in SoC design flows: tools by different vendors integrate IP-XACT editors and generators and it seems that its adoption will grow over time. In fact, the IP-XACT has been recently adopted as a standard by the IEEE , with the working name of IEEE 1685-2009, IEEE Standard for IP-XACT, Standard Structure for Packaging, Integrating, and Reusing IP within Tools Flows [88].

### 2.3.2/ Reusable IP

The main prerequisite for creating SoC through IP Reuse is the existence a library of IP blocks that support plug-and-play integration (using a component based approach) [89]. An IP library supporting various timing, area, power configurations is the key to SoC success as it allows mix-and-match of different IP blocks. In this way, a SoC integrator can apply the tradeoffs that best suit the needs of the target application. The process of creating a reusable IP blocks, however, differs from the traditional design approach. Typically, it may require five times as much work to generate a reusable IP block compared to a single-use block. Details on the IP design process may be found in a wide variety of sources. However, the most noteworthy reference on the subject is the Reuse Methodology Manual (RMM) [90] , which provides a comprehensive set of guidelines on the reusable IP design process. In the sections to follow, we provide an overview of the issues in design issues for reusable IP.

Digital IP blocks are the most popular and ubiquitous form of reusable IP in industry today.

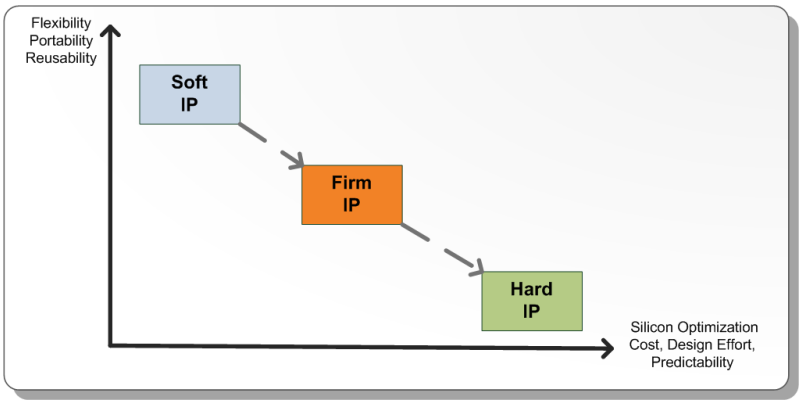


Figure 2.7: Types of IP, their relationships and tradeoffs

Many of the tools and design methodologies for creating digital IP were already in place when the concepts of reusability emerged [44, 73]. However, there have been wholesale changes in the design flow to fully enable reusable design. In addition, there are many technical issues that need to be addressed, as the IP developer must anticipate all of the applications in which the IP block may be used. The first step includes the generation of suitable documentation for the IP block. The second step includes code design, synthesis, and design for test. Somewhat surprisingly, the second step of implementation is only a small part of the reusable IP design process [91]. In fact, depending on the size and type of the IP, the third step of verification may take up to 50% of the total time. Since the IP may be reused many times in a variety of unanticipated applications, design errors within the block cannot be tolerated. The goal of verification is to achieve 100% code coverage and close to 100% functional coverage to ensure the IP block works correctly.

### 2.3.3/ Classes of IPs

The actual form of a reusable IP core can vary depending on the way in which the IP developer/vendor chooses to provide the core to the system designer. There are three main categories: soft, firm, and hard. These forms are described below and their relationships and tradeoffs are depicted in Figure 2.7 and described as follows:

- **Soft IP blocks:** Usually specified using RTL or higher level descriptions. They are more suitable for digital cores, since a hardware description language (HDL) is process-independent and can be synthesized to the gate level. This form has the advantage of flexibility, portability, and reusability, with the drawback of not having guaranteed timing or power characteristics, since implementation in different processes and applications produces variations in performance.
- **Hard IP blocks:** They have fixed layouts and are highly optimized for a given application in a specific process. They have the advantage of having predictable perfor-

mance. This, however, comes with additional effort and cost, and lack of portability that may greatly limit the areas of application. This form of IP is usually pre-qualified, meaning the provider has tested it in silicon. This adds some assurance to its correctness.

- **Firm IP blocks:** These blocks are provided as parameterized circuit descriptions so that designers can optimize cores for their specific design needs. The flexible parameters allow the designers to make the performance more predictable. Firm IP offers a compromise between soft and hard, being more flexible and portable than hard IP, yet more predictable than soft IP.

Until very recently, most digital IP blocks came in the form of hard IP. For example, ARM and MIPS core vendors would provide behavioural models and black-box layouts of the processors for use during design and verification, and then drop in the hard IP at the foundry facility before fabrication. This afforded the vendor some level of IP protection while allowing the customer to carry out designs using the IP. More recently, soft IP has become the preferred handoff level, specifically in the area of Reconfigurable Logic and Reconfigurable computing, where companies offer IP solutions along their logic and electronic platforms [11, 92].

Typical soft IP include interface blocks (USB, UART, PCI), encryption blocks(DES, AES), multimedia blocks (JPEG, MPEG 2/4), networking blocks ( Ethernet), and microcontrollers (Xilinx PicoBlaze, MicroBlaze or Leon). The RTL descriptions usually are configurable in the sense that certain parameters are user-definable to tailor the block to the needs of the customer. This allows selective inclusion or exclusion of distinguishing features which can impact the final implementation performance, cost, and power. In the case of a processor core, parameters such as the bus width, number of registers, cache sizes, and instruction set may vary from customer to customer, so the flexibility of soft IP allows for their modification before synthesis. Other IPs, for instance video controller, might support multiple interfaces or standards which do not need to be synthesized and implemented on the FPGA or ASIC fabric) and thus, a customization mechanism through constraints must be used at the deployment level.

### 2.3.4/ The IP Reuse Cycle

In the previous section, we have discussed some of the most important concepts on IP reuse. We have also provided a very general IP taxonomy, based mainly upon on their degree of flexibility regarding parameterization. However, when using a hardware description language such as VHDL, usually all the IPs are reused in the same manner, by following a reuse cycle methodology [93], which consists of four main steps, as depicted in Figure 2.8

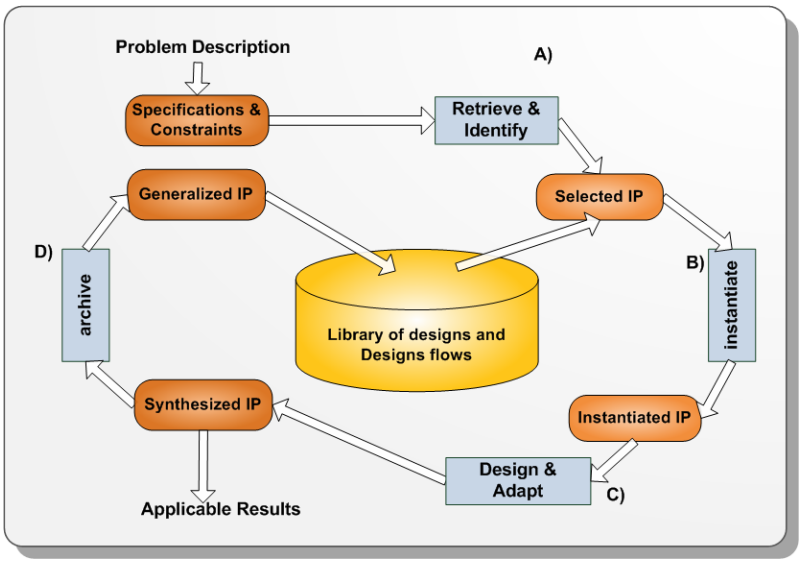


Figure 2.8: Typical IP Reuse Methodology

- **Retrieval (a):** An IP block is selected corresponding to the respective specifications. This selection can be made manually or automatically.
- **Instantiation (b):** If an IP has been selected, the necessary designs are instantiated. This implies the interconnection of the IPs, their parameterization and in some instances, customization via controlling parameters. This instantiation can be performed at several levels of abstraction, depending on the intended flow.
- **Design (c):** Once the design or top-level netlist has been obtained, a design flow is applied. This means that a target technology has to be chosen in order to obtain a synthesized description, optimized for the chosen back-end.
- **Retain (d):** In some instances, a design can be reused in the form of a hierarchical component. Then, its specification must be generalized (for reuse-relevant parameters and creating and interface wrapper for plug and play, are two examples). Moreover, a design flow (in the flow of tool specific parameters, directives or options) must be associated before it is stored in the library.

This reuse cycle flow can then be applied to any methodology, being a conventional VHDL-based approach or through high-level synthesis techniques, (e.g. platform-based design, MDE), if certain conditions are met and by taking the following steps [94]:

- The standardization and analysis of specifications (steps a,d on Figure 2.8). This implies applying constraints in the way the IPs are coded, the use of common interfaces and synthesis procedures, and on the other hand, the analysis of the specification with regard to correctness, the feasibility of synthesis, and reusability. The standardization also applies for the IP parameters specification, which provides

many opportunities for defining automatic procedures during the integration phase, and through graphical user interfaces (GUIs), their parameterization and customization (i.e. soft IPs)

- The design of a platform (at multiple levels of abstraction) by using pre-existing IP blocks from the library (design by reuse, steps a,c, and d). The design of the platform requires the use of heterogeneous IPs, and thus, defining an IP taxonomy becomes a necessity; for instance, IP can be divided in groups such as communication, video processing, mathematical operations, compression, amongst others. Such a classification is reasonable, since each class of IP requires different kind of attributes for its characterization. Then the creation of platform through composition consists, as mentioned before, on instantiating a set of components (adding parameterization information) and in defining their interconnections.

- The development of parameterized procedures (e.g. scripts for steps b,d) for synthesis or compiling associated IP drivers depending, eventually, on other controlling parameters (customizable IPs) through constraints propagation. This information must be associated to the IPs in the library, and used along an appropriate design flow for obtaining the desired results (e.g. logic synthesis, drivers compilation, test-bench generation)

- The development of parameterized data sets (e.g. stimuli) for verification purposes (steps c,d). These data sets can be used for facilitating the IP evaluation, and prove whether the functionality of a piece of IP meets the requirements.

- The design of reusable IPs (design for reuse, step d), for IPs not already in the library [95]. Through the definition of standard coding techniques and common interfaces, the automatic integration of IPs is greatly improved; then, it is important that new IPs undergo similar procedures if they are not suitable in their current form. Making reusable components also encompasses the use of a component-based approach by using well accepted bus interfaces such as IBM's CoreConnect [81] or ARM's AMBA AXI [82], for promoting easy of use plug-and-play methodologies, or in the case of MDE methodologies, for abstracting the complexity of inherent in describing an interface.

- The database design for IP libraries (data management, step d). The use of IP databases and the associated retrieval mechanisms, implemented in many CAD tools, provide means for obtaining parameterized specifications, a hierarchical selection/navigation of the IP information, and support for the creation of GUIs that facilitate creation of complex systems employing design by reuse techniques. However, standardization in the design of the IP and system descriptions need to be attained so that IP reuse and design by reuse can be effectively exploited.

As the last point makes it clear, the standardization of IP interfaces and specifications is not enough, given that the functionalities of the IP and how they are intended to be used cannot always be inferred from the interface and the parameters themselves. Moreover, the way IPs have traditionally been provided by IP vendors (VHDL code or pre-synthesized netlists) does not promote design by reuse effectively: the IP data has to be extracted by humans, and the interconnection performed manually, which is a very prone to error process (increasing in many cases the productivity gap).

The productivity gain looked after by IP reuse methodologies can be further improved if the processing of IP data (e.g. interfaces, parameters, scripts directives) is automated. To be more precise, a flow and flexible methodologies (supported by CAD tools) are needed that can process multiple IP sources and which can effectively transform their associated data into an SoC platform. This flow and methodology need to address automatic and correct integration of IP blocks, as their parameterization, verification, and synthesis. Automating the processing is important in order to be able to quickly handle updated versions of subsystems and generate consistent outputs.

This problematic will be addressed in the next section, and the ideas discussed in it will serve as a launching pad for introducing the contributions of this thesis work.

## 2.4/ METADATA-DRIVEN SOC IP REUSE AND INTEGRATION

In the Electronic Design Automation (EDA) domain, the management of data and metadata plays an important role in the context of system-level design [96]. Many design flow methodologies use a pattern which consists on assembling reusable sub-systems and/or Intellectual Proprietary blocks (IP) to construct complex system platforms. This design process relies heavily on metadata. The formalisms used to describe system architectures within this process are often referred to as models of architecture.

Over the last decade, special architecture-oriented languages have been developed in order to support these formalisms; they are known as architecture description languages (ADL) [97] . Many projects such as Colif [98] and OpenFPGA [99] have built the syntax of their ADLs on the XML [100] technology stack, as a means for metadata management and its manipulation. XML has attracted significant attention over the years as a flexible way to define structured information exchange or languages in various domains [101, 102].

As one of many mark-up languages, it trades off brevity, readability, and human comprehension for ease of programmatic parsing and use in automate tools and systems. XML is a meta-format from which particular languages or notations can be constructed either using a DTD (Document Type Definition) or XML schema. Therefore, XML can be used for classifying information in electronic databooks, which have been used in many domains, such as in Electronic System Level design [103]

### 2.4.1/ The Metadata-driven Component Composition Framework

The use of metadata in the context of SoC design has been an active area of research in which tools, globally known as Metamodeling-driven Composition frameworks (MCF) [104] have been developed for addressing several issues in SoC development, such as IP reuse and automatic platform composition. These techniques have orthogonal concerns with MDE approaches and Platform-Based Design, and have proven successful in facilitating tasks related to IP reuse, which are described as follows:

- **System requirements specification:** A medata-based approach lets the designers specify the system requirements as a platform or as an abstract template. These templates facilitate the visualization of data intended for automation, which can be simultaneously modified (for IP parameterization or customization) and then used in subsequent stages of a design flow (i.e. synthesis through model transformations).
- **IP metadata for reuse:** The IP metadata necessary to enable reuse includes compositional characteristics at both the structural (ports, widths, parameters) and the behavioural levels (usually by pointing to the implementation artifacts of the IP, but also the design flow directives used for obtaining synthesizable results). This metadata is mined from the IP library through automated extraction techniques, at multiple levels of abstraction.
- **IP metadata representation :** This meta-information is represented using XML schemas and populated in XML data structures to enable processing through XML parsers and promoting highly effective design automation capabilities to the flow in which they are inserted.
- **IP selection and composition:** Metadata in the XML component description can be used to select IP implementations that structurally and behaviourally match the components in the visually defined platform. This is especially important when exploring multiple system configurations, since it facilitates design space exploration by abstracting low-level details.

In the discussion that follows, we aim at drawing an analogy between the MCF paradigm and those efforts in MDE methodologies for SoC. For both approaches to be of any use, the syntax and semantics of the employed visual composition model must be captured into a metamodel [105]. This corresponds to the architecture and deployment models in current MDE methodologies, in which the meta-models can capture the platform information at several levels of abstraction (e.g. TLM, RTL). As explained previously, in MDE approaches, the deployment phase associates the abstract high-level representations of the IPs with those the artifacts used to implement their functionality. Moreover, the deployment models are usually target specific, meaning that from this phase it is possible to generate an executable model as the end product.

These meta-models can be developed using a combination of UML and the Object Constraint Language (OCL) [106]. Then, a design environment lets a user/modeler to visually construct a platform conforming to the meta-model underlying restrictions [107]. This step is shown in the top block (a)) on Figure 2.9, which is labelled as Modeling Front-End. The entities allowed in the Component Composition Model at various levels of abstraction such as RTL and Transaction Level (TL) represent hierarchical descriptions [108] . The visual model and the behavioural properties are converted into an internal XML representation (b), XML-IR on the figure) which is used for design automating the selection, instantiation, and parameterization of the IPs that comprise the model, which must conform to a platform metamodel.

The most important feature of an MCF approach for co-design, is the existence of an IP library of RTL or SystemC cores, as depicted on the block c) on Figure 2.9. The reflection mechanism is used to identify, extract, and represent IP-specific meta-data contained in the IP XML structures. For RTL components, the IP reflection extracts two kinds of meta-data: information about ports and interfaces for interconnections, as well as data-types and other parameter values for customization/parameterization, when creating an instance of the component. This meta-data is obtained from mining the IP descriptions or by filling XML structures with information about the IP, manually or automatically [109],which also conforms to a metamodel (in this case an IP Block metamodel).

This XML representation of the components, along the system representation, enables IP reuse and facilitates system composition, but the correctness of the selected IPs and of the obtained platform must be ensured. Checkers must be implemented in both phases [108], but the IP selection problem can be alleviated by providing ID tags capabilities in the XML component descriptions. The XML intermediate platform representation is known in the literature as an introspective architecture [110], which is one that lets the system query its own state and topology. This means that the XML-IR representation can be used

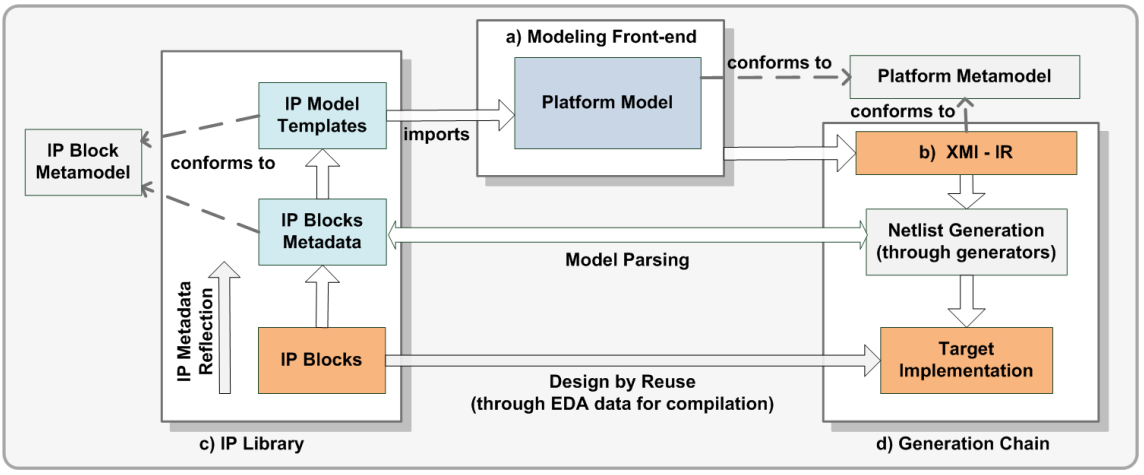


Figure 2.9: Metamodeling-driven component composition framework (MCF) design flow

for parsing the metadata contained in the XMLized IP representations (usually low-level details such as ports and parameters). Along a generator, such a netlister, it helps in the creation a top-level netlist (i.e. RTL) which points to the actual implementation files (for instance, HDL modules). Moreover, EDA information also contained in the XML structure is used for generating the desired target implementation (which depends of the targeted back-end or design flow), such as FPGA netlists. This is accomplished by a generation chain (block d)), which imports the XML-IR and executes a set of programs for producing the desired artifacts.

### 2.4.2/ Interoperability Issues

Although metamodeling for SoC has shown great promise during the last decades [51], an important handicap preventing its adoption is the non-standardized nature of the platform XML intermediate representations used in most approaches. Different methodologies target different back-ends, and make use of specialized metamodels for the platform description, leading to XML intermediate representations that might be effective for solving the problem at hand, but which cannot be exploited by other tools or design flows. The lack of support for achieving interoperability has become a major hurdle for promoting the much required productivity gains in the development of complex embedded systems [111]. This is especially true in the context of IP reuse and design by reuse: if each tool or design flow makes use of its own intermediate representation, a major effort is required from the perspective of the different actors in the SoC industry (IP providers, system integrators, CAD tool developers) to make third party IP blocks and their associated representations to work in their own design flows. Furthermore, this lack of standardization makes it more difficult for the academia and the industry to cooperate, since the efforts proposed by the former have been largely ignored by the latter due to the aforementioned issues.

These challenges have been addressed by the academia in recent years, with many efforts looking to promote standard libraries of IPs, such as the OpenFPGA CoreLib [99]. However, it was until the creation of the SPIRIT Consortium [112], and later, the introduction of the SPIRIT Specification, that efforts in the interoperability of IP representations by using metadata began to gain traction, strongly motivated by the EDA companies promoting the endeavour. The efforts of the SPIRIT consortium were brought to fruition by the adoption of the specification into a full-fledged standard, the IEEE Standard for IP-XACT, Standard Structure for Packaging, Integrating, and Reusing IP within Tools Flows [88].

The standard describes a set of XML schemas used to document IP meta-data for SoC integration, in a structured manner. Several industrial cases studies have demonstrated that its adoption facilitates the configuration, integration, and verification in multi-vendor SoC design flows [113, 114]. Furthermore, IP-XACT also provides ways to automate the design flows where different tools are used. It is meant to be used by IP providers and system integrators and all major EDA vendors as way to standardize the access to IP

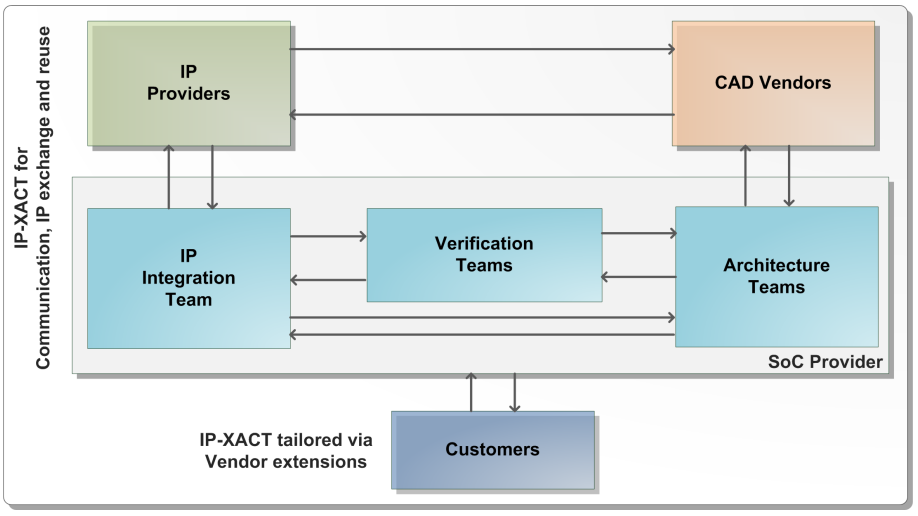


Figure 2.10: IP-XACT in an complex IP ecosystem

information, as depicted on Figure 2.10, where arrows represent flow of XML/VHDL objects. The IP-XACT standard has been slowly adopted by FPGA vendors as well (in no small part due to the complexity of accommodating procedures, tools and IP representations to operate with the standard) such as Xilinx, with the very recent introduction of the Vivado Tool Suite [115].

The goal of the IP-XACT standard is to provide a standard XML abstraction of hardware components, whatever the language. Hence, by using a standard XML representation, IP and system data can be seamlessly exchanged among the major players of the SoC industry, promoting IP reuse and alleviating the aforementioned interoperability issues. Moreover, the IP-XACT description can also be exploited by customers through specific extensions for tailoring the IP descriptions to their particular needs. Since its original inception by the Spirit Consortium, the IP-XACT standard has gathered enormous interest from the industrial and scientific communities as a means to overcome the complexity of system integration and tool interoperability. A thorough description of the IP-XACT standard is outside scope of this chapter, but the most important notions will be addressed in the next section. Subsequently, full details about the different components of the standard will be provided in the main contribution chapters of this manuscript.

### 2.4.3/ THE MERGE OF TWO WORLDS: IP-XACT AND MDE

The use of IP-XACT is of particular interest in the development of metadata-driven tools for the conception of System on Chip, both industrial and academic. The use of a common intermediate representation for IP and platform descriptions is then a positive development for the industrial and scientific EDA communities, but also for the ongoing research in System Level Design and MDE for SoC; this is due to the fact that the interest of using IP-XACT has been recognized beyond the industrial circles where it was originally devel-

oped, particularly as a means for closing the gap between the academia and the specific needs of the SoC industry [116].

In recent years, there has been a widespread interest in merging the best from MDE methodologies and metadata IP reuse approaches based on IP-XACT. Initial attempts at coupling UML MARTE with IP-XACT have been gaining traction in recent years, as first presented in a general manner in [117]. Subsequently, an important development towards the integration of IP-XACT in MDE-based methodologies was fully detailed in [118], where one the first metamodels for IP-XACT using UML MARTE were presented. This metamodel has served as a blueprint for providing a front-end for building platform composition frameworks, exploiting the transformation capabilities inherent to MDE in the automatic generation of different back-end platforms from high-levels of abstraction. These efforts will be discussed in more detail in the next chapter, but we outline how the IP-XACT standard was conceived for being integrated into SoC tools in the next section.

As outlined at the end of the previous chapter, in this thesis we propose the use of a component based approach, in which components are assembled from an IP library at high-levels of abstraction, and then through model transformations, a SoC platform is created. Since only Xilinx provides DPR, we make use of the Xilinx XPS tool, which supports IP components making use of CoreConnect and AMBA AXI bus interfaces. In this way, we provide plug-and-play capabilities to our methodology. However, for attaining these goals in the context of an MDE methodology, the IP blocks information needs to be available at high-levels of abstraction (at deployment level, for being able to generate system description) and thus, an IP reflection mechanism must be provided so the IPs metadata can be visualized and exploited in the UML models. Moreover, an intermediate description between the UML models and the back-end must be deployed, and for that, we make use of the MCF paradigm, in which MDE approaches for SoC can be circumscribed. For alleviating the interoperability problems discussed above, the IP-XACT is used as our intermediate representation. The Xilinx specific models can then be obtained from the IP-XACT description through model transformations, using a transformation engine. The rationale behind these choices will be fully detailed in the second part of this thesis.

## 2.5/ THE IEEE 1685 IP-XACT STANDARD

The objective of this section is to summarize the capabilities of IP-XACT as a method for IP Reuse and Electronic Design Automation (EDA). IP-XACT approach to reusing IP-cores is through a design flow that utilizes metadata exchanged in a design-language neutral format. This enables IP configuration and integration by utilizing metadata exchanged in XML format between tools and SoC design flows [119]. SPIRIT provides the API standards that should be implemented to export the design and configure the actual IPs to create the executable code. However, the designer is required to specify the IP-

explicit meta-information, which is a tedious task and limits the rapid integration. However, this step has been highly automated by several vendors, and it is now possible to obtain up to 60% of the required information of an IP, from sources such as documentation, C header files and VHDL/Verilog descriptions.

IP-XACT is structured around the concept of IP re-use. Electronic Design Intellectual Property, or IP, is a term used in the electronic design community to refer to a reusable collection of design specifications which represent the behaviour, properties, and/or representation of the design in various media. The name IP is partially derived from the common practice of considering a collection of this type to be the intellectual property of one party. Both hardware and software collections are encompassed by this term. In the context of IP-XACT, these collections may include the following:

**a) Design objects** - These are referred as artifacts in UML, but they can also described other models. Some examples of these can include the following:
1. TLM descriptions: SystemC and SystemVerilog
2. Fixed HDL descriptions: Verilog, VHDL
3. Configurable HDL descriptions (e.g., bus-fabric generators)
4. Design models for RT and transactional simulation (e.g., compiled core models)
5. HDL-specified verification IP (e.g., basic stimulus generators and checkers)

**b) IP views** - This is a list of different views (levels of description and/or languages) to describe the IP object. In IP-XACT v1.4, these views include:
1. Design view: RTL Verilog or VHDL, flat or hierarchical components
2. Simulation view: model views, targets, simulation directives, etc.
3. Documentation view: Standard, User Guide, etc.

IP-XACT XML meta-data descriptions provide a standardized way of collecting much of the structural information contained in the file sets. IP-XACT also can contain the information that identifies the appropriate files included in a collection to be used for different parts of the design process.

### 2.5.1/ IP-XACT MODELING OBJECTS

IP-XACT defines four central elements for the description of IPs, which are bus definition, abstraction definition, component, and design. These elements are depicted on the top of Figure 2.11, as well as their interactions in each of the steps of a IP-XACT based design flow. A bus definition describes a bus, such as the PLB, in terms of its most basic properties such as the supported number of masters and slaves. It basically serves as a reference for providing more detailed descriptions of the same bus at different levels of abstraction.

1. An abstraction definition provides a complete description of the set of interfaces of a bus at a specific level of abstraction, e.g., TLM (Transaction Level Model). For this, all interfaces, such as the master and the slave interfaces are described in terms of the respective ports. A port is a distinct interaction point on the bus, which may be a wire port carrying logic values, or a transactional port, typed by a high-level interface. The latter typically refers to a TLM port.

2. A component packages an actual, potentially configurable, IP-core. It may contain multiple views at the core covering different aspects or levels of abstraction. In particular, a component describes the bus interfaces exposed by the core, and the mapping between component ports and the respective ports from the corresponding abstraction definition. In addition, model parameters that enable configuration of IP core itself can be defined, e.g., the actual width of a bus interface.

3. A design describes an actual design as a set of interconnected component instances. These connections are usually made between bus interfaces, but component ports may also be connected directly (ad-hoc connections). Furthermore, the model parameters can be individually configured for each instance. If two components are connected through bus interfaces at different abstraction levels, e.g. one RTL and one TLM, an abstractor can be inserted between the two components to act as an interface adaptor.

4. Finally, the Tight Generator Interface (TGI) enables external tools to perform operations related to an IP-XACT specification such as configuration or model generation. For this, it defines a comprehensive set of SOAP methods for retrieving and modifying all information in a SPIRIT XML file. It is required to be implemented by any IP-XACT conforming design tool.

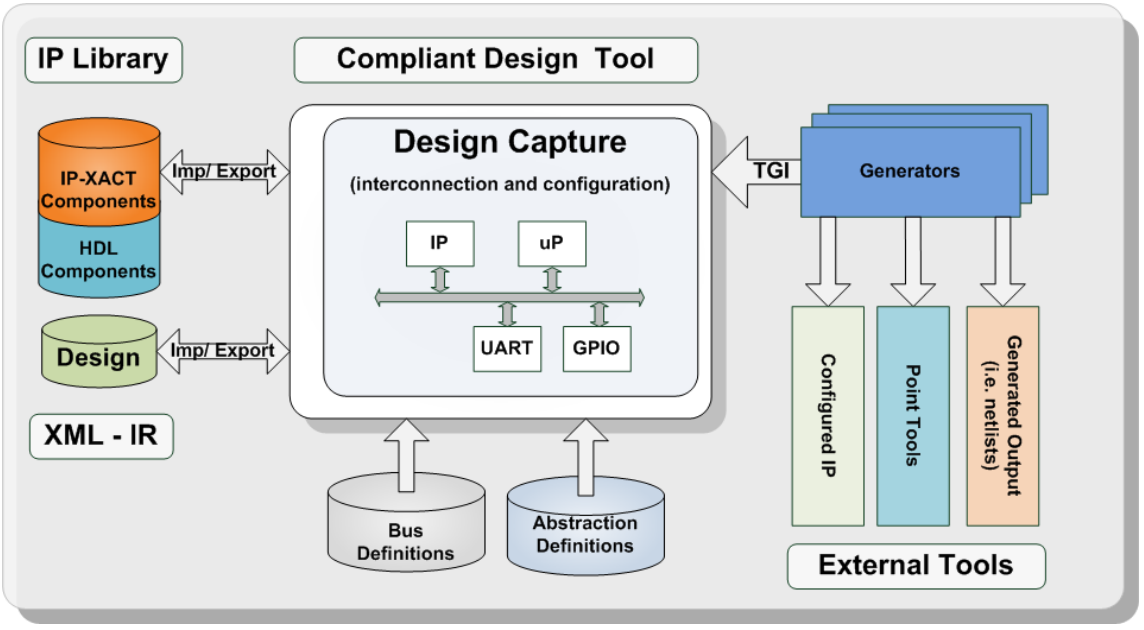


Figure 2.11: IP-XACT Design Environment and supported XML schemas

### 2.5.2/ THE IP-XACT DESIGN ENVIRONMENT

The second part of Figure 2.11 provides an overview of the IP-XACT approach for SoC design, known as the Design Environment (DE). A design environment enables the designer to work with an IP-XACT design through a coordinated front-end and an IP database. These tools create and manage the top-level meta-description of system design, and may provide two basic types of services: design capture, which is the expression of design configuration by the IP provider and design intent by the IP user; and design build, which is the creation of a design (or design model) to those intentions.

Reusable cores are defined as components in XML and automatically imported into an IP-XACT compliant DE (Packaging). The designer selects IP from the component library and creates complex SoC designs with limited effort (Integration). After composing the design, generator chains are interfaced with third party tools to verify and synthesize the design (Tool Flow). The design approach of IP-XACT has many advantages for a project of the kind of MDE for several reasons. First, it is a standard adopted by the IEEE (IEEE 1685) which means that is expected that IP Providers, IP Integrators and EDA vendor alike will create IP-XACT compatible IPs, which will facilitate enormously the reuse of IPs. However, by creating a design flow that conforms to IP-XACT, tedious tasks like IP wrapping can be highly automated, while still being compliant with tools offered by different vendors.

Moreover, IP-XACT facilitates the interconnection of complex IP using bus definitions defined in XML. An IP-XACT bus definition describes all the ports of a standard bus or IP interconnection scheme. Similarly, bus definitions of standard protocols have been or are being created, which will facilitate the exchange of applications conceived for a system to a new environment in a shorter time and with less effort. Previously, the integration of complex systems has been one of the most complex and prone to error parts of the process, and nowadays this have been completely automated by IP-XACT enabled tools. This capability of IP-XACT has clear advantages in the context of FAMOUS approach, in which very complex SoC systems have to be defined.

However, the system design tool set only defines a DE where the support for conceptual context and management of IP-XACT metadata resides. This means that the IP-XACT specifications make no requirements upon system design tool-architecture or a tool internal data structures. To be considered IP-XACT v1.4 enabled, a system design-tool shall support the import/export of IP expressed with valid IP-XACT v1.4 meta-data for both component IP and designs, and it needs to support the tight generator interface (TGI) for interfacing with external generators (to the DE). This has been done by EDA vendors such as Magillem [120] and Mentor Graphics [121], and we propose to make use of these tools to facilitate the conception of the IP-XACT based Design Deployment Flow.

## 2.6/ Discussion and Conclusions

From the previous section it must be clear by now that the IP-XACT design environment supports all the features required in metadata-driven component composition frameworks. First, the modeling front-end is represented by the design environment, as depicted on Figure 2.12 a). The XMLized IP-XACT components are obtained through IP meta-data reflection and stored in an IP library, containing both the RTL or TLM implementations and their XML counterparts. These objects are imported into the design environment, in which an introspective architecture can be created by visually instantiating the components from the library and interconnecting them; parameterization can be performed through the use of software APIs supported by the DE, and finally, an executable platform (i.e. described in RTL) can be obtained through generators supported in the standard (interaction shell). It is not the intention of this section to describe in detail how these steps are performed; more information will be provided in the second part of this manuscript. It suffices to say that the creation of the IP-XACT standard has benefited from years of research in both the academia and the industry, and its advantages in the creation of the hardware descriptions of the SoC has been successfully proven.

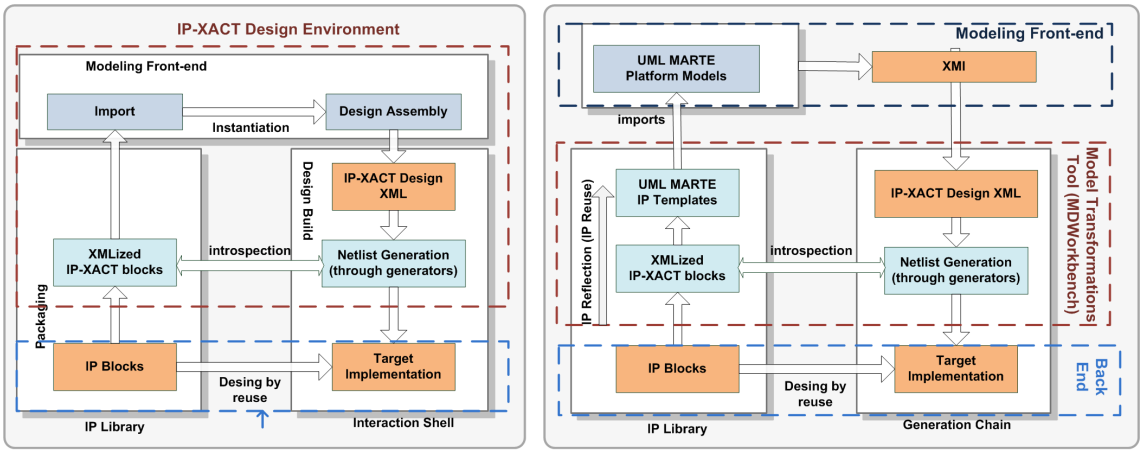


Figure 2.12: A comparison of MDE for the conception of FPGA-oriented SoC

Nevertheless, the IP-XACT standard, in its current version, does not support all the modeling aspects that approaches such as the MDE look after. Examples of these modeling endeavours are the conception of the application through Models of Computation, Schedulability or Design Space Exploration, and other models such as configuration and state machines for describing the system behavior. However, the inclusion of IP-XACT in an MDE-based design flow can alleviate the interoperability issues of many current methodologies, as discussed in Section 2.4.2, greatly benefiting from the standard nature of the intermediate representation of the IPs and the SoC platform. However, in order to integrate IP-XACT in an MDE methodology, two aspects must be ensured. First

an IP reuse mechanism through IP reflection that permits visualizing IP components at high-levels of abstraction, so a platform can be composed from these IP templates, as depicted on Figure 2.12 b). Secondly, a mechanism for transforming the UML MARTE platform specification (stored as an XMI IR) into its IP-XACT counterpart, a design object description. As discussed in Section 2.2, the model transformations are the central part of any MDE methodology; in the scenario depicted on the figure, a Model Transformation Tool (MTT) would permit the creation of the XMLized component descriptions, and on the other hand, the transformation from the XMI platform representation into an IP-XACT corresponding object. Furthermore, the MTT is also exploited for transforming the IP-XACT design description into the target implementation model, effectively decoupling the high-level models from the intended back-end.

This approach has already gathered attention from the academia, as stated in previous sections, with different purposes. Some of the related works in combining UML/MARTE and IP-XACT would be thoroughly discussed in the next chapter, using the MCF paradigm discussed here as a launching pad. However, none of these approaches takes into account DPR systems nor completely incorporates the IP reuse and design by reuse requirements discussed in this chapter. These are the main contributions of this thesis, as well as the model transformations just described, for supporting a component-based approach using Xilinx Platform Studio, which is the only SoC targeted CAD tool supporting partial reconfiguration. A the end of the next chapter, the complete framework summarizing the discussion of Chapter 1 and Chapter 2 will be provided, and then discussed in detail in the second part of this thesis.

# 3

# MDE-BASED METHODOLOGIES FOR THE CREATION OF SOC

## Contents

## 3.1/ INTRODUCTION

The use of model based approaches for co-design has been thoroughly discussed in different works. Comprehensive overviews can me found in [46] for UML Platform-based Design, as a means for improving the design flow of SoCs in [50], and in a more general framework, for the design of electronic systems in [48]. The approaches vary in terms of the chosen modeling language (UML, SysML), the intended back-ends and purposes (i.e. SystemC for Design Space Exploration) and moreover, the use of specific UML profiles for dealing with the complexity of such endeavours. Nonetheless, the FAMOUS methodology, in which our work is circumscribed, deals with extending the UML MARTE profile with Dynamic Partial Reconfiguration concepts, and furthermore, aims at providing a framework in which the generation of such platforms can be performed from high-level models. Therefore, in the subsections that follow, we focus our attention in methodologies that make use of UML for SoC or UML MARTE; however, it must be noted that the proposed framework could potentially be used in other contexts. The focus of this thesis is to provide these mechanisms to FAMOUS by defining a suitable design flow, and standard intermediate representations for the IPs and the platform, promoting tool interoperability and remaining flow/technology agnostic. By using the IP-XACT standard, an effective decoupling of the platform from the front-end can be achieved, concentrating solely in the desired back-end through model transformations.

The FAMOUS framework differentiates from previous MDE approaches in the sense that the creation DPR systems is its main focus. Nevertheless, the creation of the DPR platform, as mentioned in Chapter 2 is closely related to IP reuse and metamodeling composition frameworks. Therefore, we have taken inspiration from MDE methodologies that make use of UML (in general) or other UML profiles such as MARTE, for the creation of SoC platform models; some of the approaches have different aims, such as design exploration (using SystemC), whereas others, whilst targeting the creation of FPGA SoC platforms, do not make use of standard intermediate representations for the platform model. Moreover, only some approaches have specifically targeted DPR systems, but without explicitly promoting IP reuse.

In the next section we introduce a comparative framework, based on the MCF introduced the previous chapter, for classifying the efforts carried out in recent years in the modelling of SoC platforms using MDE. Then we embark in a description of each of these approaches, and we highlight its advantages and shortcomings; we divide the discussed approaches into two broad categories: Platform Modeling using UML/MARTE Profile using proprietary meta-models (Section 3.3), and secondly, Platform Modeling using UML/MARTE in tandem with the IP-XACT Standard as IR (Section 3.4). This analysis is then used to introduce the contributions of this thesis in terms of IP reuse and Automatic DPR System generation. The chapter ends with a general conclusion, serving as a planning for the rest of the thesis.

## 3.2/ A FRAMEWORK FOR COMPARING MDE APPROACHES FOR SOC

In this section we present a framework that will be used, as a launching pad, for carrying out a comparative study of the state of the art in the design of SoCs using MDE methodologies. For this purpose, we will make use of a simplified version of the Model Composition Framework first introduced in Section 2.4.1. We have kept the most relevant constituent elements for the comparison, as depicted on Figure 4.14. We proceed at detailing the phases as follows:

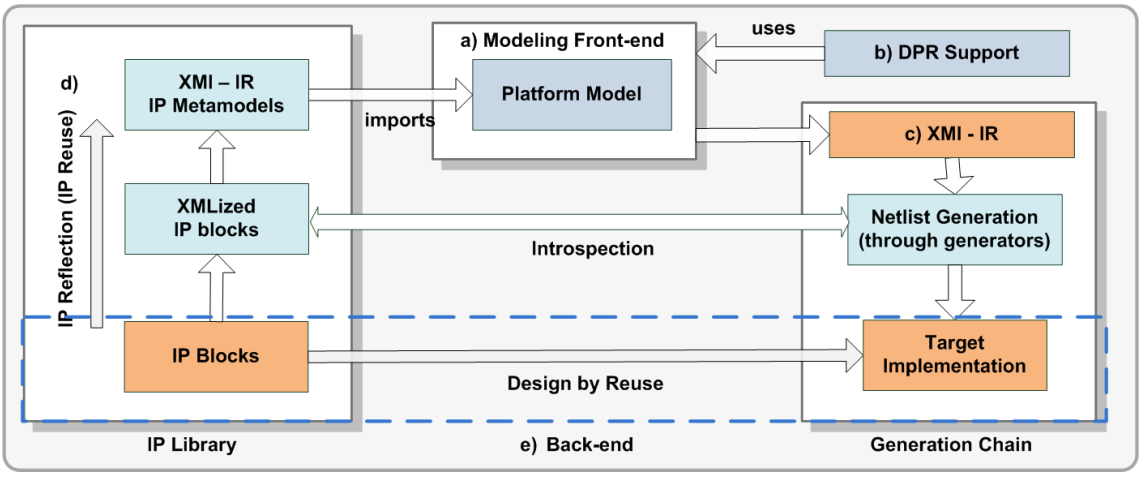


Figure 3.1: A comparison of MDE for the conception of FPGA-oriented SoC

- **a) Modeling Front-end:** The FAMOUS approach is based in UML MARTE, but some interesting MDE approaches based solely in UML or UML 2.0 have been proposed. These approaches make use of specialized meta-models and in some instances, their own extensions in the form of profiles for the modeling of real-time embedded systems.

- **b) Partial Reconfiguration Support:** The second classification criteria takes into consideration whether or not the proposed methodologies support the modelling of DPR SoCs. Not all the methodologies target the modelling of FPGA-based reconfigurable systems, concentrating more in DSE, but regardless of the chosen abstraction level, the requirements often overlap.

- **c) Used Intermediate Representation:** This is the most important aspect that we address in this work. By comparing how the MDE in the past have carried out the meta-modelling of the platform description, we can contrast how much IP-XACT is helping to build new design methodologies that are not only based on a standardized XML-based intermediate representations, but that can also benefit

from the unified API model, and from previous experience in the industry, while promoting IP reuse and tool interoperability.

- **d) Metadata Reflection for IP Reuse:** Many MDE-approaches do not support IP reuse through meta-modelling. Instead, many of those methodologies promote an approach in which the information of the IPs is directly annotated onto the model (often by hand) by using comments. This approach does not take full advantage of the IP meta-data reflection mechanisms described before; therefore, creating a platform at high-levels of abstraction requires analyzing the code (i.e. for parameters), which in our opinion, contravene the very idea of what MDE stands for. Even if IP reuse is promoted through IP meta-model representations, the non-standardized nature of these meta-models does not promote design by reuse. Then, concurrently with the classification criteria c), we discuss which methodologies exploit IP-XACT as an effective means for IP reuse, at several levels of abstraction.

- **e) Supported Back-end(s):** Finally, we classify the discussed methodologies depending upon the targeted back-end. Most of the approaches to be described in the following sections target SystemC, without taking into consideration FPGA implementations of embedded systems. However, these approaches have provided us with some insightful hints into the integration of IP-XACT in our methodology. Nevertheless, some of the discussed approaches target the generation of RTL code, or as in our case, a second intermediate representation used by third-party tools, such as Xilinx, for describing their systems and making more amenable the design process.

## 3.3/ Platform Modeling using UML MARTE

### 3.3.1/ The MoPCoM Methodology

Several works have tackled the use of UML/MARTE methodologies in SoC design, specifically at the deployment phase. An interesting approach has been carried out by the MoPCoM project [122], which targets the modeling of FPGA based embedded systems, using UML MARTE [123]. In their approach, three models for the system are defined: functional, platform, and allocation. The allocation maps the behaviour onto the platform components, and then HW/SW partition is performed. As depicted on Figure 3.2, they make use of two back-ends: SystemC for simulation purposes, and a VHDL code generation tool to target behavioural modeling. The authors make use of MDA techniques for generating both system representations. As with many other methodologies, they build the platform by using MARTE elements contained in the Hardware Resource Modeling (HRM) sub-profile, and then use the allocation mechanism provided by the standard to map the application onto the platform components.

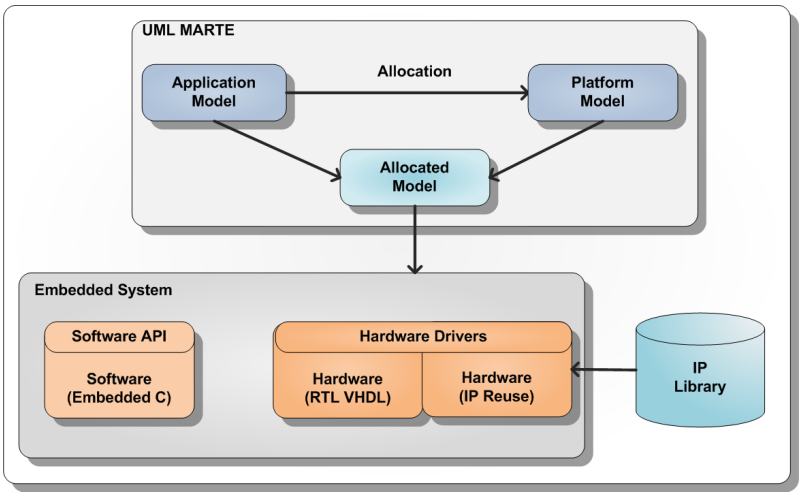


Figure 3.2: Detailed Modeling Level in the MoPCoM Approach

The authors claim being able to generate both VHDL skeletons from the platform models, and also the generation of the Microprocessor Hardware Specification (MHS), an intermediate representation used by Xilinx tools for creating the top-level system descriptions, which is subsequently transformed for FPGA implementation, as depicted in Figure 3.3. However, in both papers, as with other methodologies, it is not clear how IP reuse is performed in their approach; specifically, no IP library is explicitly used nor an intermediate representation or IP meta-model is defined. Therefore, it is not clear how they link the blocks in the platform model to their low-level (i.e. VHDL) implementation counterparts.

This handicap has been somehow addressed in [124], where the authors managed to add limited IP-reuse capabilities to their approach by creating an ⟨⟨Xilinx IP⟩⟩ stereotype, which enables them to effectively link the UML part instances of the platform model to the low level IP descriptions used by EDK. The Xilinx IP stereotype contains information about the version of the IP block the type of component (e.g. Processor, Bus, Bridge, IP, Peripheral), and permits to add parameterization information to the part instance in UML by defining a ⟨⟨Parameter⟩⟩ dataType. Using this information, it is then possible to create the MHS description that contains the components instantiations, with their respective interconnection and parameterization information. However, making a meta-model for Xilinx-oriented platform descriptions ties the methodology to a specific design flow, not promoting interoperability.

Moreover and despite of these efforts, IP reuse information has to be annotated directly in the model, making the process difficult to automate, and promoting IP reuse only partially. This is due to the fact that the parameters of many IPs might be dependent on other parameters or constrains in the design, and therefore, this approach is not suitable for customizable components. Moreover, the profiling mechanisms used in the methodology tie the IP descriptions to a specific technology, making it difficult for targeting other FPGA vendors or generating VHDL code directly, because not enough information is available

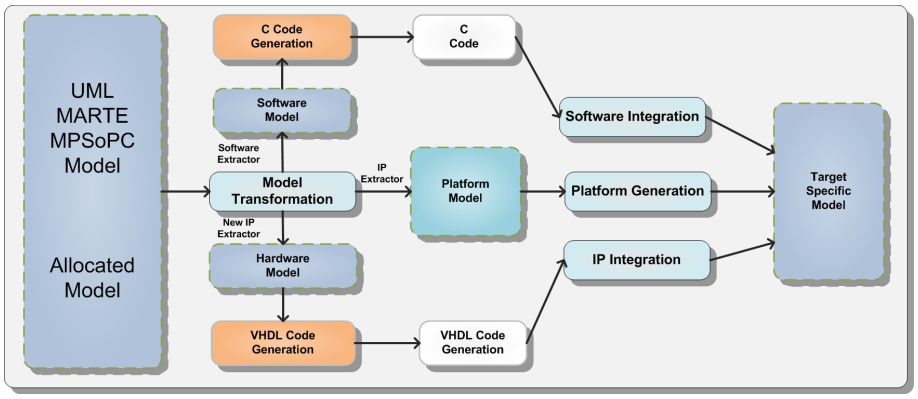


Figure 3.3: The unified model is used as input to several model transformation tools, where code generation is performed for each intermediate model

by using few parameters. As we will discuss is the next sub-section, this problem can be addressed by using IP-XACT, which contains rich information about the IP blocks and their implementations, while at the same time, permitting to choose how this data is to be used both at the frond and back-ends of a given methodology.

Finally, the works just discussed do not take into account the modeling of complex dynamically reconfigurable IPs. However, some attempts at modeling DPR systems have been introduced in [125], but also lacks IP reuse as in previous efforts.

### 3.3.2/ THE GASPARD APPROACH

Several other works explore embedded system modeling using UML, but only a few explore dynamic and partial reconfiguration capabilities. In [126], authors detail a dynamic reconfigurable system by extending MARTE Profile with specific stereotypes. Their approach is developed in the GASPARD [72, 127] framework, a design environment that provides designers with some of 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. The proposed design framework is referred to as GASPARD (Graphical Array Specification for Parallel and Distributed Computing), and its capabilities in terms of the supported back-ends, are depicted on Figure 3.4

In GASPARD, the abstract models resulting from the high-level modeling phase are enriched with specific information according to the target technologies. Finally, different automatic refinements from the higher abstraction level are defined, according to the Model-Driven Engineering (MDE) paradigm, towards lower levels for various purposes: simulation at different abstraction levels with SystemC [128, 129], and hardware synthesis with VHDL [130]. They make use of a custom Deployment Meta-model [131], which in fact provides several mechanisms to link low-level implementation (e.g. component

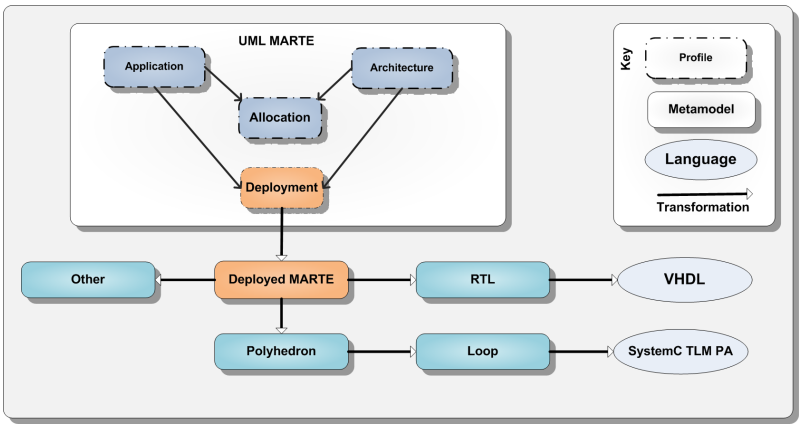


Figure 3.4: A global view of the Gaspard2 environment for SoC design

interfaces, configuration parameters, hardware, and software implementation files) to the high-level models, as shown in Figure 3.5

For this purpose, they have introduced the concept of Virtual IP as a generic wrapper that captures various implementations of a given elementary component (either software or hardware), independently from the usage context [132]. A Virtual IP is implemented by one or several IPs, each one used to define a specific implementation at a given abstraction level and in a given language. 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 used IP is selected by the SoC designer by linking it to the elementary component through the Implements dependency.

However, authors do not detail the specifics to link the MARTE model to this level nor describe how the meta-model is exploited to move to lower levels of abstraction. More-

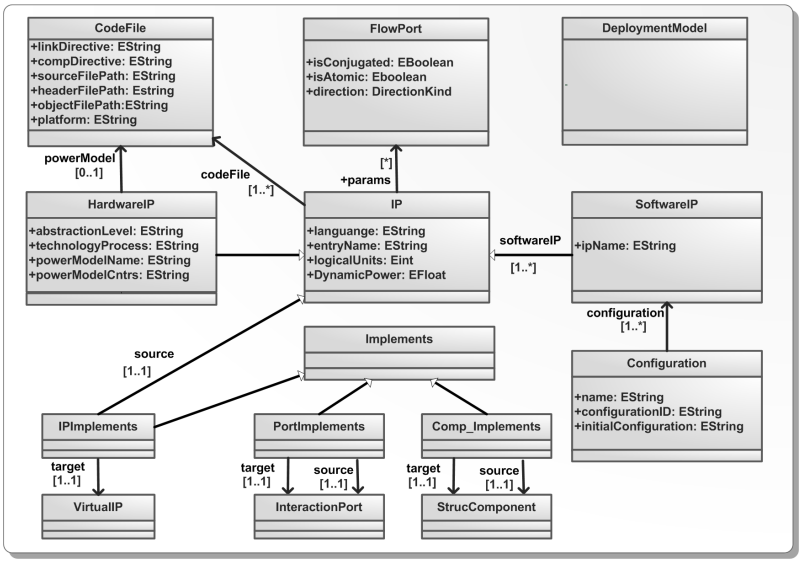


Figure 3.5: Global overview of the MARTE integrated Deployment package

over, the proposed meta-model, as in other approaches, it is too methodology dependent, lacking sufficient generalization capabilities to be applied to a variety of design flows and to permit tool interoperability. Furthermore, the approach requires a strong level of expertise as all elements of the DPR design flow need to be modelled. Despite its complexity, in [128]the authors demonstrate how their methodology can be exploited to move from MARTE models to automatic code generation (i.e.VHDL) and in other works, such as in [133], the authors propose future avenues of research in applying UML MARTE in the modeling of DPR systems. However, effective IP reuse or IP-XACT support is still not considered.

### 3.3.3/ THE MADES APPROACH

Another interesting approach, MADES (Model-based methods and tools for Avionics and surveillance embeddeD systEmS) [134] aims at developing the elements of a fully model-driven approach for the design, validation, simulation, and code generation of complex embedded systems to improve the current practice in the field. The project does not target dynamically reconfigurable systems, but it is closer to our approach in the sense that one of the axes of their methodology targets Xilinx Platform Studio as back-end. MADES differentiates itself from similar projects in the way it covers all the phases of the development process: from system specification and design down to code generation, validation and deployment. Design activities exploit a dedicated language developed on top of the OMG standard MARTE, and foster the reuse of components by annotating them with properties and constraints to aid selection and enforce overall consistency.

Code generation addresses both conventional programming languages (e.g., C) and hardware description languages (e.g., VHDL), and uses the novel technique of Compile-Time Virtualisation to smooth the impact of the diverse elements of modern hardware architectures and cope with their increasing complexity. A conceptual model of the different inter-relationships between the various artifacts of the approach is illustrated in Figure 3.6

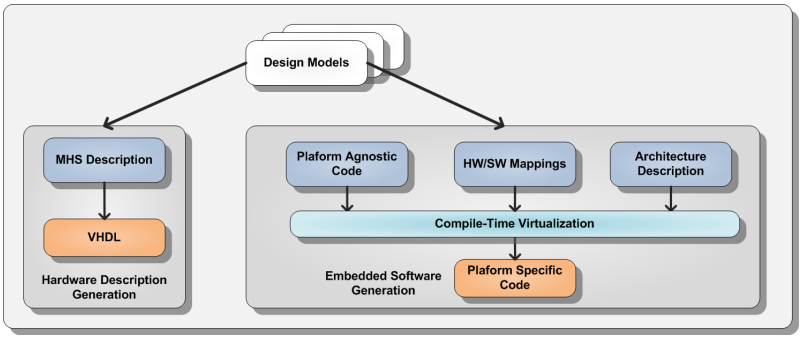


Figure 3.6: Global MADES Methodology and supported artifacts

One of the main characteristics of the MADES approach to software development for em-

bedded systems is that it is model-driven and transformation-based. From the previous figure, it can be seen that the MADES model-driven approach focuses on the generation of platform specific embedded software from architecturally-neutral software specifications, generation of hardware descriptions of the modelled target architecture, and verification. The platform models are described using a custom meta-model, despite being very recent (they seem not to be aware of IP-XACT), which leads to the aforementioned issues. As with the previous methodologies, the authors do not present how IP reuse is actually achieved; furthermore, the authors claim that other tool chains can be targeted by modifying the annotations in the models, which clearly shows that their IP representations lack generalization capabilities for targeting multiple design flows.

Regarding the supported back-ends, Xilinx MHS descriptions are generated by the MADES tools from an allocated model using a model-to-text transformation. The Xilinx tool Platgen is then used as an underlying HDL generator, as it can take an MHS description and generate a set of synthesizable VHDL files for implementation on an FPGA [135]. The use of MHS files and Xilinx FPGAs is not required by the MADES tool chain, but according to the authors, it is a useful language to work with as it already has robust industrial-quality tool support available.

## 3.4/ Hardware Modeling using UML and IP-XACT

### 3.4.1/ The SPRINT Methodology

In [136], the authors have investigated the application of the UML for modelling IP-XACT compatible component and system descriptions by mapping several IP-XACT concepts to corresponding UML concepts. They make use of a full fledged UML profile for IP-XACT that should cover the same amount of information as the corresponding IP-XACT description. This capability should, according to the authors, enable the automated generating of IP-XACT components and design descriptions from respective UML models. Moreover, such an approach would permit the automatic integration (IP reuse) of existing IP within UML. Their approach targets SystemC generation from UML, using IP-XACT as an intermediate representation, as depicted in Figure 3.7 a). In their work, they present a series of UML meta-models for representing different concepts of the IP-XACT standard, such as the IP-XACT component description depicted in Figure 3.7 b). They present an application targeting a Core Connect based system, but it is oriented to generation of SystemC Transaction-Level Model (TLM) without reporting the integration of RTL components [137].

The SPRINT methodology was one of the initial approaches aiming to merge UML and IP-XACT. Although interesting, we believe that this effort is ill-suited for SoC design using UML, since it tries to make available to the modeller the totality of the meta-data contained

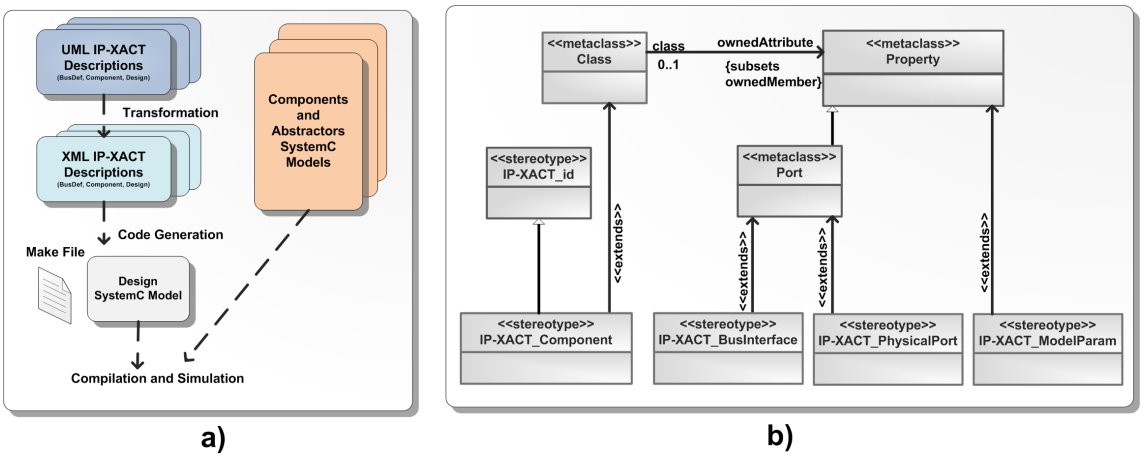


Figure 3.7: a) SystemC flow based on IP-XACT b) Component diagram

in the IP-XACT descriptions such bus interfaces, components and designs, among other objects defined in the standard. An issue with this approach is that it can make the UML diagrams illegible due to information clutter, and secondly, that does not abstract information that is very low-level. This is especially true with IP-XACT representations, which have been conceived to store as much information as possible, and to be processed by machines.

These handicaps has been recognized by others ongoing efforts, in which only a sub-set of the IP-XACT components description is made available; this approach at integrating IP-XACT in UML makes more sense, since the modeller at high-levels of abstraction does not require to know the full details of the component description. A simple meta-model at the deployment phase should enable to capture the most important information of the IPs (namely, parameters and bus interfaces) to permit the UML designer to build a platform containing enough information for the models transformations phase, in which introspection is used for adding more details for creating an executable model.

### 3.4.2/ THE HELP APPROACH

A second approach at integrating IP-XACT in UML methodologies for SoC has been carried out by the AOSTE Team at INRIA, in the context of the HELP project [138]. The aims of the project are closely related to those of the SPRINT initiative, namely, providing an UML profile for IP-XACT for integrating IP blocks into high-level models. The main difference between the two approaches lies in the selection of modeling language: the former makes use of pure UML, whilst the latter opts for the UML MARTE profile. Similarly, both approaches tackle the generation of SystemC code for rapid system design exploration; HELP makes use of a tool, SCIPX, for extracting information of SystemC modules to create their component library though reflection [139].

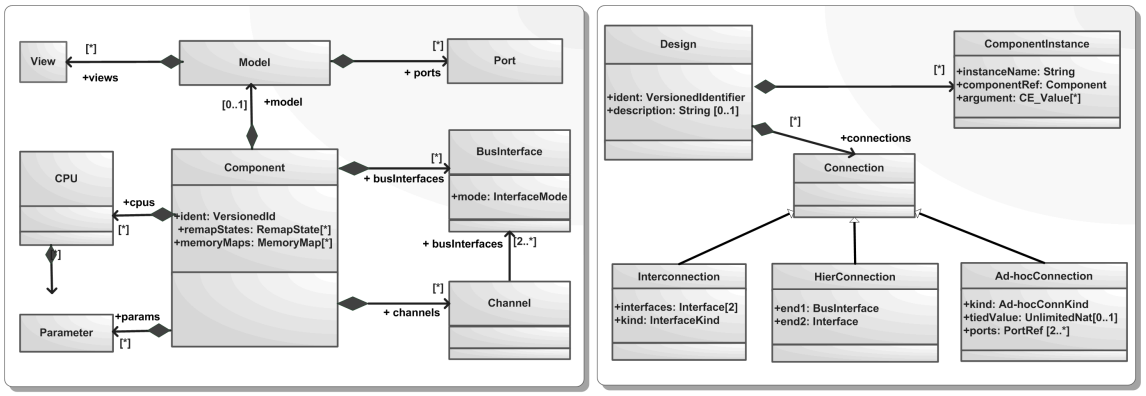


Figure 3.8: Two IP-XACT meta-models in the HELP approach: a) Component meta-model b) Design meta-model

As with the SPRINT methodology, the HELP project makes use of the extension mechanisms provided by UML to create a set of IP-XACT meta-models for the different objects described in the standard, which are presented in [140]. Examples of these meta-models are shown in Figures 3.8 a) and b), which depict the component and design meta-models, respectively. However, contrary to the former approach, the authors present the basic transformation rules necessary to move from a UML MARTE model using these extensions to an IP-XACT design description. These rules have been widely used in ongoing efforts; however, it must be noted that, as with the SPRINT methodology, most of these efforts do not make use of the complete IP-XACT meta-model in UML, but a sub-set of concepts that are sufficient to generate an IP-XACT platform description. This is the approach that we have decided to follow, as will be described in the next chapter.

### 3.4.3/ The TUT Profile and Methodology

In [141] an extension of UML is done defining a new profile, the TUT profile, for the modeling of embedded systems designs. In [142] an UML design flow is defined. The authors define a set of stereotypes to model application tasks and platform. Figure 3.9 a) shows the elements of the TUT profile. An application is composed of application components, which are instantiated as application processes. Next, application processes are grouped into a process group. Correspondingly, a platform is composed of platform components, which are instantiated as platform component instances. Finally, process groups are mapped to platform component instances using platform mapping. Figure 3.9 b) shows the design flow. It is composed of five main phases: Specification and requirements, UML design, UML interface, Architecture Exploration and Physical implementation. Each phase refines the previous one. The main goal is to design an embedded system from an IP library rapidly, allowing design space exploration and software code generation. The platform uses a library to allow performance analysis. The design flow allows software

code generation in C. It has an architecture space exploration tool that back annotates the UML model. This approach supported IP reuse, though the original proposal did not support IP-XACT as the chose meta-model intermediate representation.

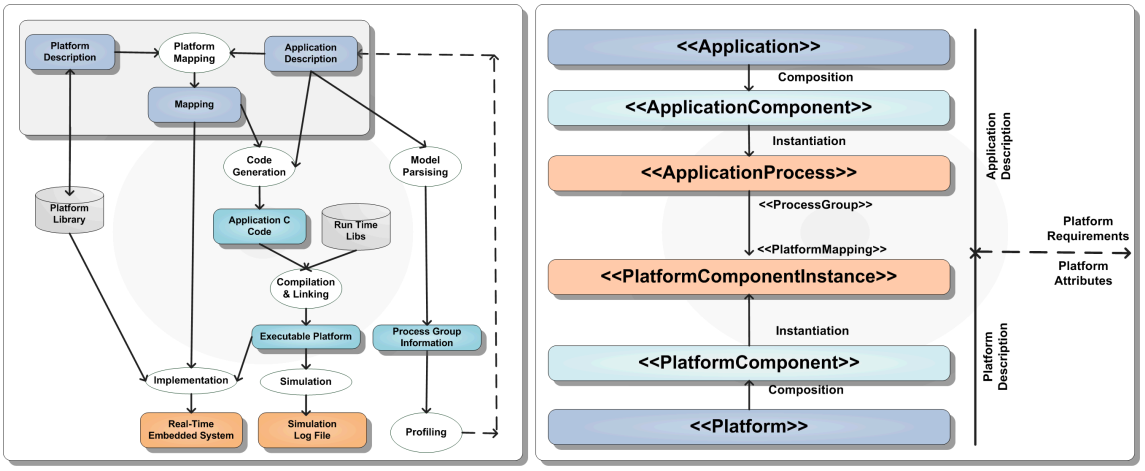


Figure 3.9: a) TUT-Profile design and profiling flow b) TUT-Profile hierarchy

The original proposal for the TUT profile has been extended in [143] which maps the TUT UML profile for embedded systems design to IP-XACT meta-data. The hardware components and the associated IP-XACT descriptions are contained in an IP component library, effectively promoting IP Reuse; they are abstracted by corresponding UML2 models that are contained in an UML model library. Subsequently, the top-level IP-XACT design description is represented by the UML HW platform model. This framework enables the designer to carry out the structural design of a HW platform using UML notations, and automatically generating the IP-XACT design descriptions by using a set of transformation rules similar to those presented in the SPRINT and HELP projects. The resulting IP-XACT design flow using UML2 is also presented which allows automatic RTL component integration based on the proposed transformation rules. Subsequently, the authors further demonstrated their approach in [144], adding modeling concepts, to implement a complex MP-SoC. The main phases of the TUT approach are depicted on Figure 3.10; the structure of the platform is modelled in UML2 using the rules and stereotypes defined by the TUT-profile. An UML parsing tool transforms the TUT system models into a custom XML file, called XML System model (XSM). This file contains information shared by different design tools in the TUT framework (i.e. for DSE). This file is further transformed into the top-level IP-XACT design XML file which is fed to a Hardware Platform Configuration Tool (HPCT). This tool integrates the IP components based on IP-XACT meta-data descriptions of the IP blocks. Furthermore, it governs the synthesis tools for their chosen back-end, targeting Altera FPGAs.

This work is one of the first successful attempts at integrating IP-XACT into UML methodologies, but the alleged IP reuse capabilities and automatic IP integration aspects are not

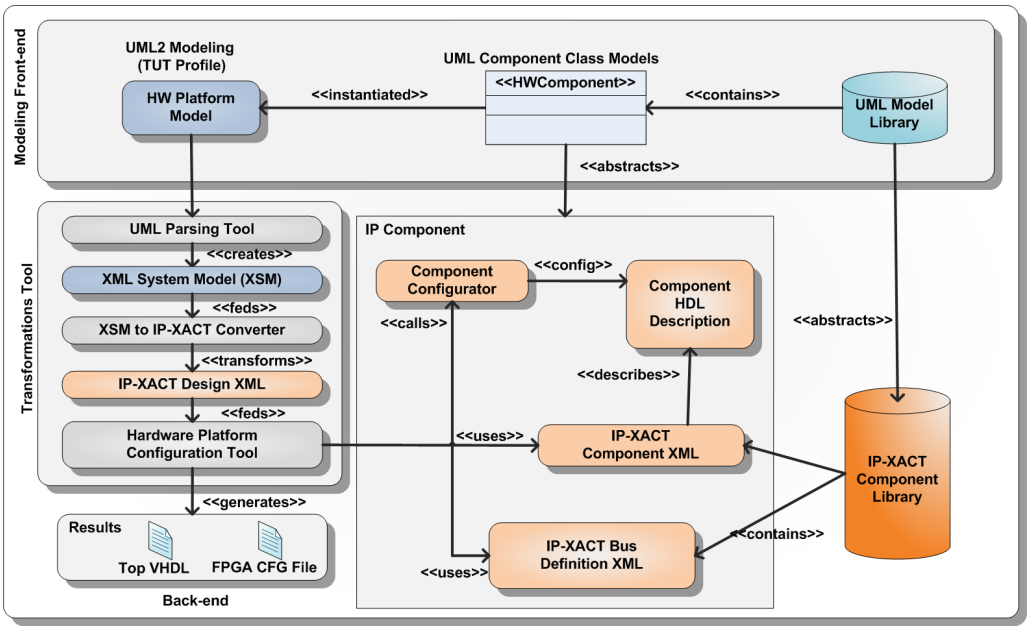


Figure 3.10: MDA-based hardware platform configuration framework

explicitly defined. Furthermore, it targets Altera FPGAs, which do not support full fledged dynamic partial reconfiguration as in Xilinx devices. However, our methodology for IP integration takes some of the ideas presented in this work for creating the IP libraries, which are similar to the schema presented in Figure 3.10. The main difference of our approach is that we make use of the MARTE profile, which is gaining acceptance for modeling SoCs; we have defined extensions that enable us to integrate IP reuse capabilities and IP integration in a similar manner to the TUT methodology, but exploiting IPs that are to be integrated into a DPR design flow, which require specific capabilities.

### 3.4.4/ THE COMPLEX APPROACH

An interesting approach merging the UML MARTE profile with the IP-XACT standard is the European project COMPLEX [145], whose aims are mainly directed towards the automatic generation of fast performance executable models. The COMPLEX project is based on an UML MARTE framework supported by a tooling fully integrated in Eclipse, which enables the automatic generation of different intermediate representations. As shown in Figure 3.11, the performance model is generated by means of a set of code generators, relying on intermediate standards formats and languages such as IP-XACT, XML, SystemC and C/C++. In the discussion that follows, we concentrate in the platform generation branch of the approach, since it is this component of this project which is closer to our methodology.

In COMPLEX, an UML/MARTE-based specification methodology supports the capture of the hardware and software architectures, and of the main parameters of the system.

Moreover, the user can specify a design space (roughly, the set of tuneable parameters) where a design exploration tool will look for the optimal implementation solutions. In COMPLEX, the exploration tool relies on SystemC executable platform dependent models (PDMs) at different abstraction levels, which enables DSE loops where performance estimation is done with different accuracy vs simulation speed trade-offs.

The IP-XACT standard is used in this framework as a means to leverage the targeting of the UML MARTE platform dependent model to different simulation infrastructures. A specific front-end plug-in for IP-XACT has been developed to enable another tool, the SCoPE simulation infrastructure, to be able to read the IP-XACT platform description for building a fast executable model for functional validation and performance estimation. The generation of the IP-XACT platform description and its use in the COMPLEX approach is presented in [145], while the transformation rules implemented to obtain the IP-XACT description from the UML MARTE models are presented in [146], where the MARTE to IP-XACT (MARTIX) generator tool used in the COMPLEX framework is introduced.

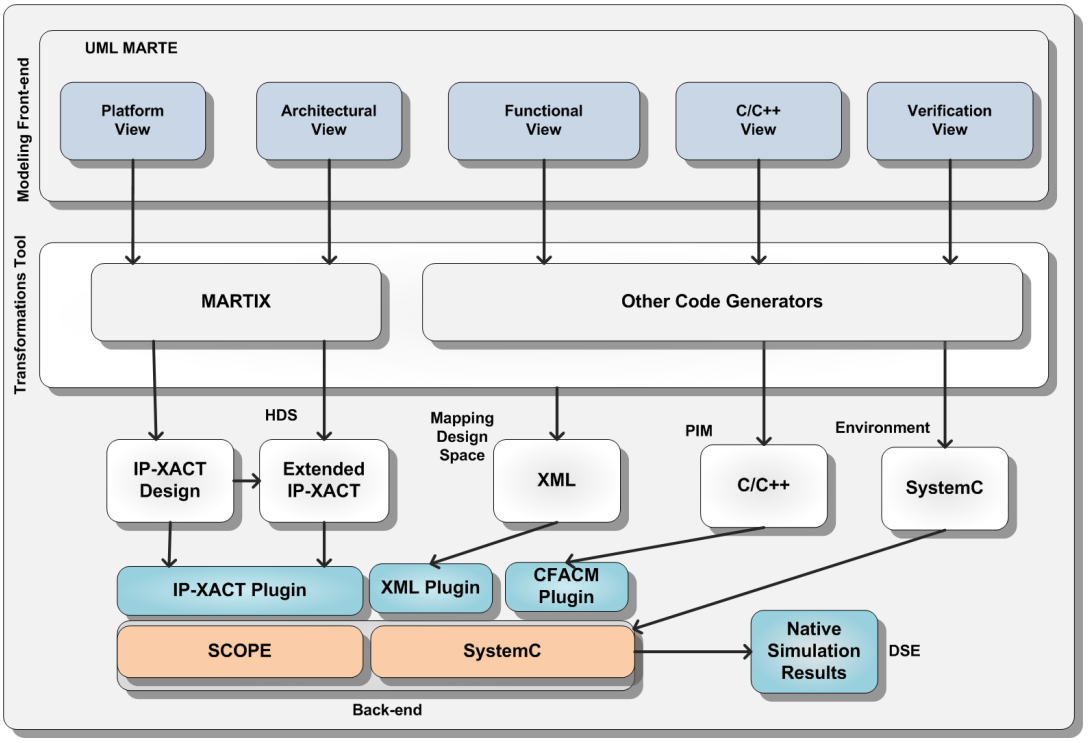


Figure 3.11: IP-XACT flow within the performance model generation flow of COMPLEX

This tool produces an IP-XACT description of the HW platform architecture from the COMPLEX UML MARTE models. More specifically, MARTIX reads the architectural and platform views of their framework and produces a top-level IP-XACT design file that contains the components instantiations, and their parameterization and interconnection information. This approach implements and automates the transformation rules already proposed in the HELP project, and shows that is indeed possible to obtain a usable system description in IP-XACT. However, they target the generation of SystemC code for design

space exploration (DSE) and moreover, IP reuse capabilities are not supported, since the models are manually annotated for linking the component instances in the UML models to the underlying IP-XACT component descriptions.

However, in the FAMOUS methodology, we make use of some of the ideas presented in [146] to perform the transformations from UML MARTE to IP-XACT. This phase of our methodology differs from the MARTIX tool in the sense that the input UML models make use of a deployment extension in MARTE which facilitates IP reuse, a feature that will be introduced in the next chapter. Secondly, we make use of the IP-XACT design description not for generating SystemC models, but for obtaining the models used by Xilinx Platform Studio to build the SoC platform.

## 3.5/ Discussion and Conclusions

Despite the relatively large number of proposals in the modeling of SoCs using UML MARTE on the one hand, and a combination of UML and IP-XACT on the other, so far there are not methodologies that use a Meta model-based approach for DPR systems, as depicted on Figure 3.12. Moreover, just about half of the approaches described in this chapter target FPGA implementations, and furthermore, they do not provide clear or effective mechanisms for IP reuse, due in large part to the intermediate XML representations for the deployment phase.

| | MDE-based Co-Design Approaches | | | | | | |
|---|---|---|---|---|---|---|---|
| **Comparison Metrics** | **MoPCoM** | **GASPARD** | **MADES** | **SPRINT** | **HELP** | **TUT** | **COMPLEX** |
| Modeling Front-end | UML2 | MARTE | MARTE | UML2 | MARTE | UML2 | MARTE |
| Intermediate Representation | Custom | Custom | Custom | IP-XACT | IP-XACT | IP-XACT | IP-XACT |
| IP Reuse Support | No | No | No | Yes | Yes | Yes | No |
| Target Back-end | VHDL | TLM/VHDL | TLM/VHDL | TLM | TLM | VHDL | TLM |
| FPGA Implementation | Yes | Yes | Yes | No | No | Yes | No |
| DPR Support | Yes | Yes | No | No | No | No | No |

Figure 3.12: Different approaches making use of UML for Co-design

These XML IRs usually are not oriented for the description of SoC platforms, and thus have failed to attract the interest of the industry. Other approaches target the generation of SystemC code, but are more aligned to the aims of this thesis and we have gathered some inspiration from them, in particular from the Complex project. We believe that the combination of UML MARTE and IP-XACT can improve the applicability of the model-driven approaches to the development of FPGA-based SoC platforms, and in particular, facilitate the conception of DPR systems. This can be achieved by combining a component-based

approach (for which IP-XACT was conceived) and a well fixed IP taxonomy for easily associating different blocks in the UML MARTE models to their IP-XACT (and PRM, partitions and DPR wrappers counterparts). However, targeting pure VHDL generation might be counter-intuitive, since it does not lend itself to further IP reuse; it is thus preferable to exploit the capabilities of IP-XACT as a standard intermediate representation (for both IP blocks and the platforms they compose) for generating the desired back-end representations. We have proceeded in this manner: we use IP-XACT for promoting IP reuse (IP reflection from the Xilinx XPS library) and design by reuse for generating the XPS platform representation from IP-XACT. In both cases, the model transformations enable this passage seamlessly, without compromising the necessary abstraction in UML MARTE and the flow agnostic philosophy of IP-XACT regarding the target back-end.

As discussed at the end of Chapter 2, an MCF methodology requires the use of a Modeling Front-end and of a platform generation shell. In the case of IP-XACT, both functions are carried out by the so-called Design Environment (through the Design Capture and Design build phases), while the MDE methodologies discussed in the previous sections use UML MARTE and the associated tools as the modeling front-end. On the other hand, the generation phase, through model transformations, takes place in other tools, usually based on the Eclipse Framework. In the FAMOUS framework, we make use of Sodius MDWorkbench, an industrial tool developed by one of the partners of the FAMOUS consortium, which enables the development of metamodels, the import of models conforming to these metamodels, and the definition and application of transformation rules for performing transformations between these models. MDWorkbench is at the heart of the FAMOUS framework, and federates the FAMOUS compilation chain; thus, the diagram depicted on Figure 3.13 shows only the hardware components of the complete methodology.

As depicted on the diagram, the use of MDWorkbench (2) represents an intermediate step between the high-level models in UML MARTE (1) and the low-level Xilinx Platform Studio (3) representations, and its eventual use in Xilinx PlanAhead (4) in the form of synthesized netlists. The tool has been used to create a set of metamodels, first for all the objects defined in the IP-XACT standard (e.g. bus and abstract definitions, components and designs, amongst others), but also for the models used by Xilinx Platform Studio for describing the IP and the platform. A set of transformation rules have been proposed to move from this Xilinx XPSF models to IP-XACT components (for promoting IP reuse), which can be imported into modeling tools such as Papyrus in the form of XML templates for composing the SoC platform. The UML platform description obtained in this manner is then imported into MDWorkbench in the form of an XMI file and transformed into an IP-XACT design (and analogous of the design capture capabilities of the design environment). Then, model transformations from IP-XACT to XPS are used for obtaining a description used by Xilinx tools to obtain the complete VHDL platform description automatically. As discussed before, proceeding in this manner effectively promotes an

MDE philosophy in which a high-level user only deals with abstract concepts, while the metamodeler defines the transformations rules required for moving to different back-ends.

The reminder of this thesis manuscript deals with the contributions of this thesis for implementing the compilation chain shown on the figure. First, in Chapter 4 we describe in detail how Xilinx Platform Studio is embedded in the creation of SoC and DPR systems, the modeling of the different Xilinx XPS IPs and their back-end representation. Then, in Chapter 5, the metamodels and transformation rules used to move from these IP models to IP-XACT components and subsequently to UML templates, for creating the reusable IP library, are thoroughly described. Finally, the composition of DPR platforms from this library and the required deployment metamodels are introduced in Chapter 6, as well as the used transformation rules.

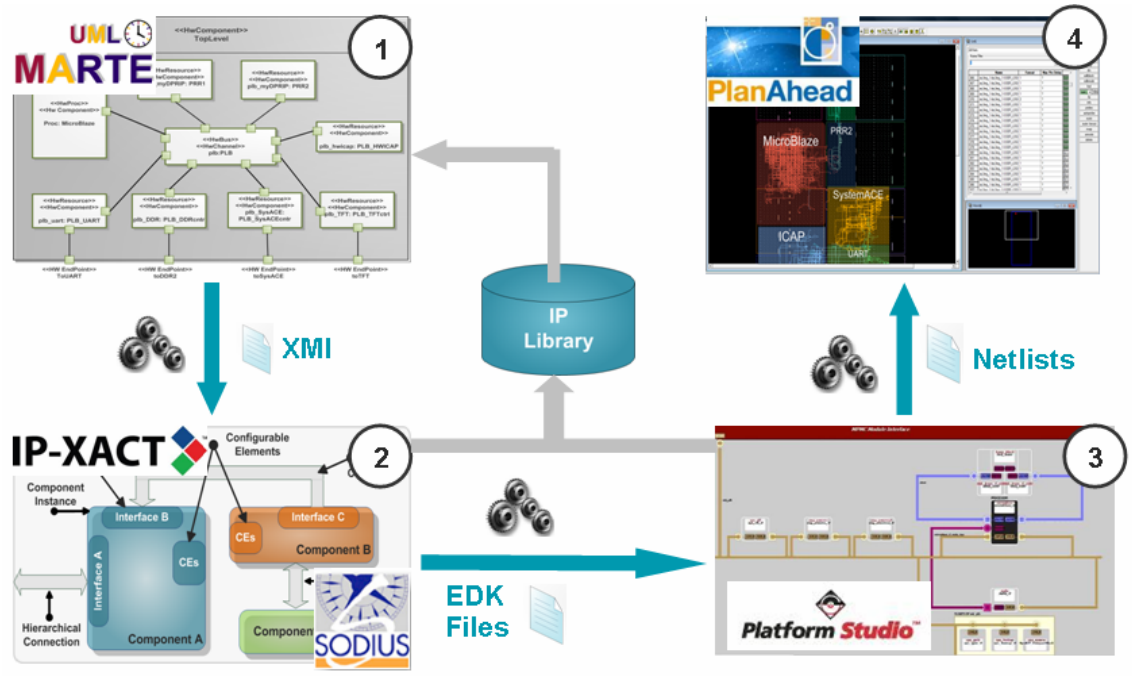


Figure 3.13: Tool support for the proposed DPR system flow

# II

# PROPOSED METHODOLOGY FOR DPR IP REUSE AND PLATFORM GENERATION

# 4

# THE ROLE OF THE PROPOSED APPROACH IN THE DPR DESIGN FLOW

## Contents

## 4.1/ INTRODUCTION

Metadata-driven composition frameworks have been introduced in order to cope with the complexity of the different aspects involved in the creation of complex systems. Applications in many areas have been demonstrated, such as in CAD tools for mechanical design, avionics, and in the conception of electronic systems. In the context of System-on-Chip, the use of MDE has raised increased interest, but only recently this paradigm has coalesced with efforts such as IP-XACT. As described before in Chapter 2, IP reuse and design by reuse are two of the problems that can be efficiently solved by combining the MDE and MCF paradigms. In particular, from the previous part of this manuscript, it must be clear by now that the composition of the DPR systems, and the configuration of the different reconfigurable modules (the design entry phase), is essentially a design by reuse problem. Design by reuse methodologies advocate for the use of well-defined interfaces, generic coding standards leading to well-documented IPs (information about parameters, and design flow specific information) that can help to facilitate the creation of complex systems. This is especially true in the context of Systems-on-Chip (SoC) and Networks-on-Chip (NoC), where many efforts at defining standard bus interfaces have been carried out during several decades. This is due to the fact that the IP ecosystem requires of common interfaces and programming models to boost IP reuse by companies developing new products.

Therefore, IP providers and EDA tool vendors have proposed several bus topologies over the years, such as AMBA [82], and CoreConnect [81]; competition among these players in the SoC industry has lead to an ecosystem in which IP integrators face the issue of having to stick to a given design flow in order to find support for the chosen bus topologies. The same applies to FPGA vendors aiming at providing tools for facilitating the creation of FPGA-based SoC systems, such as Xilinx Platform Studio and Altera SoC Builder; for instance, Xilinx have made agreements with IBM for being able to use CoreConnect and AXI interfaces within their tools, which offers the final user a larger range of IPs to choose from, but which also limits the applicability of the so obtained (or created) IP blocks to design flows that support these standards. Academic endeavours have aimed at defining bus standards that facilitate the exchange of IP blocks without the burden of using expensive tools, but many of these efforts, such as OCP-IP [85] and Wishbone [147] haven't gained enough traction.

Another advantage of using standard interfaces is that the IP integrator deals with a reduced set of interconnection information: instead of dealing with virtually hundreds of different interfaces, IP blocks (e.g. hardware accelerators, communication IPs, video IPs) are wrapped with a logic interface that copes with the bus transactions, exposing the user with a restrained set of interfaces. Moreover, communication with the IP blocks is simplified, since they become task that can be accessed by the processor through an API. Furthermore, by bundling all the master and slave signals together, the interconnection

between IPs becomes simpler and more effective, reducing the burden of interconnecting hundreds of different signals. The parameters of the IPs attached to the logic interface can then be associated with the IP wrapper, facilitating the overall IP reuse cycle.

However, standard bus interfaces alone do not completely solve the IP reuse problem. The ports comprising the IP wrapper interfaces still require a great deal of effort to interconnect if the work has to be performed manually. Moreover, hardware IPs are often complex modules which are highly parametrisable and in some instances, support functionality customization; these parameters can affect the behaviour of the interfaces (e.g. data and address widths), internal functions or the inclusion of functionalities into the IP block depending on other parameters (e.g. flow or technology dependent). This information, as mentioned previously, is considered meta-data about the IP, and current practices by many IP integrators consist in analyzing the IP blocks behaviour via complicated datasheets and then parameterizing the blocks through generics in the hardware instances of a top-level description.

In order to cope with this problem, EDA tool and FPGA vendors alike have developed proprietary meta-models that aim at abstracting the IP low-level implementations. These meta-models have lead to the creation of Metamodeling-driven Component Composition Frameworks for facilitating the interconnection and parameterization of IPs deploying sophisticated EDA techniques. For instance, Xilinx Platform Studio makes use of a set of proprietary meta-models for describing the IP blocks (the Microprocessor Peripheral Definition, MPD) and the top-level platform (the so called Microprocessor Hardware Specification, MHS); these two descriptions, whose role will be explained in more detail in the next section, represent an improvement over a purely VHDL descriptions. This is due to the use of a formal semantic, which enables their use as machine readable formats for design automation purposes. These two meta-models greatly facilitate the reuse of Xilinx IP blocks and their interconnections, moreover, through the use of Xilinx tools, a top-level HDL description can be obtained automatically.

Furthermore, the FAMOUS methodology aims at the creation of DPR SoC systems. Currently, only Xilinx supports full Dynamic Partial Reconfiguration capabilities, and therefore, we concentrate our efforts in the design for reuse of Xilinx IP cores supporting CoreConnect bus interfaces. This is due to the fact that we aim at interfacing our methodology to the creation of Xilinx Platform Studio SoC designs; as it will be explained in the next section, by using IPs with a fixed interface and a well defined programming model, we can constrain the problem of having a plethora of heterogeneous IPs with a set of different interfaces. Moreover, Xilinx provides many EDA mechanisms to abstract the low-level implementation details, making it easier to couple with a high-level methodology for the creation of a Metamodeling-driven Component Composition Framework. Nevertheless, it must be noted at this point that we decouple the FAMOUS front-end from the Xilinx Platform Studio back-end through the use of the IP-XACT standard for describing the components and their interfaces, along the obtained by composition system description.

## 4.2/ COMPONENT-BASED APPROACHES FOR SOC DESIGN: A HARDWARE PERSPECTIVE

The interface of the reconfigurable socket is a critical system-level design decision. Since this interface is fixed with the design of the static system, it must be flexible enough to allow any anticipated applications to be implemented inside the reconfigurable region. For systems where the static design must support a set of reconfigurable modules designed for a particular application, this can be done relatively easily. However, in order to implement an application-independent static design and to enable reuse of the socket IP core, a generic interface must be chosen.

The simplest approach (beyond a strict point-to-point interface) is to provide a bus interface. Architecting this interface around a standard bus protocol, such as the IBM CoreConnect processor local bus (PLB), provides this flexibility and promotes IP reuse via a component based approach. Most currently available Xilinx IP are based on an FPGA-optimized variant of version 4.6 of this specification, as depicted on Figure Figure 4.1 a). The processor(s) and memory controller(s) connect to the bus via a standard bus interface. The bus interface is specific to the particular bus, but at the simplest level consists of address, data, read/write requests, and acknowledgment signals. The bus also includes a bus arbiter, which controls access to the bus. When a core needs to communicate with another core on the bus, it issues its request for access to the bus. Based on the current state of the bus, the arbiter will either grant access or deny access, causing the core to reissue its request. A core that can request access to the bus is considered a bus master. Not all cores need to be bus masters; in fact, many custom cores are created as bus slaves, which only respond to bus transactions. A bus on an FPGA is implemented within the configurable logic, making it a soft core. For example, Xilinx uses IBM CoreConnect library of buses, arbiters, and bridges, and a large set of IP components developed around the bus standard interface. By using a well defined bus interface, the designer of an IP needs only to interface the internal static interface to the reconfigurable partition (implemented using partition pins), effectively componentizing the functionalities to be mapped to the so-called IP wrapper (or static placeholders). This approach also fosters a component based approach in which the IP block obtained in this manner becomes a mere black-box to be instantiated into a DPR SoC platform along other PLB-wrapped static IP blocks.

However, using this standard protocol directly is very complex. One difficulty is the large number of signals that are required to implement an arbitrary PLB slave (if the entire socket was to be used as a Partially Reconfigurable Module), including up to 128 data signals each way, 64 address signals, plus a large number of additional control signals. In total, this requires over 300 signals to be passed across partition pins, even though most systems are unlikely to implement 128-bit wide slaves. A second difficulty is that some of

the widths of the bus control signals depend on the number of masters and slaves. This makes modifying the system difficult, since each signal must be given explicit placement constraints.

In order to simplify the process of attaching a user core to a CoreConnect bus, the user can make use of a portable, pre-designed bus interface (called the IP Interface, IPIF) that takes care of the bus interface signals, bus protocol, and other interface issues. The IPIF presents an interface to the user logic called the IP InterConnect (IPIC), as depicted on Figure 4.1 b). The user component part of the IP is logic that has been designed with an IPIC interface to make use of the IPIF bus attachment and other services. The user logic that is designed with an IPIC has the advantage that it is portable and can be easily reused on different processor buses by changing the IPIF to which it is to be attached.

### 4.2.1/ The Intellectual Property Interface Wrapper

The PLB IPIF module, depicted on Figure 4.1, simplifies the number of signals and the complexity of the handshaking required to connect a custom core to the PLB bus. The PLB IPIF is designed to provide an IP developer with a quick method to implement a highly adaptable interface between the CoreConnect PLB bus logic and the User IP core. Through the use of VHDL generics, the design provides various services and features that can be optioned in or out, depending on the developer requirements. The back end-interface standard, the Xilinx IPIC (IP interconnect), exposes the interface between the IPIF logic and the user created logic. The most basic user logic contains a set of software registers that can be accessed by the processor and that are mapped to the system memory. The registers provide a simple interface to exchange data and control signals

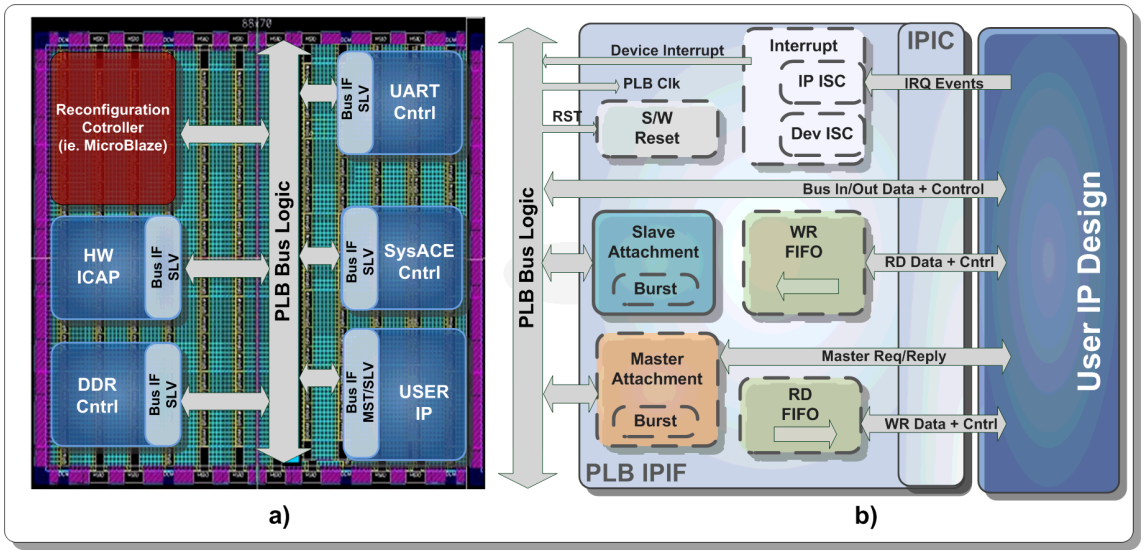


Figure 4.1: a) A typical DPR architecture based on the PLB bus b) Xilinx PLB Intellectual Property Interface(IPIF)

between the processor and the attached IP block.

In its current version, the PLB IPIF provides nine services for the IP developer of which, seven can be optioned in or out (shown in dashed squares on the figure). The base element of the design is the Slave Attachment. This block provides the basic functionality for PLB Slave operation. It implements the protocol and timing translation between the PLB Bus and the IPIC. Optionally, the Slave Attachment can be enhanced with burst transfer support. This feature provides higher data transfer rates for PLB Cache line accesses and enables the transfer protocol for PLB fixed sized burst operations.

In addition, the IP developer may option in a software Reset service. This service allows the system microprocessor to perform a local reset of the IP by writing a data key value to the User assigned address space for the Reset Service. This Reset Service is in addition to the hardware reset provided by the PLB Reset input. When activated by the write, a reset pulse is generated that is sent to the User IP and the various internal IPIF elements (excluding the Slave Attachment which has to complete the write timing). Moreover, an Interrupt Service is provided in the PLB IPIF; this module collects interrupts from the IP and internal IPIF interrupt sources. It then coalesces them into a single interrupt output signal that is available for connection to a system interrupt controller or the microprocessor. The block contains user accessible capture registers and enable/disable registers.

Furthermore, the IPIF module provides read and write FIFO Services. They may be independently optioned in or out of the IPIF implementation. These FIFOs are synchronous to the PLB clock and have user parameterized depth, width, and packet support features. Moreover, the IPIF provides an optional Direct Memory Access (DMA) Service. This service automates the movement of large amounts of data between the IP or IPIF FIFOs and other PLB peripherals (such as System Memory or a Bridge device). The inclusion of the DMA Service automatically includes the Master Attachment Service. The DMA Service requires access to the PLB as a Master device. The DMA Service may be optionally enhanced with Scatter/Gather mechanization that allows the User to automate DMA operations via Buffer Descriptors stored in System Memory.

The IPIF module is responsible for interfacing with the PLB and generating the appropriate IPIC signals. The IPIC signals are a much simpler set of bus interface signals, abstracting the complex bus signals away from the designer.

### 4.2.2/ ADVANTAGES OF COMPONENT-BASED DESIGN

The Xilinx EDK includes a large repository of prebuilt, customizable soft cores and wrappers that provide an interface for instantiating hard cores. These cores range from UARTs to buses to memory controllers and even soft processor cores. IBM CoreConnect adds to this repository by providing soft core versions of the already widely used buses, bridges,

and arbiters that have been configured for use with the Xilinx FPGA devices. By using these cores it is possible to build a processor-memory system and then add custom cores to implement specific, user defined, functionalites (such as Hardware Accelerators).

Most providers of the IP blocks pre-verify their designs, thus, helping the downstream system designers save additional verification effort. For IP blocks with a large number of underlying logic design and interfaces, this design time savings can be extremely valuable. Sharing a common IP catalog and a standard interface ensures that the software and hardware design teams can build sub-systems and integrate them into larger systems. It also means that updates to any of the IP cores or the sub-systems can be readily assimilated into existing as well as new system designs. Xilinx design tools such as Xilinx Platform Studio (XPS) are designed to automate the assembly of IP blocks and interconnect with the PLB interface into a system. XPS provides a graphical system assembly tool for connecting blocks of IPs together using interface level connections to implement systems, with or without processors. XPS provides native support for PLB Interconnect IP, allowing designers to explore and tailor system topologies to their application requirements.

## 4.3/ SIMPLIFYING DPR IPS MANAGEMENT THROUGH COMPONENTIZING

In this section, we describe how the wrapping of DPR components can facilitate the mapping of heterogeneous Partially Reconfigurable Modules (PRMs) into FPGA-based SoC platforms. We make use of the Intellectual Property Interface (IPIF) module introduced in the previous section, but this approach can be deployed to any other wrapper architecture, such as AXI, also used by recent Xilinx families, or even the Open Core Protocol (OCP-IP), which has been gaining traction in recent years to improve reusability of hardware components among IP providers and integrators.

### 4.3.1/ WRAPPER-BASED DESIGN FOR DPR IPS

As with any piece of IP, the DPR design flow can be benefited by incorporating reconfigurable functionalities into DPR placeholders which are nothing more the mere prebuilt black-boxes which can be easily instantiated into a top-level description, through the use of standard external (top-level) bus interfaces. The basic idea to map any hardware accelerator in these placeholders (wrappers or containers), not in the design entry phase, but at run time after the system has been implemented. In order to do so, the placeholders need to be architected in such a way that they internally contain a reference to a customizable black-box definition, which in fact represents a Reconfigurable Partition (RP), and to which any number of Partially Reconfigurable Modules can be linked at the floor-

planning phase of the DPR flow. As discussed in Chapter 1, one of the main constraints of the current design flow, and which is unlikely to change in the future, is the necessity of a static interface between the Reconfigurable Partition instance and the static logic (represented on the figure by the IPIF services and the glue logic required to attach it to the eventual PRMs to be mapped onto the RP). This interface is labelled DPR Interface in the figure, and requires undergoing a standardization process so that different hardware accelerators (HWAcc, or other user defined functionalities) can be mapped to it. This means that all HWAcc to be used in a given DPR application (e.g. such as DCT or FFT blocks) need to be created in such a way that their functionalities (communication and control protocols) conform to this predefined interface.

This approach can seem as a handicap at a first glance, but as with many other methodologies, it can foster IP reuse and an eventual agreement in common interfaces for DPR hardware IPs. In the context of the FAMOUS framework, it further promotes a second degree of component-based design, in the sense that not only the static and DPR IP placeholders can be easily attached to the top-level SoC platform, but also DPR tasks (PRM modules) can be easily associated to these containers, through a plug-and-play approach. As components can be seen merely as black-boxes with a set of interfaces to be connected, and parameters to be set by the designer (in the form of metadata), the implementation details are hidden by the use of a top-level description (the IPIF wrapper). Hence, they can easily be abstracted by metamodels (i.e., the MPD file, which is created automatically by the XPS tools when defining the functions of the wrapper, through a set of GUIs) and used at high-levels of abstraction using the IP metadata reflection mechanisms introduced in Chapter 2, and which will be discussed in-depth in the next chapter.

In recent years, there has been plenty of research in the area of interface-based design

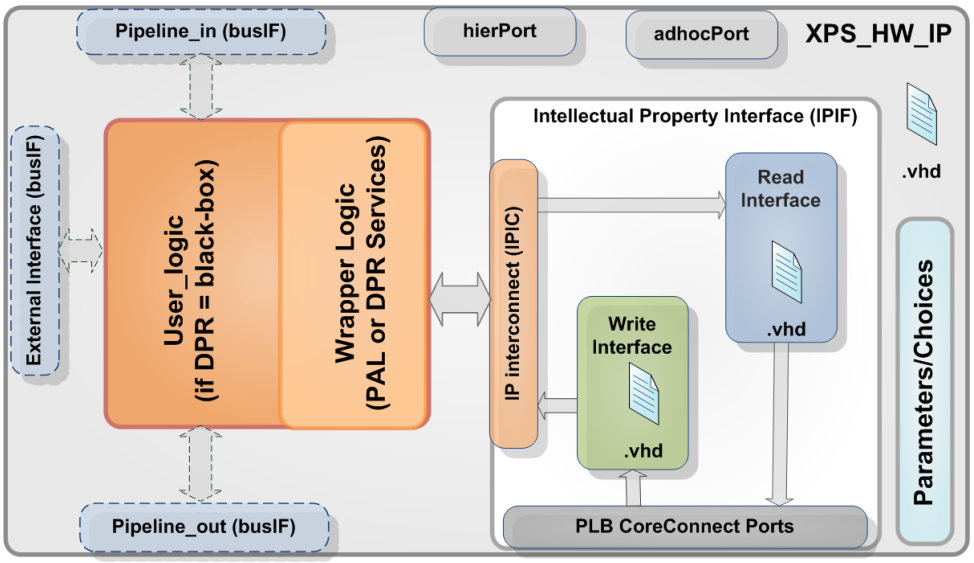


Figure 4.2: A generic model for DPR Wrapper Design

for DPR IPs [148,149]. It is not the goal of this thesis manuscript to engage in a thorough description of such approaches, but to use what has been learned from them in order to incorporate such advances into a MDE-based methodology. As discussed in the introduction, other works in the FAMOUS project deal with the architecture of such DPR IPs, and in the sub-sections that follow, we discuss the interaction between those endeavours and the intended flow for hardware generation presented here.

### 4.3.2/ IP WRAPPERS DEPLOYED IN THE FAMOUS FRAMEWORK

In this sub-section, we introduce the two types of hardware IPIF-based wrappers used in the FAMOUS framework. As mentioned in the previous section, the rationale behind the use of wrappers is to abstract the low level details of the hardware block through a componentizing process: similarly to the other, non-DPR IPs in the platform, the partially reconfigurable placeholders need to be attached to the bus logic through the use of the Intellectual Property interface, as depicted on Figure 4.3 a). In the FAMOUS framework, we make use of two types of DPR wrappers, depicted on Figure 4.3 b): Wrappers with and without Context Services Support. We discuss the rationale behind each kind of DPR Wrapper as follows.

Traditionally, Dynamic Partial Reconfiguration has been deployed in systems or applications in which different configurations of a given block can be swap in of out of the reconfigurable fabric, as explained in Chapter 1. These systems make no assumptions about the internal workings of the PRMs placed in the FPGA: the high-level application only requires to know at what time the task can be placed (e.g. triggered by an external event, for instance), and the time it requires to me mapped (the reconfiguration time for a chosen Partially Reconfigurable Area) and the amount of time necessary for performing the function (which can be stopped by an interruption, external event or be application dependent). This scenario works well in most current applications, but in recent years, there has been an increasing interest in the creation of Extensible Embedded DPR Platforms, in which an Operating System can take care of the complete application, and wherein the PRMs are seen as mere services (hardware virtualization), which can be stopped and taken out of the reconfigurable region to leave place to other, probably more urgent task. When the second task has been completed, the original task can be resumed, and the context (the internal state of the PRM before it was swapped out) needs to be reinserted. These applications require an entirely different design methodology, and that is the reason we have decided to provide two types of DPR wrappers, so that a designer at high-levels of abstraction can choose which approach undertake.

As depicted on Figure 4.3 b), both wrappers differ mostly in the hardware support for the Reconfiguration Partition (Glue Logic), but the IPIF wrappers contain different functionalities, mainly in the control and protocol signalling. However, it is not the intent of this thesis work to engage in a detailed description of each of the wrappers, for more details,

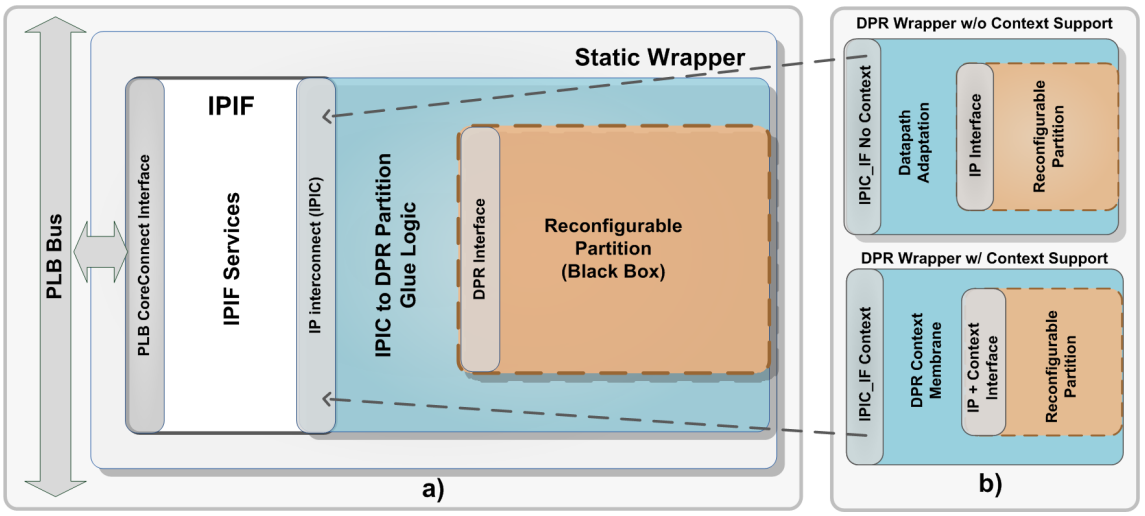


Figure 4.3: a)Generic DPR placeholder model b) DPR Wrappers used in the FAMOUS framework

the reader is directed to previous work [40], and the work of other FAMOUS partners [39], respectively. In this work, we are mainly interested in promoting IP reuse capabilities to the FAMOUS framework and therefore, we mostly deal with the modeling of the IPs at multiple levels of abstraction, in terms of the standardization of the static, DPR wrappers, and PRMs to be used in the methodology. Componentizing each of these heterogeneous blocks permits to associate set of metadata, which can then be exploited for different purposes in a design methodology; as mentioned before, the packaging of this metadata is better facilitated by encapsulation the blocks functionalities into standard IP descriptions (with common interfaces, standard coding techniques, but also the associated metadata representations and underlying metamodels).

The packaging of the IP permits to use it as a plug-and-play block to be integrated/attached to the system bus fabric, thus facilitating its use for building the top-level, static system description of the DPR system. However, the PRMs to be hold by these containers also must be standardized, especially regarding the interface between the static area (the wrapper) and the functionality of the PRM itself. Essentially, the hardware blocks to be used in DPR applications need to be created to be compatible with the corresponding DPR wrapper, depending on the needs of the intended applications, and therefore, the underlying glue logic, as depicted on Figure 4.4. Therefore, the PRMs to be assigned to the Reconfigurable Partition must conform to the corresponding interface. The wrapper depicted on Figure 4.4 represents a module in which no context awareness is supported: only a datapath and the corresponding Finite State Machine to control the write and read operations are provided, leading to a relatively simple interface. On the other hand, a context aware wrapper must support additional features: apart from the datapath for injecting and retrieving data from the HWAcc, the glue logic must support modules for handling the context data internal to the HWAcc, and the associated FSM

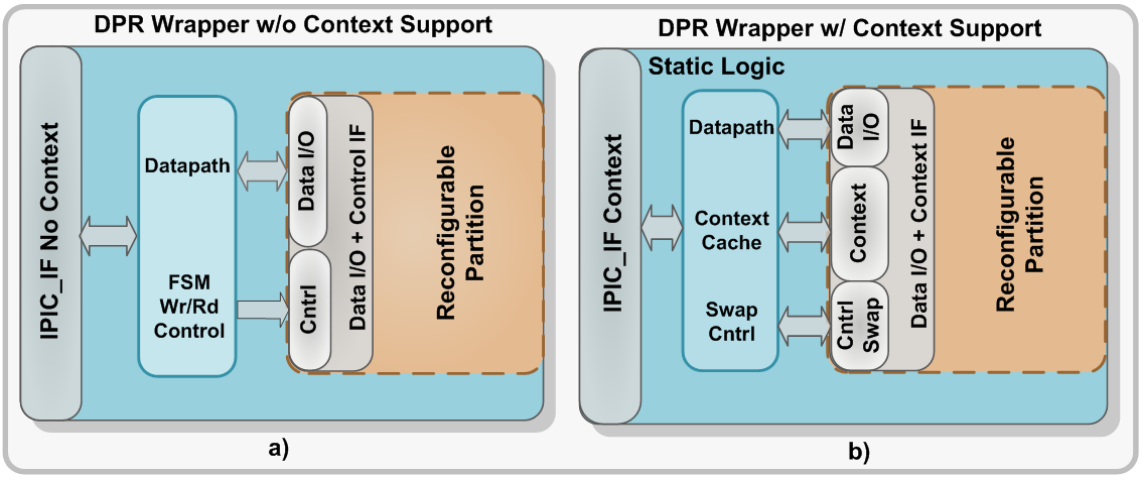


Figure 4.4: a) Placeholder wrapper for IPs without context saving. b) DPR wrapper model integrating reconfiguration services

control machine for the datapath and, in addition, for controlling the states in which the IP might be stopped or not. The DPR wrappers, as will be soon described, require very less parameterization information, which facilitates their use in the flow.

In the sections that follow we describe each of the components in terms that serve the discussion in subsequent sections and chapters. More details of the packaging will be provided in Chapter 5, and the use of the packaged components to build DPR platform will be detailed in chapter 6. Nonetheless, at the end of this chapter we will describe how the components wrapped by the IPIF module are exploited by Xilinx Platform Studio, and furthermore, the way in which we integrate these capabilities into our MDE-based DPR design methodology.

#### 4.3.2.1/ DPR Wrapper without context saving services

As mentioned before, most applications do not require context saving capabilities. For instance, an application might have several implementations of video front and back-ends, which are implemented into the FPGA fabric depending on the type of video signal. These applications do not require saving any state data: the interface and associated logic is simply modified. The same applies to many current applications of partial reconfiguration, which in general consist in mission change scenarios. Therefore, using a DPR Wrapper that contains context saving capabilities might be very costly in terms of the additional hardware resources (e.g. BRAM used for storing the intermediate computation values in a pipeline) required to implement it. Moreover, a Context Wrapper requires many more I/Os in the DPR Interface, which will lead to the use of more partition pins (and therefore, to more LUTs used for implementing them). If this wrapper had to be added to all the reconfigurable slots in a typical DPR application, the benefits of partial reconfiguration, in terms of resources utilization would be severely diminished. There-

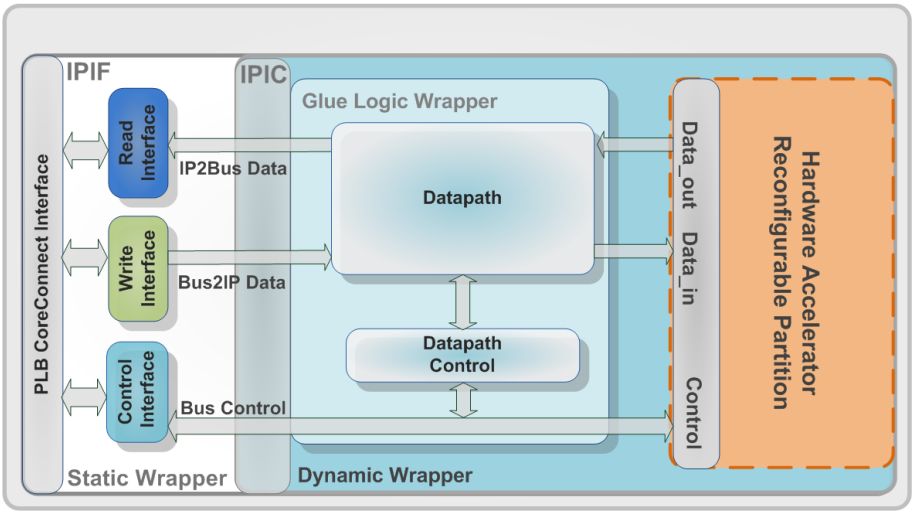


Figure 4.5: The application of the wrapper for a streaming IP

fore, for IPs that require very less interaction with the processor, we provide a very simple interface, as depicted in Figure 4.5, which basically consists on Read and Write Interfaces for interacting with the HWAcc, and a Control Interface that is used for managing these transactions. The interfaces can be configured for single and burst transmissions, through top-level parameterization.

As mentioned before, the datapath needs to be configured as well. Typically, this requires parameterizing the data width for the input and output interfaces. This parameter is in fact dependent in the types of data used for the PRMs to be allocated into the reconfigurable partitions. As we have described in Chapter 1, the interface to the reconfigurable partition must be accommodated in such a way that can handle the different PRMs (in the design partitioning stage of the design entry phase); in our methodology, we provide means to retrieve the interface width information of the different PRMs to be deployed into a Reconfigurable Partition, which is instantiated with a parameters corresponding to the largest width scenario. Furthermore, the PRM descriptions contain latency information that is used for configuring the counters in the FSM Control logic, and this must be taken into account by the application to initialize the corresponding registers after loading a PRM into the FPGA. Such parameters are included as generics in the top-level VHDL description of the module, along other generics for configuring the IPIF logic.

#### 4.3.2.2/ DPR WRAPPER SUPPORTING CONTEXT SAVING SERVICES

The second kind of wrapper utilized in the FAMOUS methodology is the so-called membrane, developed co-jointly by the Lab-STICC partner of the FAMOUS project. The membrane represents in fact the glue logic necessary to interact with context-aware HwACC (IPs for which managing the context is crucial during the run-time of an application). Thus the membrane is composed, apart from the datapath of the conventional DPR Wrapper,

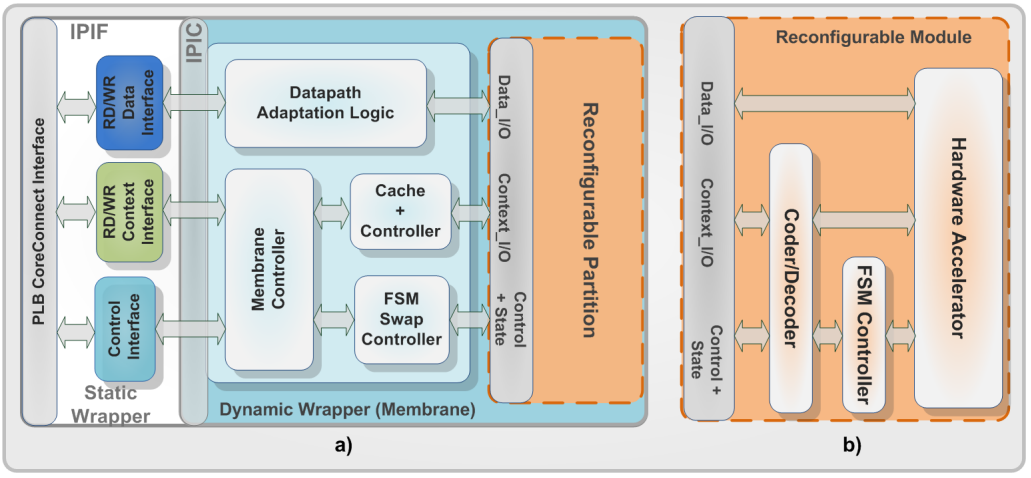


Figure 4.6: The application of the wrapper for a streaming IP

of backup and data context change services, in the form of sub-components attached to the IPIF logic, as depicted on Figure 4.6.

The command controller is in charge of interacting with the application running in the main processor of the application (and reconfiguration controller), decoding different orders and controlling the internal state of the membrane and of the associated HwAcc running at any given time. After receiving an order from the external configuration controller, the command controller makes a demand to the cache controller to check if the necessary IP (and eventually a particular context) are present. In case of cache miss, it will request it to the cache repository in the main system memory.

The membrane model integrates a cache memory able to store a parametric number of contexts, or cache size, which is configured via generics when the DPR Wrapper is instantiated. When this memory is full, contexts are moved to the global memory. This mechanism increases the performance significantly, in terms of reconfiguration time overhead (dependently on the application).The length of a line of memory depends on the needs of the application, and it is also configurable via generics; it should be the maximum size of the largest context width (i.e. PRMs with different context widths, which might be different from the data I/O width). In the case that contexts have important sizes (e.g. Image processing), the designer can define a context which is composed of variables ID and stored data addresses. In that case the context content is stored in the global memory to avoid bulky local memories in the membranes. The main reconfiguration controller writes or retrieves the context of a HwAcc by reading or writing data to the cache memory, but the access to the cache data (for loading, restoring or retrieving the IP context) is determined by the membrane FSM controller, which takes the decoded orders from the Membrane Controller.

This tightly coupled Finite State Machine (FSM) swap controller transits different states (e.g. run, init context, stop, backup context, read context) related to behaviour of the con-

text on a DPR application. One of the main advantages of this FSM is its genericity, as it is common to all components, and therefore IPs designers follow certain guidelines when creating hardware accelerators that are to be compatible with the wrapper, as shown on Figure 4.6 b). As depicted on the figure, the HWAcc (or reconfigurable modules), the IP modules incorporate and internal FSM that controls the inner functionality of the IP, which interacts with the main FSM of the membrane to determine in which moment the IP can be initialized/stopped and taken in or out of the reconfigurable logic. In addition, the reconfigurable modules must contain a coder/decoder module that is in charge of packing/unpacking the state and context data and delivering to the correct interface.

As with the non context-aware DPR Wrapper, the glue logic in this model is encapsulated by using the IPIF module, which has been customized so that the processor can interact (perform write/read operations) with the different components of the membrane. In this manner, the IP can be easily integrated into the SoC platform, but interconnecting the bus interfaces and setting the parameters described above (cache size, context width). As with the previous wrapper, the ports encompassing the IPIF2RP interface are to be configured by controlling parameters extracted from the PRM to be mapped to the Reconfigurable Partition. In this case, the data I/O width and the context data with can be customized (the width of the State interface, as mentioned before remains the same for all IPs). This information is entered in the PRMs configuration (along specific information for the particular HWAcc) and then retrieved during the platform generation phase to configure the RP in the DPR Wrapper.

### 4.3.3/ Relationship of the wrapped DPR components and the design flow

In the previous section, we have described how the different constituent blocks in the DPR Wrappers, along the corresponding Partially Reconfigurable modules have been created to facilitate the integration of heterogeneous FPGA-based SoC platforms. The IPIF module provided by Xilinx tools is used to encapsulate the static and wrappers low-level implementation details, leading to a scenario in which IP blocks become mere functions that can be used by the processor through read/write operations to registers contained in the IP description, abstracting the low-level eccentricities of the hardware implementation. However, the motivation of this thesis work, as stated in Chapter 1, is to facilitate the creation of the hardware platform in the design entry phase of the Xilinx Partition Based Partial Reconfiguration design flow, through a component-based approach. As described in that chapter, the design entry phase of the flow begins with the design partitioning phase, in which the static logic functionalities are defined, and the functions susceptible to be dynamically reconfigured are assembled together into a Reconfigurable Partition, whose interface must accommodate those of the different configurations of a given functionality (Partially Reconfigurable Modules). In a traditional design approach, the designer of such

an application would have to deal with the instantiation of a plethora of components, each of them making use of different interfaces and comprising a great amount of parameters for configuring the underlying IP implementations; furthermore, the interconnection between these components is a very prone to error process. In parallel, the designer needs to create the Reconfigurable Partitions, gathering interface and parameters information of the PRMs to be mapped to the reconfigurable sections of the design. This has the effect of further complicating the creation of complex DPR systems.

In our approach, we simplify the design entry phase by providing a set of pre-created components with standard interfaces, making use of a reduced set of parameters. These blocks represent different blocks: various static IPs and two kinds of DPR wrappers, which can be associated to any PRM created to target such a placeholder (i.e. context-aware DPR wrapper). As described in Chapter 2, the use of component-based approaches facilitates the integration of complex SoC platforms.This is done by dealing with simpler block representations, which can be associated with metadata descriptions (easying their instantiation and parameterization aided by CAD tools). Xilinx Platform Studio provides such EDA capabilities, being the subject of the next section. Here, however, we intend to shed some light on how the componentizing described in the previous sections leads to a simplified approach for the design entry phase of the DPR flow, and in fact, all along the design implementation process.

As depicted on Figure 4.7 a), the design entry phase usually implies the use of an IP library. In the context of the FAMOUS framework, the flow is divided into three branches. The left branch represents the top-level design description, comprised by a set of component instances; in a traditional design flow, this description must be created manually. In the FAMOUS framework, the complete top-level architecture is created from model

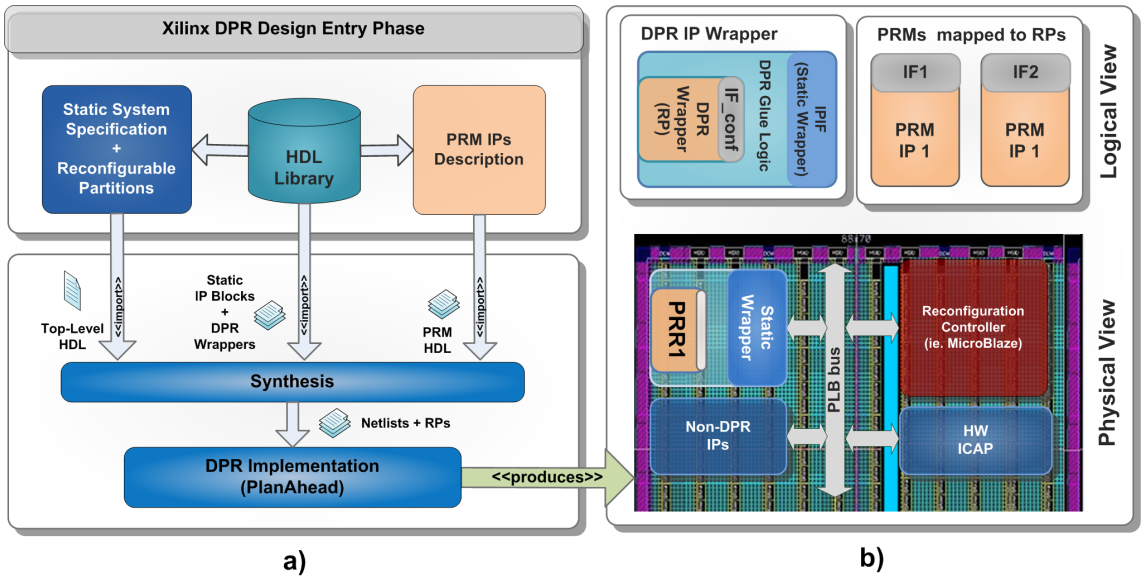


Figure 4.7: Relationship between a DPR IP Wrapper and the underlying implementation

transformations from high-level platform model, and the retrieval of the components, as well as their instantiation, integration and parameterization is done automatically. Among the component instances in the top-level descriptions are the static components (i.e. the ICAP controller) and the DPR wrappers, which are synthesized along the top-level description (second branch) and represent the totality of the static design implementation netlists. These netlists are imported into PlanAhead and the configured Reconfigurable Partitions in the DPR Wrappers are detected as Black Boxes to which some PRM netlists must be mapped.

The PRM netlists are obtained through the third branch of the design entry phase; in fact, the PRM interface descriptions (i.e. I/O data width information) are used for configuring the generics of the DPR wrappers in the top-level description. The PRMs are individually synthesized, and they need to be associated to the correct Black Boxes in PlanAhead, given that the interface information must be correctly inferred, as depicted in the top box of Figure 4.7 b). In the bottom of the figure, the implementation of the different components can be observed: the Reconfigurable Partition corresponds to the Partially Reconfigurable Region that is to be defined during the floorplanning phase of the DPR flow; the PRR is unequivocally associated to a static wrapper (containing or not context management services, depending on the top-level specification, via an IP-XACT ID value), which in fact facilitates the modeling at high-levels of abstraction, since the PRMs to be associated to the DPR Wrapper are automatically associated to the PRR (or Physical Block using PlanAhead jargon). Then, one of the objectives of the FAMOUS framework is to depart from a relatively high-level description of the DPR platform (a set of models) and to obtain a synthesizable description of the system, which serves to obtain to abstract the eccentricities of the flow and the tools involved.

### 4.3.4/ An IP Taxonomy for the Hardware Branch of the FAMOUS Approach

The goal of this section is to summarize the concepts discussed in the previous sections, and to provide an IP taxonomy that will serve as a launching pad for the rest of this thesis manuscript, providing the modeling framework and classification used in the next chapters. In the previous sections, we have discussed how the different components to be used in Xilinx Platform Studio must be wrapped by the Intellectual Property Interface module; these modules comprise the top-level, static section of a DPR SoC platform, as depicted on Figure 4.8 (modules in blue). Among these components are the DPR Wrappers, depicted on the right side of the figure, which act as placeholders for the Partially Reconfigurable Modules and that are mapped to Reconfigurable Partitions contained in the wrappers. The different components in the FAMOUS framework are described as follows.

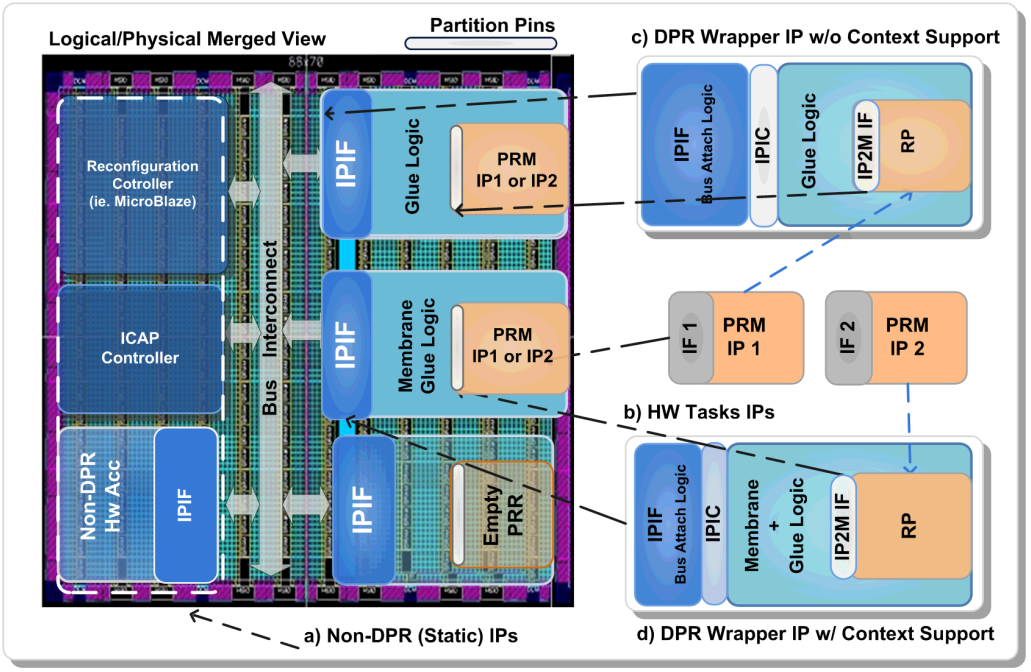


Figure 4.8: Classification of different IPs in the FAMOUS approach

- **Non-DPR (Static IPs):** These IPs correspond to those modules that are not expected to change during the execution of the application. They can be Xilinx IP such as memory controllers, communication or video controllers, both in soft and hard versions; the difference lies mainly in their interconnection and parameterization: hard IPs dot not allow any customization, but only to be integrated into the system. A second category are IP created by third party IP providers or specifically for the application, which have to be imported into the XPS library, incorporating in the process the IPIF logic layer, as depicted on Figure 4.8 a).

  Most of the IPs in the context of this thesis work are soft IPs. This kind of core might provide a wide range of parameterization and customization capabilities, which are controlled through high-level parameters on the component instance, and propagated though the synthesis phase. In this manner, dependent features are integrated into the generated netlist or not, making easy to retarget an IP for different applications. These aspects make part of the standardized coding techniques described in Chapter 2, which facilitate the reuse of IPs by providing a reduced set of configuration capabilities, along simplified interconnection mechanisms, and the abstraction of the blocks implementation by handling only a single, top-level component.

- **DPR Wrapper IPs:** The second kind of IPs are place-holder containers that enable us to incorporate the DPR hardware support aspects in the platform. These modules have been conceived in such a way that glue logic is fully integrated and connected to the IFIP core, thus enabling its interaction with the application via a

processor such as the MicroBlaze. On the other hand, the wrapper contains a fully parametrisable instance of a generic Reconfigurable Partition (RP), that has been designed in a manner that permits to easily plug the PRMs in the execution phase of the application. We have defined two types of DPR wrappers, which have a much reduced set of configuration parameters (as expected for a plug-and-play approach to be used in a high-level methodology).

In both cases (Figures 4.8 c) and d)), the parameterization of the interface to the Reconfigurable Partition (IP2PR IF) is achieved through controlling parameters (contained in the top-level wrapper description) which are resolved from a combination of the interface information of all the PRMs to be mapped to the RP (interfaces IF1 and IF2 on Figure 4.8 b)). This information is stored in IP-XACT component descriptions and retrieved during the system generation phase (achieved through model transformations from the high-level models in UML MARTE), and then used for setting the correct values in the top-level component instance of the corresponding wrapper.

It must be noted that the logic of the IPIF and the associated glue logic belong to the static part of the design (therefore is coloured in blue), whilst the Reconfigurable partition corresponds to the area of the design to be assigned to a Partially Reconfigurable Region later on the design flow. Ensuring that only the PRMs with the correct IP2PR interfaces are associated to the corresponding DPR Wrappers is mandatory, to avoid any communication/interface mismatches. This is done through the ID capabilities provided by IP-XACT to label all the objects described in the standard (thus each interface in a component can be associated to a predefined bus protocol).

- **Hardware Tasks IPs:** These IPs represent the functionalities to be exploited at run time, along the software tasks implemented in the processor, components in the application model in UML MARTE. In a sense, they can be seen as virtual tasks to be used by the processor to accelerate the computation of a particular task.. Therefore, by making this distinction, we separate the real platform components (the static system IPs described in the previous two points) from the blocks that carry out the actual system behaviour. The Tasks IP blocks represent in fact the Partially Reconfigurable Modules in the DPR design flow.

  As explained previously, each PRM to be mapped in a RP must have the same interface for successive configurations, as depicted in Figure 4.8 a). We make it easier for the conception of DPR application by providing a standard interface between the Hardware Tasks and the DPR wrappers, thus enabling plug and play capabilities. The interface information of each PRM is extracted in the system generation phase and used to configure the interface between the placeholder and the RP.

### 4.3.5/ IP Reuse and Design by Reuse in a Component-based Design Methodology

Using the IP taxonomy presented above facilitates the modeling of the IPs at high-levels of abstraction, as well as their deployment (association of the high-level component instances to their underlying implementation artifacts). As with any other IP reuse methodology, classifying IPs into well defined, meaningful groups facilitates their retrieval from the IP library, as depicted on Figure 4.9, which revisits the IP Reuse Design Flow introduced in Chapter 2. However, the classification of the IP alone does not suffice: automatic retrieval mechanisms must be put in place so that the designer of an IP reuse tool can easily select the correct IP. Furthermore, we have discussed the strategies necessary to promote IP reuse mechanisms to design by reuse (or component-based) methodologies, namely the standardization of the IP interfaces and its packaging by well-established bus wrapper interfaces (such as the CoreConnect IPIF), coding and parameterization procedures for facilitating its integration and customization in the intended platform, and the association to parameterized procedures (i.e. scripts) for back-end generation purposes. Nonetheless, all these efforts are useless if, in order to obtain the applicable results of the figure, the user has to deal with the low-level specifics of the component during the instantiation process (as with typical VHDL-based design flows); in an RTL-based flow, the designer of a top-level VHDL architecture has to look at the code of each of the selected components, determine how the ports are to be connected (and to insert signals to this effect), manually access the documentation of the IP for determining the characteristics of the parameters, and furthermore, launch each of back-end generation procedures either manually or via custom scripts.

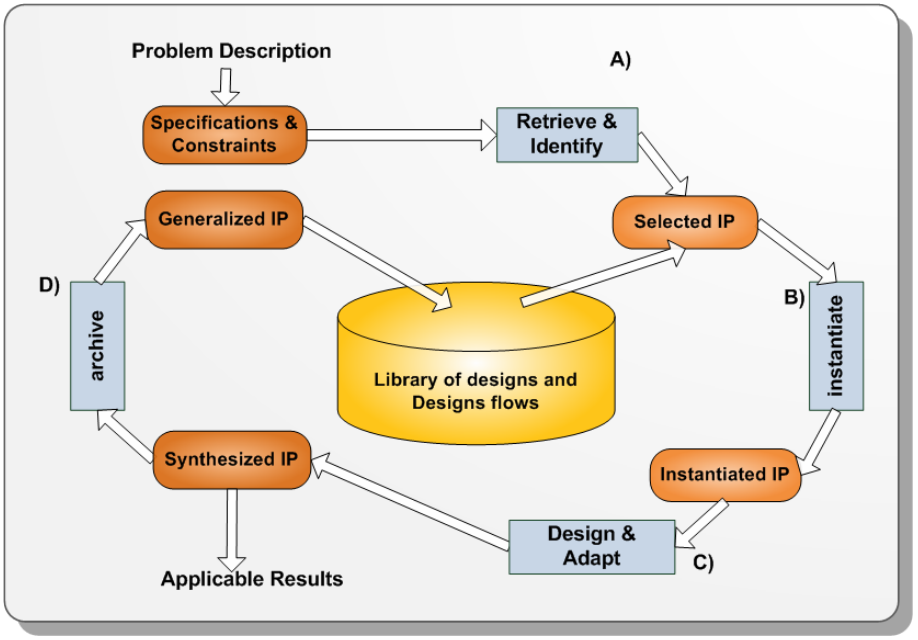


Figure 4.9: Typical IP Reuse Methodology (Revisited)

Therefore, CAD tool vendors have developed Metadata-based Composition Frameworks in order to deal with the design by reuse most tedious and prone to error steps. As discussed in Chapter 2, this is archived by the definition of metamodels for the IP descriptions and the intended back-end platform; the metamodels permit the creation of models that contain metadata that can be exploited for performing the retrieval, selection, and instantiation (parameterization and interconnection) of the IP blocks in an automatic manner. This process in depicted on Figure 4.10. In order to make this possible, the first step is to create a library of reusable IP components with standardized interfaces and parameters (i.e. wrapped IP blocks, VHDL library in the figure) which is exploited, through a data-mining process (IP reflection) to create an IP Metadata Library. This library contains a rich set of metadata used by the CAD tool to perform the tasks described above. The typical process implies the selection of the IP blocks from the library, which are instantiated in the form of IP templates onto an intermediate platform description (conforming to a predefined metamodel); the CAD tool used this platform description through Graphical User Interfaces (GUIs) or other means, to parameterize and interconnect the component instances. The introspective architecture is then used for gathering any dependent parameters, ports and bus interfaces (parsing the IPs detailed metadata), and using a netlist generator, to produce the back-end platform description. This back-end platform description points at the IP implementation files contained in the VHDL library, so the Applicable Results (i.e. FPGA netlists) can be obtained by applying the corresponding design flow and scripts for controlling the desired generator (i.e. synthesis tool).

In the FAMOUS methodology, we have decided to make use of Xilinx Platform Studio (XPS), which is a CAD tool created by Xilinx to perform many of the aforementioned tasks. The rational behind this decision is that currently only Xilinx supports full-fledged Dynamic Partial Reconfiguration, and XPS aids in the creation of FPGA-based Systems-on-Chip, where the hardware and software aspects of a co-design methodology can be integrated seamlessly. This implies that the proposed flow, which departs from a high-level of abstraction in UML MARTE, has to interact with the Xilinx XPS IP and platform metamodels; this means that mechanisms must be provided for passing from UML MARTE to a Xilinx XPS model (i.e. model transformations). However, we have chosen not to plug the UML MARTE generation approach directly to the Xilinx XPS metamodels given that, as discussed before, we aim at using flow-agnostic intermediate IP and system representations, and thus, we make use of the IP-XACT standard. Moreover, we have to emphasize that the use of CoreConnect-based wrappers does not attach our methodology to XPS, since this bus protocol is used by many other SoC vendors; the flow can then be re-targeted to other design flows by choosing the correct back-end metadata information, an IP-XACT capability. These aspects will be discussed in more detail in the following sections, but before, we analyze how Xilinx Platform Studio implements the EDA capabilities depicted on Figure4.10.

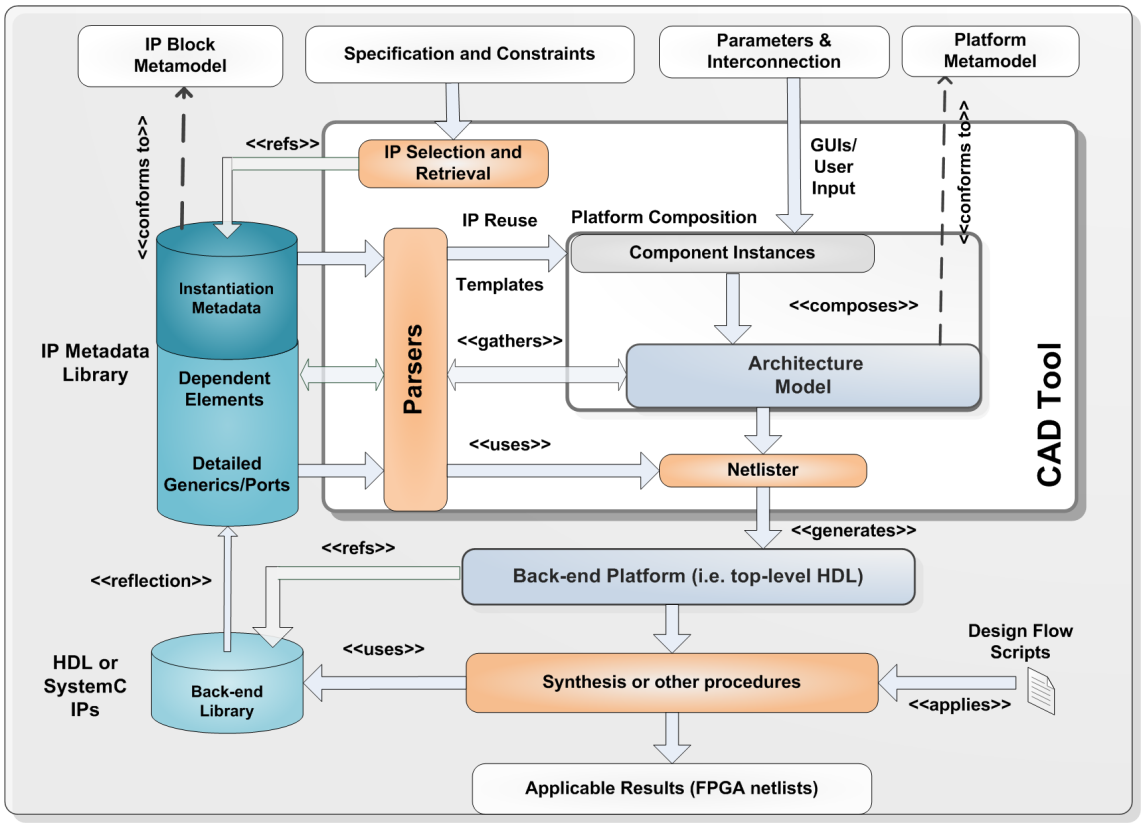


Figure 4.10: General Metadata-driven Composition Framework (detailed)

### 4.3.6/ The EDK Framework for the creation of FPGA-based SoC Platforms

The FAMOUS methodology makes use of component-based approach. The modeler must be able to compose a system using a set of high-level component descriptions in MARTE, making implicit the use of a library containing the IPs to be deployed at lower levels of abstraction. At the so-called deployment level, the model components must contain enough information for parameterization (subsequently used for generation) and, at the same time, provide a link to the actual implementation of the IPs. However, a question arises: how to link the components in the implementation library in such a way that enables their parameterization and customization? Furthermore, if the library is composed solely of the HDL (for an RTL based design flow), how to extract information about the IP from a description that is not intended to do so?

The VHDL description of an IP contains only information about the in/out ports, and in the best case, generics allowing the designer to parameterize and customize it. If the VHDL implementation had to be associated with a high-level description (typically containing parameters and bus interfaces), there will not be an easy and automatic way to determine which ports of the IP belong to a bus interface, and to use additional information important for the design flow. Xilinx solves the aforementioned problems by providing an

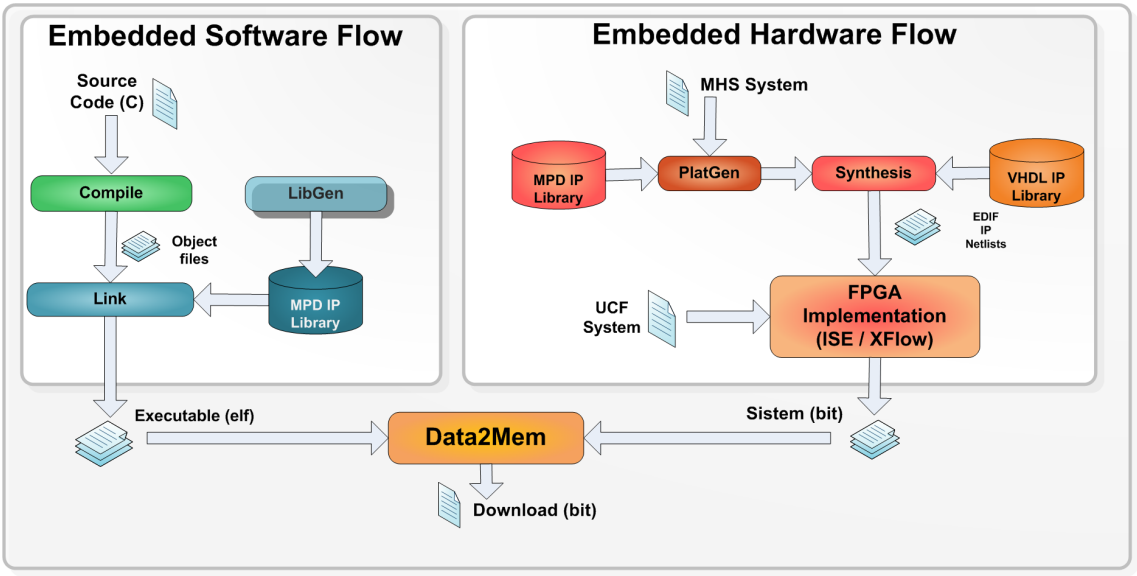


Figure 4.11: Xilinx EDK Design flow for the creation of processor-based FPGA platforms

intermediate representation layer, the Microprocessor Peripheral Description (MPD) file, as depicted in Figure 4.11, which corresponds to the IP Metadata Library described in the previous section. The MPD file contains rich metadata of the underlying IP VHDL/Verilog implementations (generics, ports), adding flow dependent attributes to the parameters and ports, used for configuration. The ports can be bundled together using bus interfaces, allowing the designer to customize the use of certain interfaces by setting a reduced set of attributes. Similarly, parameters and options can be made dependent on other parameters, and attached to specific groups (e.g. parameters that affect certain interfaces but not others, parameters that are only used when another feature have been chosen by the user, etc). An important aspect of the MPD file is that it allows adding information about the IP that is tool/technology specific, which facilitates the configuration of the IP in different scenarios, customizing their behaviour. Moreover, the targeted FPGA family of specific device determines the usability of certain features of the IP; the inclusion or not of these features is controlled via dependent parameters that are enabled only when they are supported by the target FPGA.

In a pure VHDL design flow, the description of the individual components of a platform remains in an IP library. Then, the designer composes a platform by choosing components from it and instantiating; this parameterization and interconnection process of the IPs produces the so-called top level description, which is subsequently used to obtain the netlists used by the FPGA vendor tools to implement the design. The usage of the MPD file in an EDK flow abstracts the low level implementation details, and facilitates the configuration of the IPs, since the tools interact with a machine readable component representation. However, as with the VHDL case, these IP representations need to be parameterized at a higher level; this is done through the components instantiation in the

Microprocessor Hardware Specification (MHS) file. The MHS file represents the Introspective Architecture in the MCF flow of Figure 4.10, and it is created through Xilinx XPS GUIs that allow the designer to create a basic platform, to which other IP blocks can be added from a richer IP library. The Xilinx Platform metamodels as described in the Xilinx Platform Specification Reference Manual [42]; the metamodels are encoded using a set of ASCII commands, and stored in a textual manner. The metamodel describes a set of objects related to the top-level platform description; the first being a set of external ports, those that encompass the top-level design interface. Subsequently, the components instances are listed one by one; each of them contains a list of parameters, a list of bus interfaces and the signals used to connect to other instances in the platform (bundling complex interface signals together facilitates their connection), and a list of ad-doc ports, that can be connected to other instances in the design or to external pins. As with parameters, the presence of individual ports and bus interfaces can be made dependent on the values of parameters in the top-level MHS description, typically used for customizing the IP core functionality (by adding features such as sub-component and their related parameters, ports and bus interfaces).

The MHS and MPD files comprise the hardware branch metamodels of a System-on-Chip design flow. However, EDK is used for implementing processor-based SoCs, where a software component needs to be specified as well. Xilinx divides the design flow for processor based systems in two branches, the Software and Embedded Hardware flows (making use of Xilinx Platform Studio for the hardware generation phase). Since we are discussing the hardware platform generation aspects of the FAMOUS flow, we concentrate in the latter in the discussion that follows. As shown in Figure 4.11, Platgen reads the MHS file as its primary design input. Platgen also reads various processors IP blocks hardware description files (MPD) from the EDK library and any user IP repository referenced in the MHS file. Platgen produces the top-level HDL design file for the embedded system that stitches together all the instances of parameterized cores contained in the system. In the process, it resolves all the high-level bus connections in the MHS into the actual signals required to interconnect the processors, peripherals and on-chip memories. It also invokes the XST (Xilinx Synthesis Technology) compiler to synthesize each of the instantiated IP blocks. (The system-level HDL netlist produced by Platgen is used as part of the FPGA implementation process). In this sense, Platgen acts both as the Netlist Generator and the Back-End Batch of the general MCF flow presented in the previous section. After the netlists of the platform and the IP are obtained, they are used by the MAP and PAR Xilinx tools to map, and subsequently, place and route the design.

The Xilinx XPS intermediate descriptions, based in the MHS and MPD file metamodels, represent an improvement over a purely VHDL description, since the textual representation has a formal semantic, and can be parsed and processed by automated tools. Therefore, being able to handle such Xilinx platform models would enable their transformation to other, high-level models, and more importantly, to integrate them with a

platform-independent standard such as IP-XACT. The rationale behind the use of Xilinx Platform Studio into a Dynamic Partial Reconfiguration design flow has been briefly delineated before: since all the components in the FAMOUS Framework have undergone a wrapping process, they are all (static and DPR wrappers) associated to their corresponding MPD models, which are then used for automatic integration into a XPS-based SoC platform. The Partially Reconfigurable Modules are similarly exploited in the flow; the only difference is that they do not require being instantiated: they are parameterized and the interface information is passed as a constraint for the top-level specification, but they are independently synthesized, using scripts stored in their associated metadata representation.

In the next section, we will delineate how Xilinx Platform Studio and the proposed IPs are integrated into our Model-Driven Engineering framework. The discussion in this section will then serve as a launching pad for the next two chapters, where all the modeling and generation steps are thoroughly described.

## 4.4/ FAMOUS Metadata-driven DPR Composition Framework

In this section, we intend to summarize what we have discussed in the previous sections, by bringing together the Xilinx Platform Studio platform creation philosophy, and the paradigms the MDE and MCF paradigms analyzed in Chapter 2. Moreover, we will shed some light on how the IP-XACT standard is used to glue the entire FAMOUS MDE framework, particularly in the sense of providing a standard intermediate representation that can be used to federate the entire compilation chain. In order to do so, we will make use of a modified version of the MCF design flow, which encompasses the different components of our methodology: front-end (UML MARTE), intermediate representation (IP-XACT) and back-end (Xilinx Platform Studio), and which is depicted on Figure 4.12

First of all, the componentized HDL IP blocks (wrapped by the IPIF module) are stored into the Xilinx XPS library, along their corresponding MPD descriptions (d)). These descriptions could be used directly for composing a platform at high-levels of abstraction but, as described previously, we make use of IP-XACT in order to obtain an intermediate and flow-agnostic (but extended with XPS design flow specific metadata) description; thus, a first IP reflection mechanism has been implemented in order to generate a third level IP library with XMLized IP-XACT component descriptions. This IP library contains the detailed metadata information in the IP Metadata Library of Figure 4.10, and it is used for back-end generation purposes. However, the IP-XACT descriptions are still very rich in terms of metadata and low-level to be handled at high-levels of abstraction; therefore, we have decided to perform a second IP reflection process in order to obtain a library of abstract UML MARTE templates that provide a visual front-end for the designer at high-

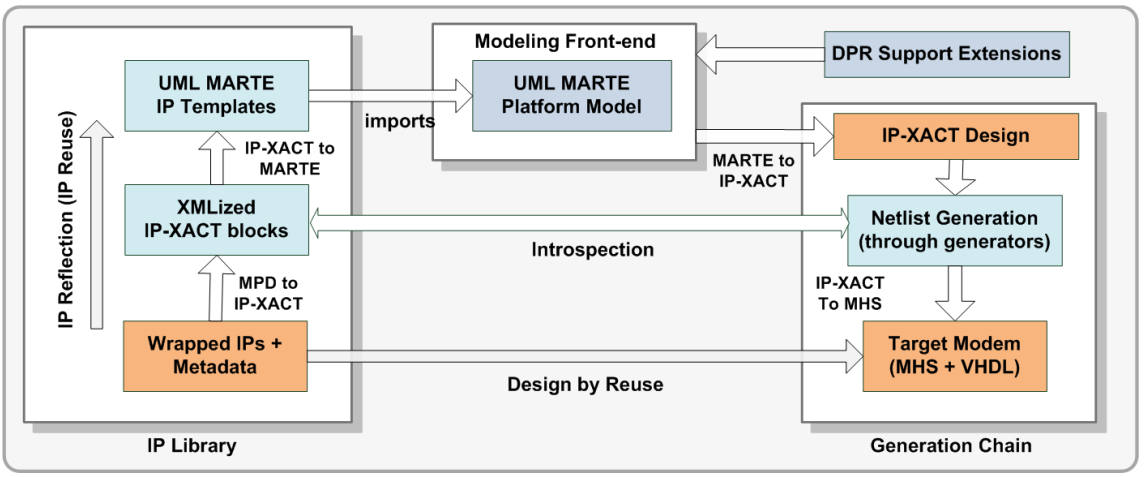


Figure 4.12: Proposed DPR Metadata-driven Platform Composition Framework

levels of abstraction (d)). The IP reflection process in both cases is performed through model transformations, as will be described shortly.

Once the multi-level library has been created, the designer can compose (instantiate and interconnect) a MARTE platform description from the components in the high-level library. The platform obtained in this manner is stored in a non-standard XMI representation (i.e. from a modeler such as Papyrus) and therefore has to be transformed into a standard XML-IR, represented by an IP-XACT design object (through a model transformation from MARTE to IP-XACT). The IP-XACT design object represents the introspective architecture (encompassed by a set of controlling parameters) which is used to parse the detailed IP metadata in the MPD files, generating the MHS platform description, as described in the previous sections. Then, the MHS is fed to Xilinx Platform Studio, in which the top-level VHDL description is generated and synthesized. The PRM VHDL files are retrieved from the UML MARTE models and synthesized independently from this flow, as required by the Xilinx Partition Based DPR design flow.

In order to carry out the totality of the hardware generation phase of the FAMOUS DPR framework, we make use of MDWorkbench, an industrial tool developed by our partner Sodius, which enables the development of metamodels, the import of models conforming to these metamodels, and the definition and application of transformation rules for performing transformation between these models. MDWorkbech is at the heart of the FAMOUS framework, and federates the FAMOUS compilation chain; thus, the diagram depicted on Figure 4.13 shows only the hardware components of the complete methodology.

As depicted on the diagram, the MDWorkbench represents an intermediate step between the high-level models in UML MARTE and the low-level Xilinx Platform Studio representations (and its eventual use in Xilinx PlanAhead in the form of synthesized netlists). The tool has been used to create a set of metamodels, first for all the objects defined in the

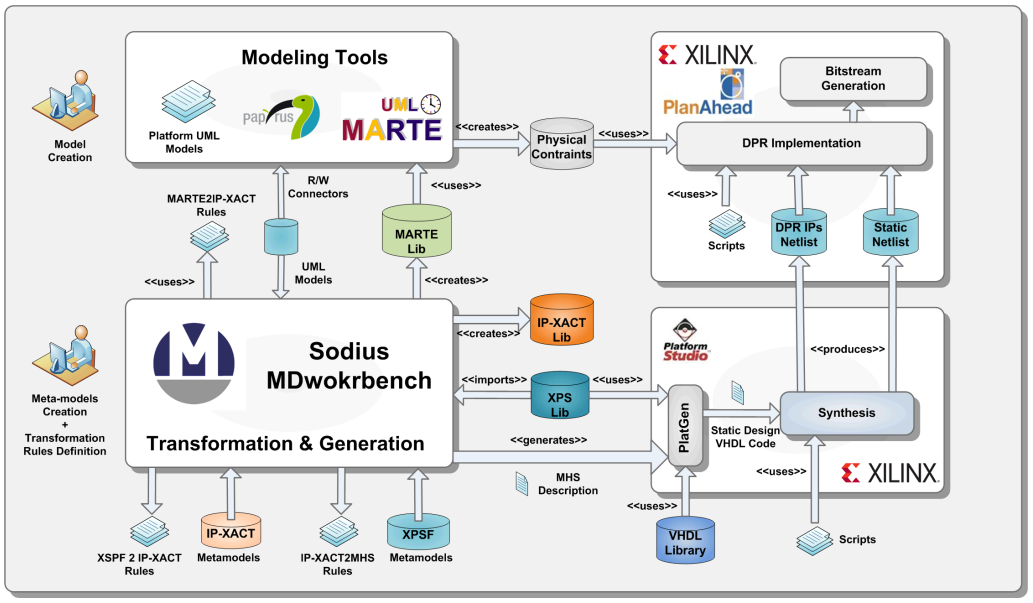


Figure 4.13: Tool support for the proposed DPR platform generation flow

IP-XACT standard (e.g. bus and abstract definitions, components and designs, amongst others), but also for the models used by Xilinx Platform Studio for describing the IP and the platform. A set of transformation rules have been proposed to move from this Xilinx XPSF models to IP-XACT components (for promoting IP reuse), which can be imported into modeling tools such as Papyrus in the form of XML templates for composing the SoC platform. In this manner, we provide the IP metadata reflection mechanisms described before, leading to an IP reuse approach. The modeler at high-levels of abstraction only needs to import the component templates into the platform model and associate the component instances in the UML MARTE diagrams to their IP-XACT counterparts, using an ID mechanism defined by the IP-XACT standard to unequivocally classify al the components in the library. Parameterization and interconnection ensue, using a set of abstract ports and parameters definitions that are propagated to the rest of the compilation chain; of course, this customization and parameterization only applies for configurable IPs.

The UML platform description obtained in this manner is then imported into MDWorkbench in the form of an XMI file and transformed into an IP-XACT design (and analogous of the design capture capabilities of an IP-XACT design environment). Then, model transformations from IP-XACT to XPS MHS are used for obtaining a description used by Xilinx tools to obtain the complete VHDL platform description automatically. As discussed before, proceeding in this manner effectively promotes an MDE philosophy in which a high-level user only deals with abstract concepts, while the metamodeler defines the transformations rules required for moving to different back-ends. The eccentricities of the design tools and the implementation details of the components are hidden to the high-level modeler, and the back-end can be retargeted by changing a small set of attributes and the utilized model transformations.

## 4.5/ Proposed Metadata Driven Composition Framework for DPR Systems

In this section we introduce the hardware component of the FAMOUS MDE framework sketched on Figure 1.14 at the end of Chapter 1. As explained previously, in this work we concentrate in the generation of the DPR platform from a set of components stored in an IP library, along configuration services in the form of DPR Wrapper IPs, to which the Hardware Tasks IP can be mapped. Figure 4.14 a) depicts the hardware MDE branch of the FAMOUS methodology, in terms of model transformations. We propose the use of a double Y-schema, in which in a first stage, through the ⟨⟨allocation⟩⟩ operation, elements in the application are mapped (⟨⟨allocated⟩⟩) to components in the platform. Using the same modeling diagram, the designer of a platform performs a deployment operation, in which the elements in both the application and architecture models are associated to their implementation counterparts (in the form of IP-XACT components that make reference to the HDL implementation files). This is the rational behind the use of the two libraries at the combined ⟨⟨Deployed Allocation⟩⟩ level; it must be noted that the Task IPs in the figure can denote software code to be run on processors, or hardware IPs to be mapped in the DPR Wrapper IPs. However, as we deal with the DPR design entry phase, in the discussion that follows we concentrate purely in the hardware branch of the methodology. In Figure 4.14 b) we depict again the Xilinx design flow for DPR systems, to contrast how the proposed methodology interacts in the creation of DPR platforms.

The Deployed Allocation phase produces a model in which all the information of the com-

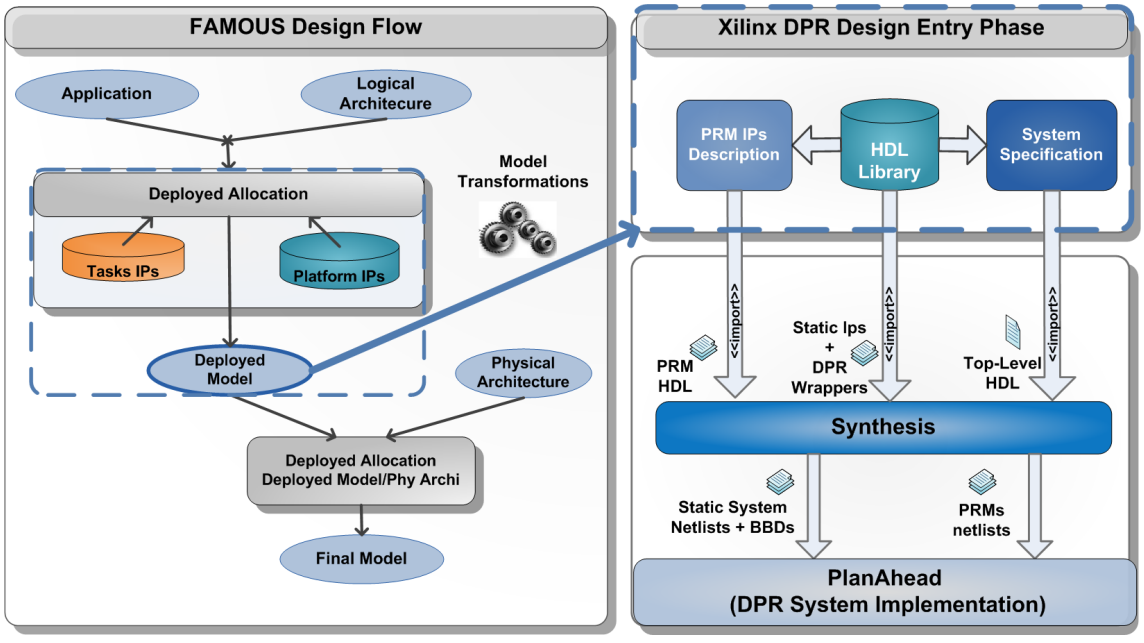


Figure 4.14: Comparison of the proposed MDE approach (a) and its interactions with the DPR design flow (b)

ponents (Static, DPR Wrappers and Hardware Tasks) is readily available and linked to IP-XACT components (and the referenced underlying MPD/VHDL descriptions) that need to be configured. A ⟨⟨Deployed⟩⟩ model, containing this information, along their interconnections (in the case of Static and DPR Wrappers) and parameterization information, can be used at this phase for creating an executable model, effectively generating the outputs of the Xilinx DPR design entry phase. In order to make this possible, the ⟨⟨Deployed⟩⟩ model is fed to a Model Transformation engine, containing a set of metamodels and transformations rules, as described in the previous section. In this context, the Deployed model and the modeling environment can be seen as a Metamodeling-driven Component Composition Framework (MCF) targeting the generation of a variety of proprietary models in its back-end. As described in [40], we target the generation of the proprietary Xilinx Platform Studio [42] models for representing the platform; this facilitates the creation of a framework in which the hardware and software components of a DPR SoC can be jointly developed. Thus, through model transformations, we convert the ⟨⟨Deployed⟩⟩ model, first into an IP-XACT design description (providing a flow-agnostic intermediate representation to our approach) and then the Microprocessor Hardware Specification (MHS) model, which is used by the Xilinx tools to create the VHDL top-level description. In parallel, the Hardware Tasks IPs are synthesized using information contained in their corresponding IP-XACT component descriptions; in particular, the information about the interfaces is used to configure the connection to the membrane contained in the DPR Wrapper. In this manner, we provide a means to generate the outputs of the design entry phase: the complete platform netlist, along the constituent IPs netlists (static blocks, DPR Wrappers containing the associated Reconfigurable Partitions), and in parallel, the netlists for the reconfigurable modules, which are to be used in PlanAhead for generating the configuration bitstreams.

The second Y-schema is outside of the scope of this thesis and it is the matter of future work. However, we can say it aims at allocating the Hardware DPR IPs to partially reconfigurable regions (PRRs) while abstracting this process (for instance, an expert could provide this information as input in he form of an User Constraint File or UCF, where this information is saved by Xilinx tools). In this manner, a complete generation of the DPR platform can be achieved.

## 4.6/ Discussion and Conclusions

In this chapter, we have provided a framework for the reuse of IP blocks into the FAMOUS MDE-based for the modeling and generation of DPR systems. First, we have described how the different IP blocks have been wrapped by the Xilinx IPIF module so they can be exploited by a component-based approach. This component-based approach is based on the MCF framework, in which component models are processed by machines in an

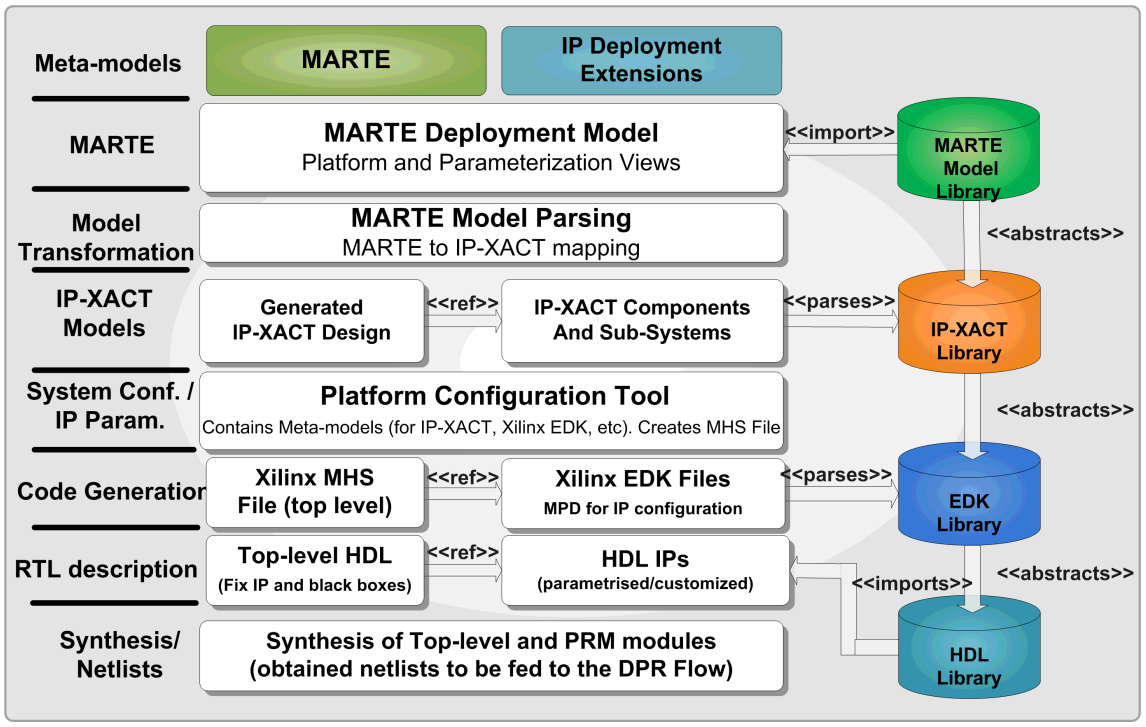


Figure 4.15: FAMOUS Design Methodology in terms of model transformations and levels of abstraction

abstract and automatic manner via the use of metamodels associated to the components low-level implementations. In order to facilitate the task of the designer using UML MARTE, we make use of the Xilinx Platform Studio formalism, which represents itself an MCF; however, plugging the FAMOUS methodology to XPS would make the framework difficult to retarget or to evolve, and therefore, we propose the use of the IP-XACT standard as an intermediate representation for both the IP and the system descriptions. We have delineated which are the requirements of such an approach, both in terms of the models and of the model transformations that need to be put in place for its implementation. The different aspects of the FAMOUS hardware branch are depicted on Figure 4.15

As described in this chapter, the first step for putting in practice a MCF methodology is the creation of library at multiple levels of abstraction. This is facilitated through the use of a component-based approach built over the Xilinx Platform Studio IP representation of the IP blocks. The metadata associated with the Microprocessor Peripheral Definition specifications can be reflected to high-levels of abstraction by defining the corresponding MPD metamodel, along the IP-XACT component and UML MARTE IP Deployment metamodels, to create a multi-level IP library. This is attained by importing the totality of the MPD IP descriptions into Sodius MDWorkbench, and using a set of model transformations (MPD2IP-XACT and IP-XACT2MARTE) to populate the different libraries used in our approach. The creation of the multi-level IP library is the subject of Chapter 5,

where the different metamodels will be discussed in detail, along the proposed model transformations.

The use of the multi-level IP library enables the designer to handle IP blocks in UML MARTE without being exposed to the low-level details of the flow, and provides a means for configuring the low-level implementations of the IPs through the use of an introspective architecture, which is created by assembling components from the IP library, and parameterizing the associated configurable elements. The high-level description is encompassed by a set of models, each of them describing different aspects (views or separation of concerns) of the DPR platform: interconnections, parameterization, the different system configurations or modes, etc. The models corresponding to the DPR platform are imported into Sodius MDWorkbench, and through model transformations (and a set of transformation rules), used for generating the IP-XACT design (introspective architecture) that is exploited, along the corresponding IP-XACT component library, to generate the back-end MHS representation. The generation flow, along the different metamodels and transformation rules are the object of Chapter 6.

# 5

# PROPOSED METHODOLOGY FOR IP REUSE AND SYSTEM COMPOSITION

## Contents

## 5.1/ INTRODUCTION

As mentioned in Chapter 2, modern CAD IP reuse and design by reuse methodologies make use of IP-block metadata for a variety of purposes. Many CAD tool vendors have created their own IP databooks, which are used for automating procedures related with their own flow-specific needs. However, these endeavours have lead to an ecosystem in which IP reuse cannot be fully exploited due to the inherent interoperability issues arising from such metadata representations. The IP-XACT standard has been conceived for exchanging IP descriptions among CAD tools, while providing means for their tailoring to the specific needs of the flows wherein these components are to be used. Thus, commonly used metadata, such as parameters, bus interfaces and ports information can be exploited by tools for tasks such as the automatic instantiation of IP components (i.e. interconnection and parameterization) at a given level of abstraction (i.e. RTL), by using a well established IP metadata standard. This approach has proven itself successful in many SoC-based design flows, where tedious tasks including the integration of IP components to complex bus fabrics (consisting on dozens of signals), their customization, and the computation of the associated map addresses (for HdS APIs), have been all highly automated.

Furthermore, in addition to purely functional, parameterization and interconnection metadata, IP-XACT provides EDA capabilities through the packaging of implementation and tool usage information. This implies that an IP-XACT component description also contains metadata about the design artifacts used for implementing the functionalities of the IP, along how these implementation files are intended to use within a tool flow. For instance, a component description typically contains a set of implementation files written in VHDL or Verilog, along drivers in the form of .H and .C files for communicating with the block from a software perspective. This information (e.g. location in an IP library, dependencies, and possible customizations), can then be stored in the IP metadata description, along point tools information for obtaining concrete results (e.g. synthesis and compilation directives, predefined TCL files). Therefore, assembling all this metadata provides a great degree flexibility for the automation (and eventual creation) of design tools, which is one of the main goals behind the standard. These efforts have fostered a great deal of research in the academia, as discussed in Chapters 2 and 3, with different efforts targeting several domains and back-ends.

In the FAMOUS methodology, we aim at the creation of DPR SoC platforms, following a component-based approach in which the Xilinx Platform Studio tool plays a central role. This tool makes use of its own IP metadata representation, encoded in the Microprocessor Peripheral Definition (MPD) file; such representations work well in the context of Xilinx-based tools, where many of the burdensome tasks for creating a SoC platform have been highly automated. However, plugging an MDE-based methodology to a particular back-end would limit its adaptability to future changes in the FPGA vendors technolo-

gies, and then we have opted for an approach in which IP-XACT component descriptions package the information contained in the MPD files. In this manner, the Xilinx XPS tool specific information is encoded as a part of the IP-XACT component description, while extensions can be added to the IP metadata without compromising the flow-agnostic philosophy of the standard. Furthermore, it enables to add DPR concepts that are typically not exploited in the Xilinx SoC tools, as described in the previous chapter.

Then, by using and XMLized description of the IPs, we provide to our methodology with IP reuse capabilities through IP reflection, by following the MCF paradigm discussed in Chapter 2, and which is shown again on Figure 5.1, albeit in a simplified manner. By reflecting the IP metadata contained in the Xilinx MPD files to IP-XACT component descriptions, we provide a means for visually importing metadata templates into a modeling environment. These IP templates can be converted into visual objects through the use of a UML MARTE metamodel, one of the contributions of this work. A visual representation aids the designer of a SoC platform at high-levels of abstraction at instantiating and parameterizing IP components (thus composing a platform) that otherwise would require a high level of expertise, both in terms of the underlying implementation languages and the used tools to create the SoC platform.

The so-obtained introspective architecture (to be discusssed in the second part of this chapter) can be used for creating the intended back-end system representation through model transformations. We will focus our attention on modeling of the platform in performed in UML MARTE, and how this description in converted into our chosen XML intermediate representation (an IP-XACT design description object). This IP-XACT description is used with two purposes: parsing the IP component descriptions in the IP library for obtaining relevant configuration and interconnection information and, once this information is available, for generating the back-end platform description (e.g. RTL or Xilinx Platform Studio MHS file). Of course, the Partial Reconfiguration Support must be taken into account as well during the modeling phase, but being this a component-based approach, this requirement does not impact the way the platform is created, but

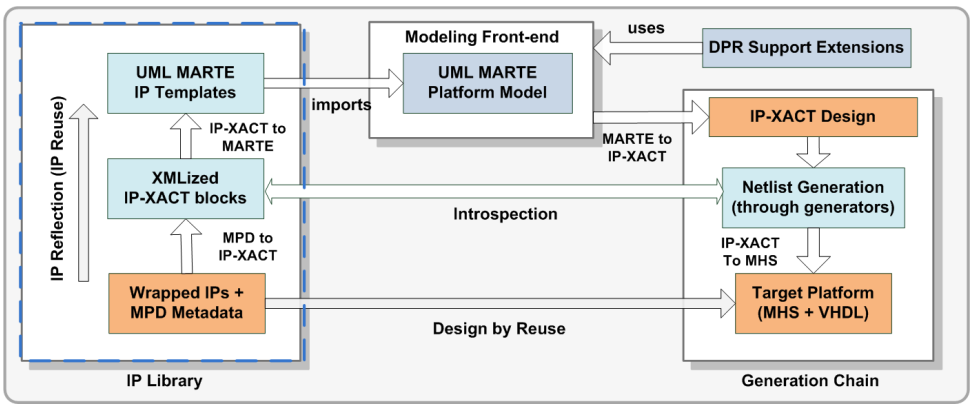


Figure 5.1: Metadata Reflection for IP reuse phase of the FAMOUS Methodology

the generation phase.

Tis chapter is then divided in two parts. First, we engage in a thorough description of the IP reflection mechanisms, from MPD to IP-XACT and from IP-XACT to UML MARTE. Then, we describe how the obtained platform is transformed into an IP-XACT design and subsequently into the Platform Studio MHS file, which is fed to the tool to obtain the synthesizable description of the design.

## 5.2/ Xilinx Platform Studio Back-end IP Representation

As mentioned at the end of the previous chapter; the Microprocessor Peripheral Definition (MPD) file is used by Platform Studio to store IP components information (in terms of parameters, bus interfaces, ports, IO interfaces and finally, design flow/technological options). This information is used to abstract the low-level implementation details contained in the VHDL files, and at the same time, to allow the underlying tools (e.g. PlatGen, using parsers and generators) to retrieve metadata for configuring the aforementioned elements, adding EDA capabilities to the processing of the IP cores. The metadata in this model is structurally encoded in a textual manner (based on ASCII), defining a syntax. This kind of structured textual format can be easily understandable by computers by defining a parser; in this way, Xilinx tools can process the information contained both in the MPD files and the MHS model, which are intimately related. In fact, the MHS, as briefly described in the previous chapter, is encompassed by a set of component instances, each referencing an MPD description (and thus, the underlying VHDL description), as depicted on Figure 5.2

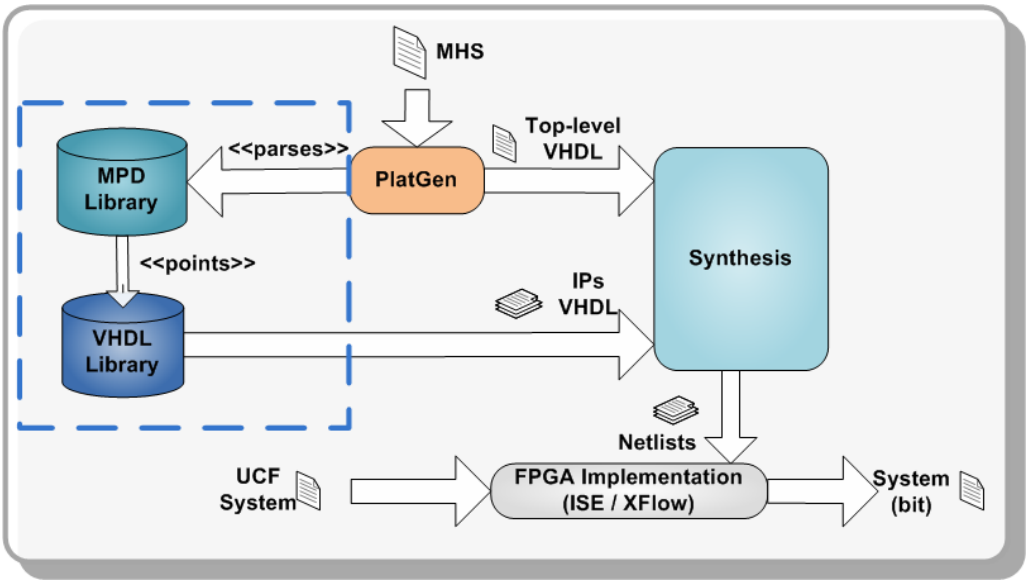


Figure 5.2: Role of the MPD file in the Xilinx Platform Studio Back-end

The MPD file contains rich information (metadata) of the underlying IP VHDL/Verilog implementations (generics, ports), adding flow dependent attributes to the parameters and ports. The ports can be bundled together using bus interfaces, allowing the designer to customize the use of certain interfaces by setting attributes such as DataType, isValid, Permit, etc. Similarly, parameters and options can be made dependent on other parameters, and attached to specific groups (e.g. parameters that affect certain interfaces but not others, parameters that are only used when another feature have been chosen by the user, etc). An important aspect of the MPD file is that it can contain IP metadata that is tool/technology specific, facilitating the customization of the IP in different scenarios. Moreover, the targeted FPGA family of specific device determine the usability of certain features of the IP; the inclusion or not of these features is controlled via dependent parameters that are enabled only when they are supported by the target FPGA.

An example of this feature is shown as well in Figure 5.3. In fact, the controlling parameters are commonly used in the code, along *generate* VHDL directives for enabling the inclusion of chosen features in the final synthesized netlist. As with a pure VHDL code, this is achieved though the use of top-level generics that are set in the component instance; in Xilinx XPS, this is done on the MHS file, which is used as input to the Platgen tool. This tool in charge of configuring and generating the netlists for each of the components in a top-level MHS description; parsers run by PlatGen analyze the metadata contained in the MPD file to seek for any parameters, bus interfaces or ports that depend upon the top-level generic. Then, the MHS file is created in such a way that the underlying IP VHDL implementation is synthesized accordingly to the user requirements. These features will be explored in more detail in second part of this chapter, when addressing the system generation phase.

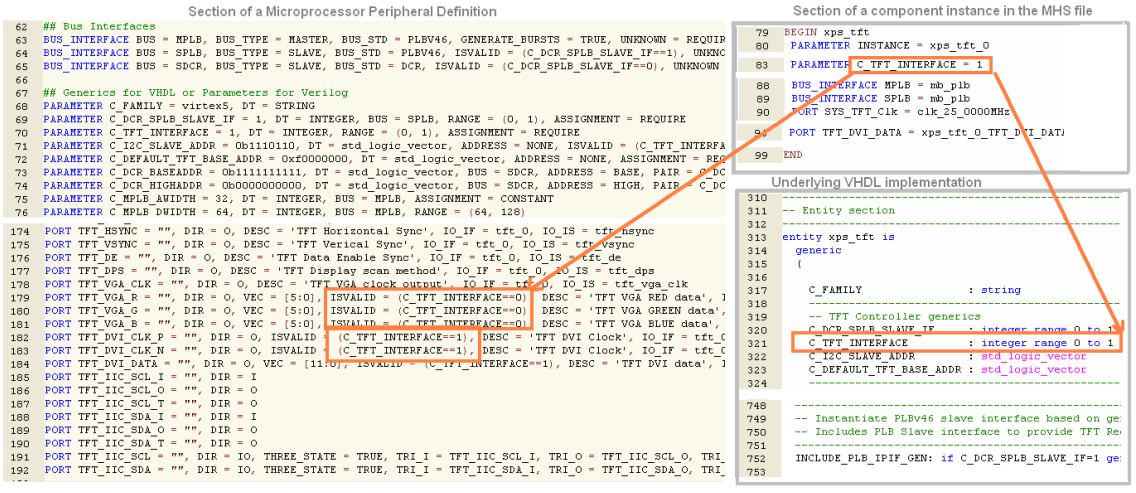


Figure 5.3: Snapshot of the Microprocessor Peripheral Definition file

## 5.3/ CREATION OF THE REUSABLE IP LIBRARY THROUGH METADATA REFLECTION

The EDK intermediate description, based in the MHS and MPD file (among others), represents an improvement over a purely VHDL description, since the textual representation has a formal semantic, and it could be used directly as the intermediate IP and platform representation in the FAMOUS framework. However, we believe that such approach (used in frameworks such as MoPCoM) would severally tie the FAMOUS approach to a set of models that are susceptible to change, especially considering the move of many SoC and CAD vendors to IP-XACT-based representations. We prefer then to foster an approach in which the metadata contained in the MPD files is reflected first to an XMLized IP-XACT standard representation (and IP-XACT component object) and then, through certain features selection, to an UML MARTE XMI template for reusing MDP/VHDL components at high-levels of abstraction. This method decouples the FAMOUS methodology from the chosen back-end, since IP-XACT enables to store multiple views (back-end related metadata) into an IP-XACT component object; then, a particular back-end representation can be obtained by selecting the corresponding view (and the associated generators/model transformations).

### 5.3.1/ MDE-BASED METADATA REFLECTION APPROACH

The FAMOUS methodology is a Model Driven Engineering methodology for the modeling and generation of heterogeneous DPR systems. This thesis work deals with the design entry phase of the DPR flow, which as described in Chapter 1, it is basically a bottom-up approach in which IP components are gathered from an component library to build a platform composed of static and partially reconfigurable modules. As discussed in the previous chapter, we have decided to follow a component-based approach for the creation of DPR Systems-on-Chip, which in the SoC industry is known as IP reuse. Such a methodology assumes the existence of a library of plug-and-play components, with standard interfaces and associated metadata information for automating their instantiation into a top-level description (design by reuse). As discussed in the previous chapter, this is achieved by abstracting the components implementations through the use of well-defined bus wrappers (based on bus standards such as CoreConnect or AXI) for both static and dynamic IPs. In this manner, the IP components become mere building blocks which can be used in by Xilinx Platform Studio to build complex FPGA-based SoC platforms.

However, the philosophy behind the MDE paradigm is to abstract as much as possible the implementation details of the deployed components, as well as the eccentricities of the implementation tools. This can be achieved through the use of a standard intermediate such as IP-XACT, which additionally enables tool interoperability and the creation of flow

agnostic IP and systems descriptions. We have already discussed the benefits of pairing UML MARTE and IP-XACT in the theoretical section of this thesis; however, in order to link the low-level implementation details to their high-level counterparts, the IP metadata needs to be made available at the deployment phase of our MDE approach. Typically, in a purely VHDL flow, this will require to extract metadata from a set of files related to the IP implementation, such VHDL legacy code and documentation files; however, the intermediate models provided by Xilinx Platform Studio make this task simpler, since they already contain a great amount of the required metadata. Thus, we only need to reflect this metadata into a much more amenable and standard IP-XACT component description, whose purpose overlaps with those of the MPD file. The advantage of doing so is that the IP-XACT component description can be made retargeatable by changing the design flow information, while the common elements for instantiation remain unchanged.

As discussed in Chapter 3, many modern MDE methodologies make use of UML MARTE and a modeling front-end for composing platform based on IP-XACT components; in order to do so, the metadata contained in the IP-XACT components has to be available at high-levels of abstraction. However, unlike previous methodologies, we perform this second metadata reflection in such a way that the designer in UML MARTE does not have to deal with the eccentricities of IP-XACT, which contains plethora of low-level metadata. Instead, just the minimum amount of information for instantiating the IPs is imported into a set of UML MARTE templates. In fact, the basic idea is the creation of a multi-level IP library, as depicted on Figure 5.4. At the lowest level, the VHDL component descriptions are already abstracted by their corresponding MPD files; therefore, what the UML modeler is actually dealing with is with the creation of a Xilinx Platform Studio platform description. However, since the MPD metadata has been reflected into IP-XACT and UML MARTE components, the creator of a high-level model is only concerned with very abstract representations of the IP blocks. The IP-XACT component descriptions contain the same information of the MPD files, but FAMOUS-specific metadata is added to the models for targeting different back-ends (e.g. purely VHDL flows forgoing Xilinx XPS); furthermore, the IP-XACT component are better fit for developing compilation chains that are independent from the chosen-back end, which can enable the creation of CAD tools which IP representations can be exchanged, fostering the collaboration in the SoC research community.

In order to perform the IP reflection mechanisms described above, we made use of Sodius MDWorkbench, a tool that was briefly introduced in the previous chapter, and that enabled us to create the metamodels for each of the models in the multi-level library (MPD, IP-XACT component, UML MARTE templates). Then, the corresponding models conforming to these metamodels can be easily imported into the tool, and through model transformations (using a set of transformation rules) each library in the figure has been created. The departure point is a library of static and DPR components with their associated MPD models, which are transformed into IP-XACT components; these IP-XACT components are

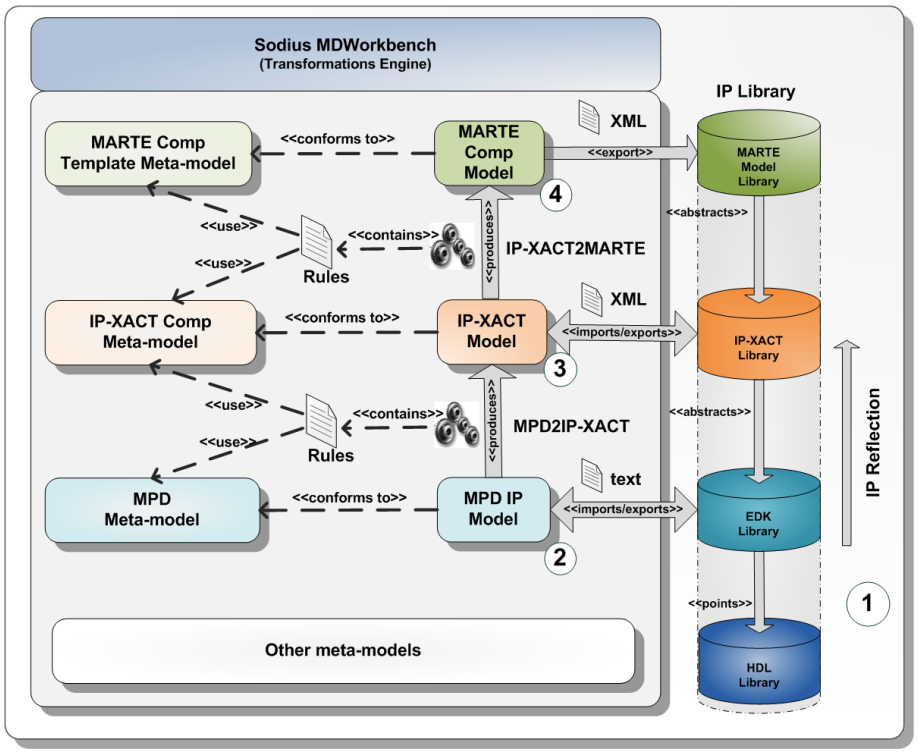


Figure 5.4: Metamodeling-driven approach for the creation of the multi-level library

compliant with the standard and can be therefore imported into other compliant design environments, meaning that non-MDE approaches can also benefit from them. Nevertheless, these IP-XACT components are subsequently transformed into UML MARTE abstract components for its use in an UML Modeler containing the associated metamodels as part of an extension to the MARTE standard.

The creation of a platform from the IP library is the subject of the next section. Here, we will first discuss how the IP-XACT component descriptions are used to package the information of the MPD files, as well as the required extensions in the description to make this possible. Then, we will introduce the metamodels and model transformations in great detail. Finally, we will demonstrate how the IP models presented in the previous chapter are actually represented in UML MARTE.

### 5.3.2/ IP-XACT Representation of the IP Components

An IP-XACT component is the central placeholder for the objects meta-data. Components are used to describe cores (processors, co-processors, DSPs), peripherals (memories, DMA controllers, timers, UART), and buses (simple buses, multi-layer buses, cross bars, network on chip). An IP-XACT component can be of two kinds: static or configurable. An IP-XACT compliant design tool cannot modify a static component. A configurable (or parameterized) component has configurable elements (such as parameters) that can

be configured by the design environment and these elements may also configure the underlying RTL or TLM models.

An IP-XACT component can be a hierarchical object or a leaf object. Leaf components do not contain other IP-XACT components, while hierarchical components contain other IP-XACT sub-components. This can be recursive by having hierarchical components that contain hierarchical components - leading to the concept of hierarchy depth. The IP being described may have a completely different hierarchical arrangement in terms of its implementation in RTL or TLM to that of its IP-XACT description. Thus, a description of a large IP component may be made up of many levels of hierarchy, but its IP-XACT description needs only to be a leaf object as that completely describes the IP. On the other hand, some IP cannot be described in terms of a hierarchical IP-XACT description, no matter what the arrangement of the implementation hierarchy.

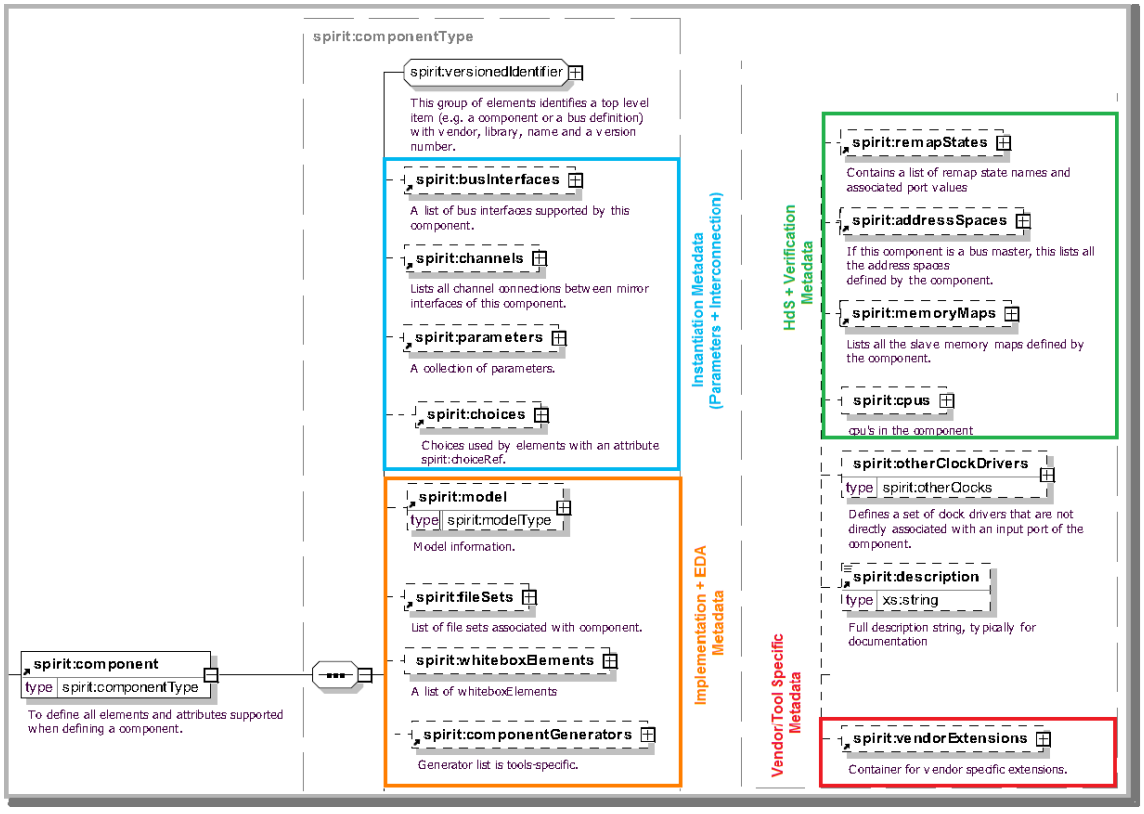


Figure 5.5: Elements of the IP-XACT component description

The IP-XACT XML schemas for the component objects have been conceived in such a way that they cover as much ground as possible, since IP blocks, as mentioned above, can be very heterogeneous. Therefore, many elements with associated metadata sub-elements have been defined, as depicted on Figure 5.5, which shows the first level of the component schema hierarchy. The elements in the figure can be roughly classified in three groups, which have be highlighted (the elements have been rearranged for clarity purposes as well): elements for instantiation and interconnection (⟨*busInterfaces*⟩,

⟨*channels*⟩, ⟨*choices*⟩, ⟨*parameters*⟩), elements for HdS related procedures (⟨*cpus*⟩, ⟨*remapStates*⟩, ⟨*addressSpaces*⟩, ⟨*memoryMaps*⟩) and elements for storing implementation information and promoting EDA (⟨*model*⟩, ⟨*fileSets*⟩, ⟨*componentGenerators*⟩).

Nonetheless, it is important to note that the standard defines most of these items as optional, given that not all concepts might be required in a particular tool/methodology; defining which elements are to be used, and how they will be exploited in design flow (along eventual vendorExtensions) are the tasks of the CAD tool developer. There are, however, certain concepts that are common (and mandatory) in certain flows; for instance, IP blocks to be used in SoC description written in HDL languages require, at minimum, metadata related to the bus interfaces used by the IP, the associated ports and parameterization information.

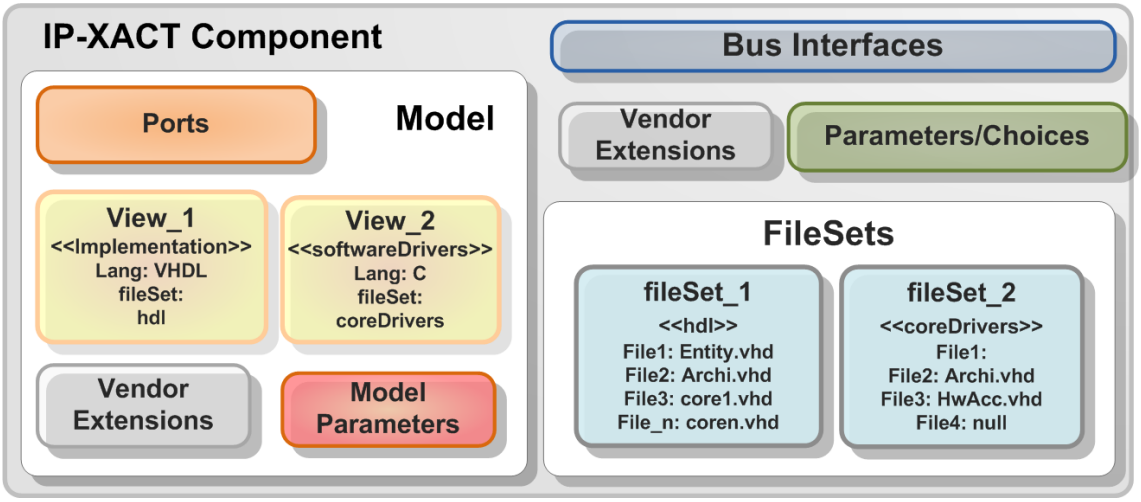


Figure 5.6: IP-XACT component block representation showing the main concepts for modeling

In this work, our main goal it to provide IP and design by reuse capabilities to the MDE-based FAMOUS methodology. Specifically, we are interested in providing a usable flow for easing the design entry phase of the DPR design flow that deals mostly with the instantiation of static and partially reconfigurable hardware components. Therefore, we mostly forgo the HdS aspects of the IP-XACT component descriptions to concentrate in their instantiation and interconnection elements, along EDA capabilities, as depicted on Figure 5.6, in order to enable the generation of Xilinx Platform Studio compatible IP cores and system representations. In the sections that follow, we introduce in some depth how these IP-XACT elements (and several sub-elements) are used, along FAMOUS vendor extensions, to build a library of reusable IP components. We shed new light on the discussion of the previous chapter regarding the modeling of the different IP blocks in Xilinx Platform Studio, and how these concepts are stored into the IP-XACT component databooks for distinguishing between static and dynamic IP blocks, along task IPs. Furthermore, we address the modeling of the different IP-XACT component elements, with the goal

of facilitating its reflection into higher levels of abstraction (UML MARTE templates) by following several design strategies.

### 5.3.3/ Generic model of a Xilinx Platform Studio IP Component

Before engaging in a detailed description of how the IP-XACT objects and the associated schema sub-elements that are used for encapsulating the Xilinx XPS core metadata, we will provide a launching pad for this discussion. In the previous chapter we have discussed the Xilinx Platform Studio philosophy behind the creation of IP components, and furthermore, we have provided an IP taxonomy based on those concepts; as with many other component-based SoC design approaches, wrapping an IP using a set of well-defined bus interfaces eases the instantiation of such components into a complex design, as depicted on Figure 5.7.

Instead of dealing with a plethora of heterogeneous blocks (and several communication mechanisms), the designer of a platform sees the components as plug-and-play building blocks, wherein interconnection and parameterization are greatly simplified. Apart from conventional bus interfaces, a component might contain individual ad-hoc and hierarchical ports, along external interfaces (DVI, VGA, USB, etc); the information of these interfaces must also be inferred automatically, notably any constraints (timing, pin assignments) for facilitating the integration of the IP into the desired platform (i.e. FPGA platform). In Xilinx Platform Studio, the information about parameters, bus interfaces and ports is stored in the MPD description, and as explained in the previous section, it must be packaged into the IP-XACT component description for obtaining a flow-agnostic IP description.

Moreover, by making use of the Xilinx XPS component-based methodology, a single top-level component wrapped by the Intellectual Property Interface (IPIF component) is obtained, simplifying its management in terms of implementation details (by abstracting how the IP core is composed, in some cases containing hundreds of VHDL files). Furthermore, the design flow deals with the configuration of a single entity, propagating the IP top-level parameters to the sub-elements in the component description. Different configurations of an IP can then be easily referenced and retrieved from the IP library, instead of dealing with heterogeneous configurations.

In the previous chapter we have introduced an IP taxonomy for static and Partially Reconfigurable IPs, in which several wrapper designs abstract the low-level eccentricities of the IP implementations (for DPR IPs, by separating conventional reconfigurable IP of context-aware components). This is attained by isolating the IP wrapper from the USER LOGIC section of the Xilinx IP VHDL description, and defining a standard interface for the IP tasks to be mapped to a Reconfigurable Partition. The user logic becomes then a sort of black-box for users at high-levels of abstraction, which only need to assign the desired functionalities to the DPR placeholders; in the case of static or non-DPR IPs, the

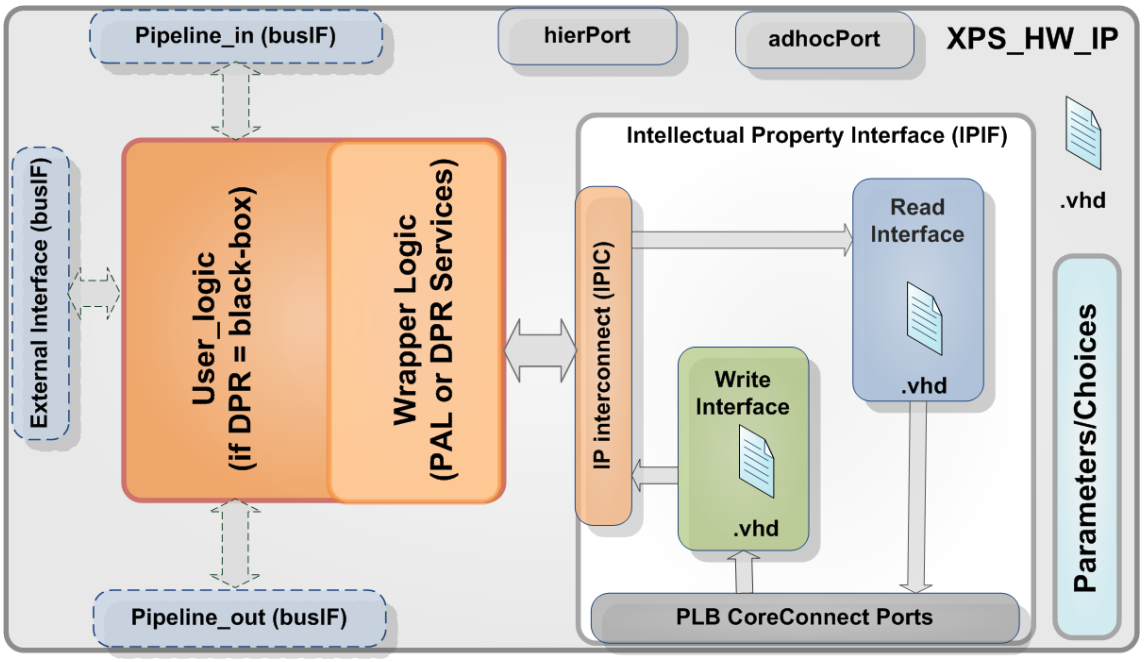


Figure 5.7: Generic Xilinx core and IP-XACT main concepts for a component description

approach is similar, the only difference being that no special care needs to be taken during the synthesis phase. All this heterogeneous implementation information is stored as well in the IP-XACT component descriptions, as briefly introduced in the previous section.

Summarizing, the goal of the following sections is to describe in detail how the different IP blocks metadata has been packaged into IP-XACT components, since this aspect has a significant impact in how the IPs are to be deployed at high-levels of abstraction (UML MARTE). As mentioned before, we concentrate only in the hardware related aspects of the IP-XACT component descriptions, with two main purposes. First, the instantiation information (⟨*busInterfaces*⟩, ⟨*choices*⟩ and ⟨*parameters*⟩) is used for providing a means to compose a top-level description using a design by reuse approach (by gathering components from an abstract IP library). Then, the implementation and EDA information (mainly ⟨*model*⟩ and ⟨*fileSets*⟩) is used for automating the generation of the desired back-end representation (a Xilinx XPS MHS file). We have decided not to dedicate a separate section for the ⟨*busDefinition*⟩ and ⟨*abstractDefinition*⟩ objects of the IP-XACT standard, but to described them in terms of their relationship to the component bus interfaces.

### 5.3.4/ Requirements on UML MARTE modeling of the platform

In Chapter 2, we discussed in great detail how SoC Co-design methodologies make use of IP reuse and design by reuse in order to facilitate the development of complex embedded systems. Then, we introduced the Model Driven Engineering paradigm, which makes use of metamodels, models and model transformations for the generation of artifacts such

as code and the system implementation netlist; this is achieved at the deployment level of an MDE approach, in which elements in the UML MARTE models are associated with components of an IP library. Before the introduction of the IP-XACT standard, many MDE-based methodologies lacked the means for an effective IP reuse and design by reuse, especially regarding the interoperability of the used intermediate representations for the IP and platform descriptions. By using the IP-XACT standard, in tandem with a component-based approach (in which the interfaces and parameterization information of the IP blocks are homogenized), these problems can be greatly alleviated.

In this section, we intend to shed new light into the problematic discussed in Section 2.2.3 regarding the requirements of the IP reuse capabilities that must be provided by the deployment level, in particular in the context of the FAMOUS framework. We will address this issue by showing how the IP-XACT standard and the metadata reflection mechanisms introduced in this chapter, provide the required IP capabilities to our methodology. As discussed previously, the basic requirements for the deployment level are:

**1. Associating IP with properties:** The IP representation at the deployment level must provide the modeler with basic IP properties. In the context of design by reuse, such properties are related to the instantiation of the components, thus the first requirement is the availability of parameterization information. This information is readily available in the IP-XACT component models, and can be easily reflected through parsing the component description; however, not all parameters need to be available to the designer. Some components might not be configurable, and as such, the reflection mechanism must only provide the designer with parameters that must absolutely be set in order to obtain a usable version of the IP. This is attained through the classification of the parameters in the IP-XACT component description into well defined groups, as will be discussed in the section. The second requirement for instantiating an IP block is the availability of interconnection information; the modeler in UML needs to interconnect IP blocks in a relatively abstract manner, without being exposed to the low-level implementation details of the interface.

**2. Component re-utilization:** The modeler in UML MARTE must be able to associate (i.e. deploy) an instance in the UML platform model to an IP in the component library. MDE methodologies have traditionally performed this mapping by providing the path to the implementation files (i.e. pointing where the code artifacts are stored); however, this approach does not promote component reutilization at all, since it assumes that the components are ready to use, without undergoing a parameterization phase. Most components in SoC platforms provide high degrees of customization, both in terms of the desired functionalities and in the way they can be interconnected. As discussed before, the IP-XACT standard provides a means for unequivocally identifying an IP in the component library, the VLNV value; by associating an UML MARTE instance to a VLNV, the correct IP information can be make available to the designer.

**3. Abstraction of associated functionality:** IP-XACT supports the concept of implementation view for supporting multiple back-end generation scenarios. For instance, a component might have several implementations (e.g. for different back-ends, different power consumption or the offered services) at the same abstraction level (RTL). On the other hand, an IP can be described using RTL and SystemC, and the user of the IP model must be able to choose the view that suits him best for the intentions of the model. IP-XACT abstracts the associated functionality of a view by hiding the implementation details (fileSets and buildCommand metadata) away from the user; thus, when a modeler in UML selects a particular view, the back-end information is automatically used for generating the associated functionalities.

**4. System Composition and Generation:** As mentioned in the first point, the IP deployment must provide mechanisms for instantiating IP components at high-levels of abstraction. This is analogous to a component declaration and instantiation in a VHDL top-level architecture. The main difference is that in UML MARTE, the information available to the modeler is much more abstract; the implementation details are gathered in a subsequent stage, when the introspective architecture obtained though composing the platform, along the component deployment information, is imported into an IP-XACT compliant design environment. As discussed before, this is attained though transforming the MARTE XMI system representation into an IP-XACT design; this is analogous to the design capture capabilities provided by IP-XACT compliant CAD tools, which enable to import IP-XACT component descriptions for system integration.

In order to provide these capabilities to UML MARTE, the metadata from the IP-XACT components must be reflected into a set of UML MARTE templates. These templates are described using an IP Deployment metamodel, which represents a much reduced sub-set of the elements in the IP-XACT component description. These templates are obtained by a model transformation between the IP-XACT components obtained from the MPD library, by a second metadata reflection stage, as depicted on Figure 5.4; In order to do so, an IP deployment metamodel must be defined. This approach has the advantage of hiding the eccentricities of the IP-XACT component schema to the user at high-level of abstraction, in contrast to some of the approaches analyzed in Chapter 3, which made use of complete IP-XACT metamodels. The IP Deployment Metamodel will be therefore described in the next section, showing how the IP reuse requirements described in this section are met, improving at the same time the high-level modeling process.

## 5.4/ META-MODELS FOR EACH OF THE LEVEL IN THE LIBRARY

The rational behind this packaging is then to be able to create a minimal metamodel for representing the IP-XACT components in UML MARTE, for providing the modeler at high-levels of abstraction with just the enough information about the IP in order to instantiate

it in a very abstract manner. This implies, as discussed previously, its retrieval from an abstract UML library, its instantiation (interconnection and parameterization) with the goal of obtaining an abstract representation of the SoC DPR platform. Given that the abstract components contains references to the bus interfaces, ports and parameters of the underlying IP-XACT component descriptions, the so-obtained introspective architecture can then be used for the back-end system generation.

### 5.4.1/ Microprocessor Peripheral Definition metamodel

However, if these modes are to be integrated into an MDE methodology, the definition of the corresponding metamodel is a necessity; this is attained by defining the relationships between these elements using an UML modeler, and then importing the UML model of the MPD metamodel into Sodius MDWorkbench. The tool can, as mentioned in the previous section, import models that conform to the different metamodels used in the FAMOUS framework for model transformations purposes.

The Ecore formalization of this metamodel was not available, and thus, the MPD metamodel has been created using the Rhapsody UML Modeler and imported into MDWorkbench. The so-obtained metamodel is depicted on Figure 5.8; as shown, a set of Platform Elements can be present in a SoC platform; these platform elements are labelled as Peripherals (actually, any IP connected to processor sub-system via a bus interface). Each Platform Element in a Peripheral Definition contains a set of attributes (Parameters, Bus Interfaces, Ports, Options and IO Interfaces) which encompass as well a chain of commands (a Value:Name pair, separated by commas). These commands are EDA metadata used in the processing of each of the elements in the MDP description by the Xilinx CAD tools, as will be briefly detailed soon after. However, before doing so, we provide an example of how the MPD looks like on Figure 5.3.

Then, we proceed to describe how the elements in the MPD file are used to store information of the IP, for a variety of tasks in the Xilinx XPS design flow. This discussion is not intended to be exhaustive, given the complexity of the Xilinx IP representation and its use in the XPS design flow. The main goal of this section is to provide glimpse in how this metadata is used, and then in the subsequent sections, how it is packaged into IP-XACT components. For a complete description of the MPD file, the reader is directed to the corresponding chapter in the Xilinx Platform Specification Format Reference Manual [42]).

**Bus interfaces:** This concept provides an abstraction used to bundle a set of signals that belong to a predefined bus standard. This is similar to the ⟨*busType*⟩ element in the IP-XACT component description. The basic principle is the bus interface can then be applied as a classification label to the PORTS description section of the MPD file. As with the ⟨*busType*⟩ element, the BUS INTERFACES elements in the MPD description provide a high-level abstraction for the ports that encompass it. When Xilinx XPS processes an IP

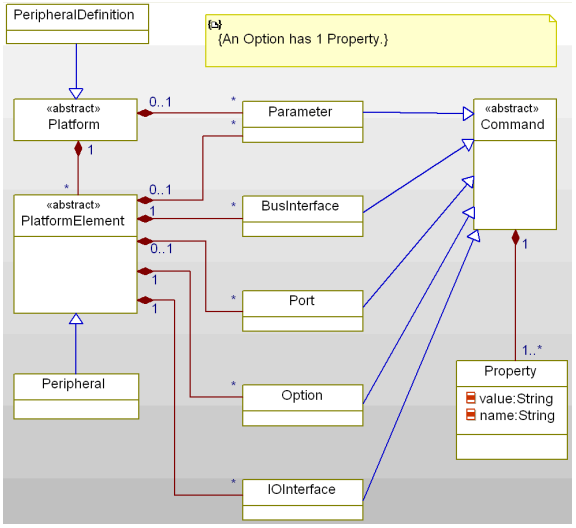


Figure 5.8: UML Model of Xilinx PSF Metamodel - Microprocessor Peripheral Definition (MPD)

MDP description, it looks for which interfaces the component contains, and generates a VHDL wrapper in the top-level SoC RTL description accordingly. Some of the commands used for the BUS INTERFACE are BUS (for naming the interface), BUS_STD (for associating the interface with a predefined bus definitions, such as DCR, MLM, PLB, FSL, AXI, etc), among others. On command of particular interest is the ISVALID tag, which can be also associated to PORTS and PARAMETERS and whose function is to provide a means for customization of the IP. This command defines the validity of the associated elements of an expression; if the expression evaluates true, the related element (in this case, as BUSIF) is included in the list of valid elements of the IP implementation. This can be used for selecting different kind of interfaces depending on the demands of the application.

**IO Interface:** These elements are similar to bus interfaces, in the sense that are used to bundle a set of signals into a single entity. Furthermore, the set of signals the interface describe do not belong to a predefined bus standard, but to predefined input/output interface, typically to off-chip devices. Examples of these interfaces are the IO signals to the TFT Controller, the DDR memory, the TEMAC Ethernet controller, etc. IO INTERFACES contain only two commands IO_IF (a user defined name for the external interface) and IO_TYPE, which is a label stored in the Xilinx Board Definition (whose intent in otside of the sope of this manuscript, but represents a sort of top-level component description for a chose FPGA board). As with BUS INTERFACES, the IO_IF can then be applied to the PORTS in the MPD description for categorizing them into predefined, meaningful groups for EDA purposes.

**Parameters:** Define values used by the tools to configure the IP. The commands for the PARAMERS are manifold, and as such, we will describe which we consider the most important. First, the Xilinx tools need to know how to process a parameter; for this purpose,

the ASSIGNMENT command classifies the PARAMETERS into four groups: constant, optional, require and update. The first two classifications imply that the user does not need to take care of the values of the parameters (in the first case, the parameters is propagated automatically, while in the second scenario, a default value can be used). Regarding the other two types of assignments, they correspond to configurable PARAMETERS, the first by the user and the second by batch tools, respectively.

The DT (DataType) command specifies the type of the value hold by the PARAMETER value; examples of DT are bit, integer, string, std_logic, and std_logic_vector. Related commands are MIN_SIZE, RANGE and VALUES, which define the minimum value, the range the value hand hold (defined as a vector) and an enumerated list of choices, respectively.

A third major classification of commands is the PERMIT and ISVALID pair. The ISVALID command acts in the same way as described before for BUS INTERFACES (and is also used for PORTS), and enables the inclusion of a PARAMETER in top-level instance depending on the value of other, controlling parameters in the MPD description. Typically, Xilinx tools will parse the MPD file looking for these dependent PARAMETERS, evaluating the expression contained in the value part of the command and including those that are valid into the top-level instance in the MHS. More details about this process will be provided in the second part of this chapter. Finally, the PERMIT command specifies PARAMETERS whose values can be modified using Xilinx GUIs; these are typically the controlling parameters used in the ISVALID expressions and thus are essential in the configuration/customization of the underlying IP component.

Other commands can be attached with the PARAMETER elements, such as BUS, IO_IF and IO_IS tags for specifying that the parameters are associated with to those elements and must be processed along. This is usually helpful for deciding whether or not a PARAMETER is included in the top-level component instance; for example, if a PARAMETER is associated with a given BUS INTERFACE and this one is not used in the design, then the corresponding PARAMETERS are ignored when parsing the MPD file.

**Ports:** These elements describe the ports of an IP and its characteristics in an individual basis. Many of the commands used for PARAMETERS and BUS INTERFACES are used as well for PORTS (ASSIGNMENT, BUS, IO_IF, IO_IS, PERMIT, and ISVALID) for the same reasons explained above (ports classification and enabling IP customization). However, many PORTS cannot be bundled to BUS INTERFACES and IO INTERFACES (such as clock, interrupt and reset signals); this is due to the heterogeneous nature of a port properties. Therefore, metadata that cannot be captured in VHDL is associated to each PORT depending on its type; for instance, for a CLK port, commands such as CLK_FACTOR and CLK_PHASE are used for providing additional information about the clock properties. The classification of the PORTS is done via the SIGIS command, which can take the following values: CLK, INTERRUPT, and RST; depending on the SIGIS

value, the aforementioned additional commands are attached to each class of port. The same applies for PORTS belonging to BUS and IO interfaces: depending on its type, the associated commands are created.

**Options:** Options are similar to parameters. The main different is that options deal more specifically with flow or technology depending aspects related with the IP. These OPTIONS are used by Xilinx tool to process the MPD files and the underlying IP implementations, and represent specific Xilinx XPS EDA information.

In the next section, we will introduce how the IP-XACT component description is used to accomplish a similar task to the Microprocessor Peripheral Definition. The creation of the IP-XACT components from the MPD descriptions (IP metadata reflection or packaging in IP-XACT jargon) is achieved through and MDP2IP-XACT model transformation; the transformation rules describing the mapping of the concepts discussed above, and those of the IP-XACT component description will be described in the next section, where more insight will be provided on how the MPD commands are used for providing EDA capabilities to the Xilinx XSP framework.

### 5.4.2/ IP-XACT Component metamodel

As discussed in Chapter 2, the IP-XACT specifications provide a set of XML schemas (.xsd) for representing different concepts in SoC design. Some of these objects are the Bus and abstraction Definitions, the Component object, and the Design description. This set of XML schemas has been processed by the improved XSD/Ecore meta-model importer in MDWorkbench, which leads to a Java/EMF implementation of the IP-XACT meta-model. The different metamodels are used for different purposes, in this chapter; we make use of the IP-XACT component metamodel, along the Xilinx MPD and UML Component Deployment Metamodel, to create the multi-level library. The IP-XACT design metamodel, along the UML Platform and Xilinx MHS metamodels are exploited for generating the system back-end representation, and will be discussed subsequently.

#### 5.4.2.1/ Bus and Abstract Definitions

In IP-XACT, a group of ports that together perform a function are described by a set of elements and attributes split across two descriptions, a bus definition and an abstraction definition. These two descriptions are referenced by components in their bus interfaces, via a VLNV unique identifier.

The bus definition description contains the high-level attributes of the interface, including items such as the connection method and indication of addressing. The sub-elements in ⟨*busDefinition*⟩ are shown in the left side of Figure 5.9. As with the BUS DEFINITION elements in the MPD file, this IP-XACT object contains a set of attributes that help to describe

a bus standard; for instance, indicates if the ⟨*busInterface*⟩ to which this ⟨*busDefinition*⟩ is attached is addressable (to indicate whether or not memory map information can be traced through this interface). Interfaces connected to buses are usually addressable, in the sense that they are used for accessing registers in the IP that are mapped to memory; however, certain external or internal interfaces denote direct connections between components in the systems or to external, off-chip devices, and thus they do not require to be addressable. Other two elements are the maximum number of slaves and masters that can be connected to a bus, ⟨*maxSlaves*⟩ and ⟨*maxMasters*⟩, respectively.

Then, an abstraction definition object, as depicted on the right side of the Figure 5.9, contains the low-level attributes of an interface, including items such as the name, direction, and width of the ports. This is a list of logical ports that may appear on a bus interface for that bus type, and represents the implementation details of the ⟨*busDefinition*⟩; in fact, an ⟨*abstractionDefinition*⟩ must be linked to the associated ⟨*busDefinition*⟩ via the ⟨*busType*⟩ sub-element, which is actually the VLNV reference to the associated bus definition type.

The most important element of the ⟨*abstractionDefinition*⟩ object is the ⟨*ports*⟩ section. These elements contains information about the abstraction of the interface, by defining a set of ports as being both or either ⟨*wirePorts*⟩ (a physical port for an RTL description) or ⟨*transactionalPort*⟩ (describing its functionality in terms of the services provided by the port). Since we are dealing with hardware components at the RTL level, we make use of ⟨*wirePorts*⟩ exclusively. This sub-element provides enough information for describing the characteristics of a port in a VHDL entity (direction and width), as depicted on Figure 5.10 a). Moreover, the ⟨*wirePort*⟩ element can be attached to particular sub-elements of the ⟨*asbtractionDefinition*⟩ (master, slave or system interface), as depicted in the example

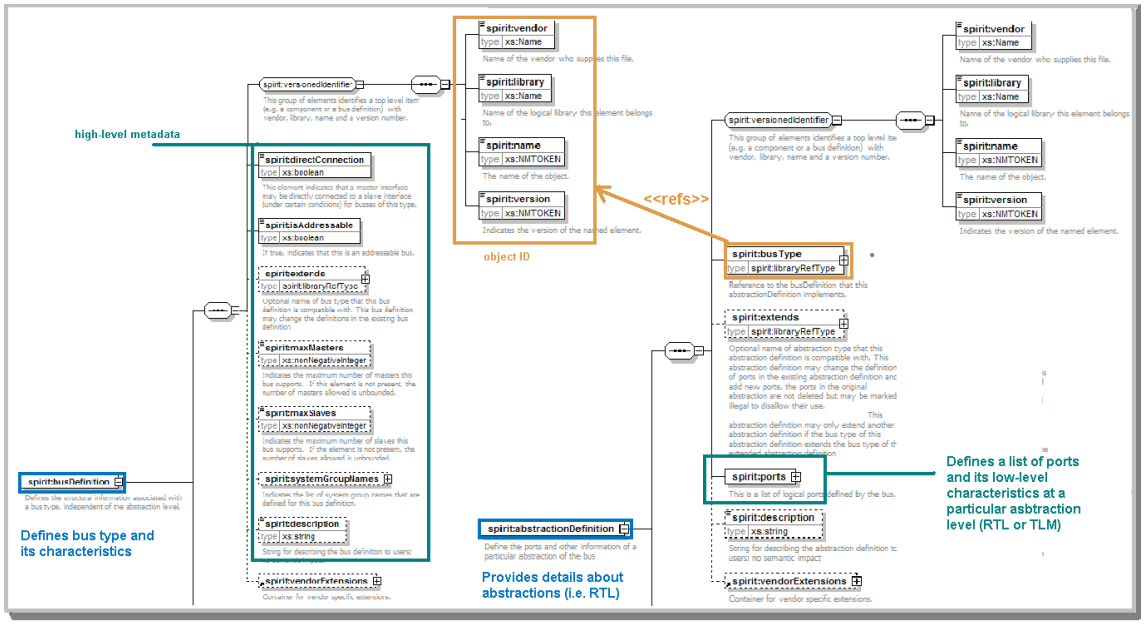


Figure 5.9: IP-XACT schemas for: a) Bus definition and b) Abstraction Definition

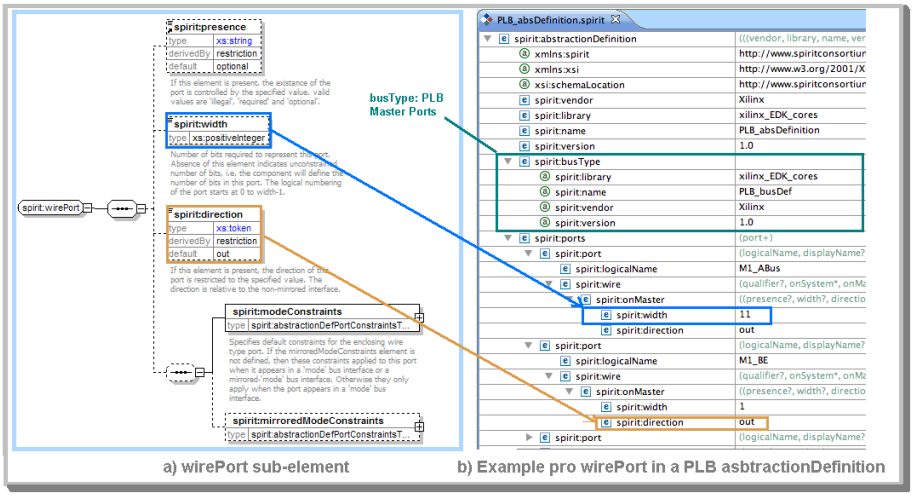


Figure 5.10: Example of a wirePort element (a) and its use in an asbtractionDefinition (b)

of Figure 5.10 b).

We further constrain the definition of ⟨*busDefinitions*⟩ and ⟨*asbtractionDefinition*⟩ to those bus interfaces supported by Xilinx Platform Studio, namely CoreConnect PLB and AXI. Furthermore, we have similarly created bus definitions for other non-bus interfaces used in FPGA platforms, such as DVI (for video output), UART, SystemACE (for external flash memories), among others. However, non-Xilinx interfaces can be defined, if the intended back-end is not to be Xilinx XPS, but a more general RTL generation design flow. In the next section we will discuss how component descriptions make use of ⟨*busDefinitions*⟩ and ⟨*asbtractionDefinitions*⟩ for abstracting the implementation details of the associated ⟨*busInterfaces*⟩.

#### 5.4.2.2/ Bus Interfaces and Ports

Each IP core in a SoC system normally identifies one or more bus interfaces. Bus interfaces are, using the IP-XACT jargon, groups of ⟨*ports*⟩ that belong to an identified bus type (a reference to a ⟨*busType*⟩, which references itself an ⟨*abstractionType*⟩, for RTL or TLM). The purpose of the ⟨*busInterface*⟩ elements is to map the physical ports of the component to the logical ports of the ⟨*abstractDefinition*⟩. This is achieved through the ⟨*portsMap*⟩ element of the component description.

Furthermore, ⟨*busInterfaces*⟩ enable individual ports that belong to a slave or master interface to be grouped together into well-defined data collections; this capability significantly reduces the complexity of connecting an IP component when instantiated in an IP-XACT design object, given that EDA tools deal with interfaces in an abstract manner, leaving the ports low-level details in the abstract definition description. When a particular back-end is chosen during the design build phase, generators access the latter in order to create the top-level design description (for instance, port information in component dec-

larations). Since the FAMOUS methodology targets the generation of Xilinx XPS IP and platform metamodels, this information is already abstracted by the MPD file, making the transformation easier to accomplish.

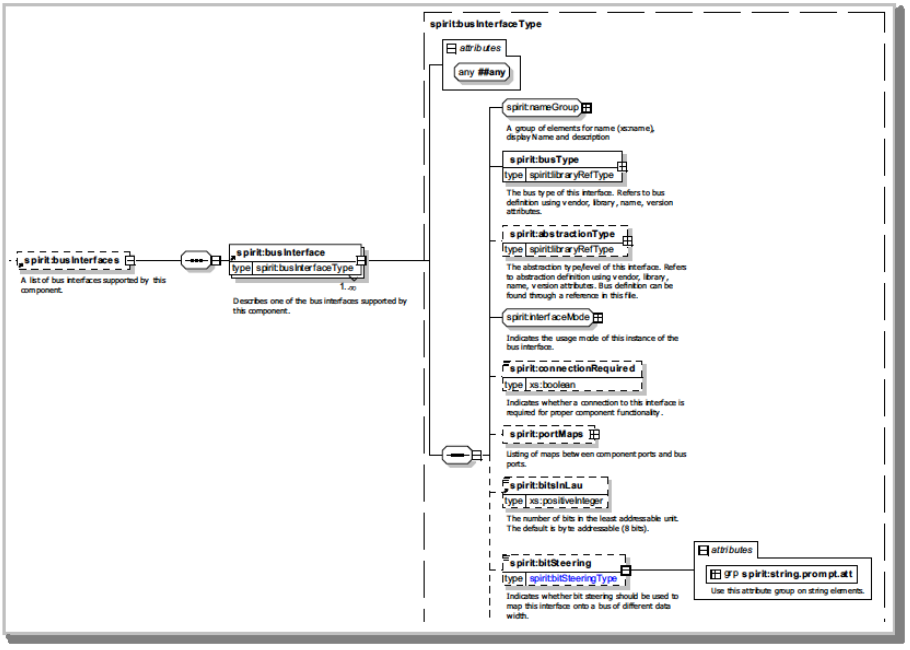


Figure 5.11: Elements of the bus interfaces element of a component description

As mentioned before, we have created ⟨*busDefinitions*⟩ and ⟨*abstractDefinitions*⟩ for several interfaces in Xilinx IPs. A typical IP contains interfaces not only for bus attachments (slave and master interconnections using bus standards such as PLB or AXI), but also for other communication and processing purposes, as depicted on Figure 5.7. For instance, the IP might be part of an image processing or DSP chain, in which data is received, treated in some way and then sent to the next component in the pipeline (ad-hoc interfaces); thus, interfaces for these purposes must be provided. Hierarchical interfaces can be used as external communication ports, for either internal signals in the FPGA or mapped to external pins.

As an example, Figure 5.12 depicts part of the ⟨*busInterface*⟩ of the System ACE memory controller, used for accessing data from an external non-volatile memory. The Figure 5.12 a) depicts a snapshot of the System_ACE ⟨*busInterface*⟩ element; since this interface does not correspond to a typical bus interface, we made use of the ⟨*system*⟩ group tag (in contrast to ⟨*onMaster*⟩ or ⟨*onSlave*⟩ for bus standard-based interfaces). As mentioned in previous sections, the IP-XACT standard permits the inclusion of the so-called ⟨*vendorExtensions*⟩, that usually correspond to a set of parameters (similar to tagged values in UML) that associated to an IP-XACT element, provide a means for storing additional, flow-specific metadata. In order to facilitate the model transformations in the FAMOUS framework, we have defined several extensions; the first are related to the ⟨*busInterface*⟩ element, in which extensions such as INTERFACE_TYPE have been de-

fined. The role of this and other extensions will be clearer in the model transformations section, but it suffices to say that depending on the interface type (bus, ad-hoc, and hierarchical interfaces) different parameters will be parsed during the back-end system generation phase.

The ⟨*ports*⟩ elements contained in the bus interfaces descriptions comprise external interface of the IP HDL entity, and in many cases, a core might encompass hundreds of them. IP descriptions with such amounts of ports data make it more difficult for CAD tools to process (and for the user to interconnect). Therefore, component-based methodologies exploit the abstraction capabilities provided by bundling together several ports into a single, more tractable element in order to facilitate the instantiation of the components into large systems. Then, during the netlist generation phase of an IP-XACT based methodology (design build), the ⟨*abstractDefinitions*⟩ associated with the components ⟨*busInterfaces*⟩ are parsed for retrieving the detailed ⟨*ports*⟩ information, as depicted on Figure 5.12 b). In the context of the FAMOUS methodology, this information is exploited by model transformation tools used to generate the top-level description used by Xilinx Platform Studio, the Microprocessor Hardware Specification file. More details of the packaging of ⟨*ports*⟩ will be provided when dealing with the model transformations.

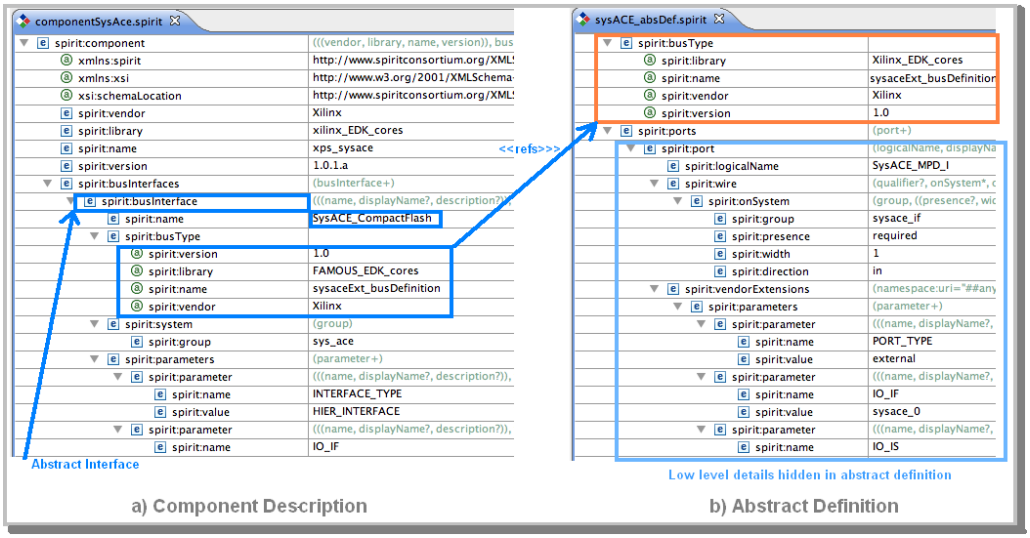


Figure 5.12: Relationship between a bus Interface in a component description and the associated abstract definition

The packaging (IP metadata reflection) of the bus interfaces and ports (internal and external) is obtained though a model transformation from MPD to an IP-XACT component. This model transformation will be detailed in a subsequent section.

#### 5.4.2.3/ MODELS AND VIEWS

The ⟨*model*⟩ element describes the ⟨*views*⟩, ⟨*ports*⟩ and model-related ⟨*parameters*⟩ of a component. An IP can contain different ⟨*views*⟩ such as RTL, TLM, software, and documentation, to name just a few. The ⟨*views*⟩ elements are used in tandem with ⟨*fileSets*⟩ and design flow information to enable the automation of component related tasks (such as FPGA synthesis, driver/source code compilation, etc). The ⟨*fileSets*⟩ and ⟨*views*⟩ elements of an IP-XACT component are closely related, since a given implementation of the component (⟨*view*⟩ in IP-XACT jargon) references to a specific implementation files bundle (a ⟨*fileSet*⟩), as depicted on Figure 5.13 b). This feature can be exploited for targeting different design flows, or for selecting specific features of a component at different levels of abstraction. When a set of components is imported into an IP-XACT compliant design environment, and used to compose a platform through design capture, a RTL or TLM view can be selected; this information is used by generators for accessing the associated metadata and performing a particular task (synthesis, netlist generation for RTL, the compilation of the IP associated software drivers, etc). The ⟨*view*⟩ element contains a set of ⟨*attributes*⟩ for unequivocally associating it to the particular design object (⟨*fileSet*⟩) it refers to; for instance, a ⟨*name*⟩ is used for categorizing the ⟨*view*⟩ , while the ⟨*envIdentifier*⟩ and ⟨*language*⟩ refer to the intended tool and implementation language, respectively. The ⟨*language*⟩ attribute can refer to VHDL and Verilog (for HDL artifacts) or C and H objects (for software files), among others. The ⟨*view*⟩ element can contain as well a set of associated ⟨*parameters*⟩ if additional, design flow information, is required.

In our framework, we make use of the ⟨*view*⟩ elements for different purposes. Each of these elements refers to a single or a set of artifacts described under the ⟨*fileSets*⟩ subelements of the component description. First, we use a ⟨*view*⟩ element for referencing to the nature and location of the IP HDL implementation files; this ⟨*view*⟩, labelled as *implementation*, references the HDL files used by the IP core. A second ⟨*view*⟩ refers to Xilinx XPS specific files, labelled as *xilinx_xps_data_files*, that are used by the tool for a variety of purposes, like retrieving TCL command files and other artifacts . Finally, a ⟨*view*⟩ called *edk_softwareDrivers* is used for referencing the associated drivers files in the form of C code.

In the particular RTL-based flow, the ⟨*model*⟩ element corresponds to the top-level VHDL (or architecture) description of the IP. An IP can reference to several implementations by using different ⟨*views*⟩ pointing to different implementation files (metadata stored under the ⟨*fileSet*⟩ elements, as will be described in the next subsection). However, this is only possible when the IPs contain the same external ports, which makes sense given that only the internal implementation details differ among the views. In our methodology, we exploit this capability of the ⟨*view*⟩ elements for describing components with different purposes but having the same interface. An example will be provided in the next section, when the ⟨*model*⟩ are described in more detail.

#### 5.4.2.4/ FILES AND FILESETS

The ⟨*fileSets*⟩ element contains a list of ⟨*files*⟩ and directories associated with a component, as depicted on Figure 5.13 b). These files might include drivers, implementation files (VHDL or Verilog), netlists, and other files related with a particular tool. The ⟨*fileSets*⟩ elements can be grouped for describing particular functions and purposes (using the ⟨*group*⟩ and ⟨*function*⟩ sub-elements), and this capability can greatly improve the EDA capabilities of an encompassing design flow. For instance, as mentioned in the previous section, ⟨*files*⟩ can be grouped to correspond to the intend of a particular ⟨*view*⟩. The ⟨*files*⟩ sub-elements contain as well a set of attributes for automating the tasks associated with the related ⟨*view*⟩. For instance, the location of the implementation file needs to be indicated to the point tools that access the metadata information during the build phase, along build commands(this information is stored under the ⟨*dependency*⟩ and ⟨*buildCommand*⟩ sub-elements, respectively).

As described above, we have separated the Xilinx XPS IP cores ⟨*files*⟩ in three groups: *implementation*, *xilinx_xps_data_files*, and *edk_softwareDrivers*. The rationale behind this choice is due to the way the Xilinx XPS IPs that we are packaging follow all a well-defined directory structure, depicted on Figure 5.13 a). This IP_cores directory structure enables Xilinx tool to automate the access to the IP repositories in an structured manner; even if Xilinx XPS is not to be used, the generators that may eventually access the IP-XACT component implementation information for the design build phase need to know the exact location of these files, along ⟨*buildCommands*⟩ metadata for executing the associated procedures (e.g. synthesis or compilation, for VHDL or drivers, respectively). It must be noted that this repository organization only applies to Xilinx XPS cores; in the case of the PRM and non-XPS IPs, which are not stored in the Xilinx XPS installation repositories,

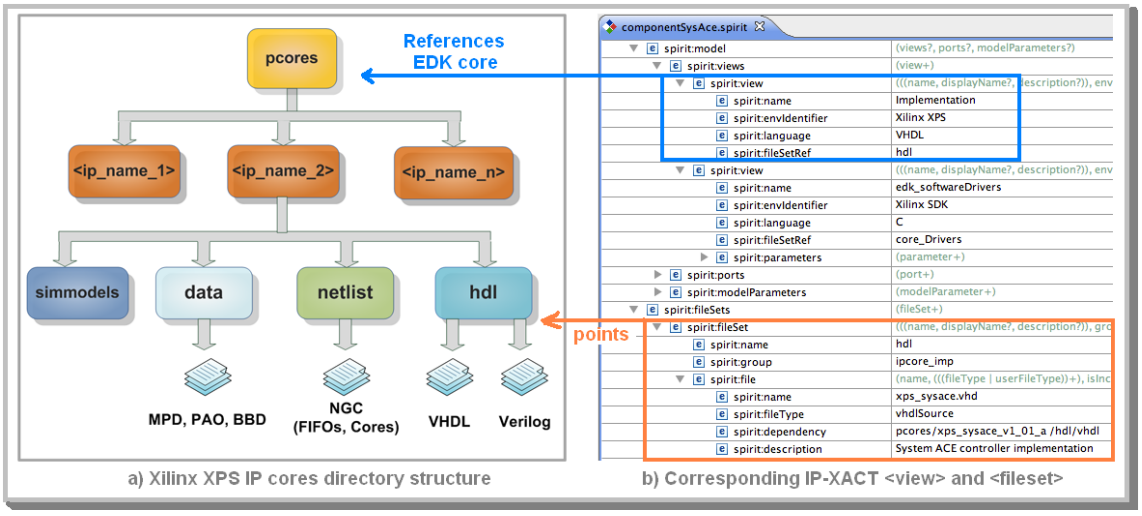


Figure 5.13: Relationship between a model, its views and the IP component implementation artifacts

the ⟨*fileSets*⟩ only reference the ⟨*dependency*⟩ path, along the required ⟨*buildCommands* ⟩ information.

The packaging of the implementation metadata is done in two manners, depending on the type of IP. For Xilinx XPS IPs, the ⟨*fileSets*⟩ sub-elements is created from parsing the Peripheral Access Order (PAO) file, which as described previously, contains the information of the HDL/Verilog files used by an IP, and the required order of synthesis. For non-XPS IPs, this information has been performed manually, but commercial tools could be exploited for this purpose. The role of the ⟨*fileSets*⟩ elements of the IP-XACT component description will be described in depth later on, when addressing the system composition and generation of the SoC Platform.

#### 5.4.2.5/ Parameters and Choices

The last elements in the component description needed for an IP reuse and design by reuse methodology are the parameters and choices. The ⟨*parameters*⟩ and ⟨*choices*⟩ sub-elements in a component description permit the configuration of the associated IP core, and enable to automate the parameterization of an IP within a tool flow, along some tangible results, such the generation of a synthesized netlist (i.e. in an RTL flow). As with any other design by reuse methodologies, parameters are normally set at the platform level, using an IP-XACT object know as ⟨*design*⟩, where a set of component instances encompassing the configuration and interconnection information of the underlying IP core are assembled in a top-level description model. We have already discussed ⟨*busInterfaces*⟩ and ⟨*ports*⟩ store the metadata required for assembling the IPs; in this subsection, we address the parameterization information of the IP, which is stored in one of the two elements introduced above.

The sub-schema of the ⟨*parameters*⟩ element in an IP-XACT component description is depicted on Figure 5.14. The ⟨*parameter*⟩ sub-element contains itself a set of elements used to characterize it; for instance, a ⟨*dataType*⟩ element defines the type of values the parameter can hold (e.g., integer, float, or string). Secondly, parameters can be bundled together into well-defined groups using the ⟨*nameGroupString*⟩, for design automation purposes (parameters might be treated in a different manner during the design configuration phase, depending upon the configuration phase). We make use of the capability to classify ⟨*parameters*⟩, information that is used during the model transformation phase, and which will be introduced in a subsequent section.

Any component contains a set of ⟨*parameters*⟩ (i.e. generics in VHDL), some of them being static (meaning that they cannot be configured, but they are propagated in the IP description) and other configurable. A configurable ⟨*parameters*⟩ implies that the ⟨*value*⟩ of the element can be set differently for each use of the IP description, which allows a single description to be used in many different ways. The value of a config-

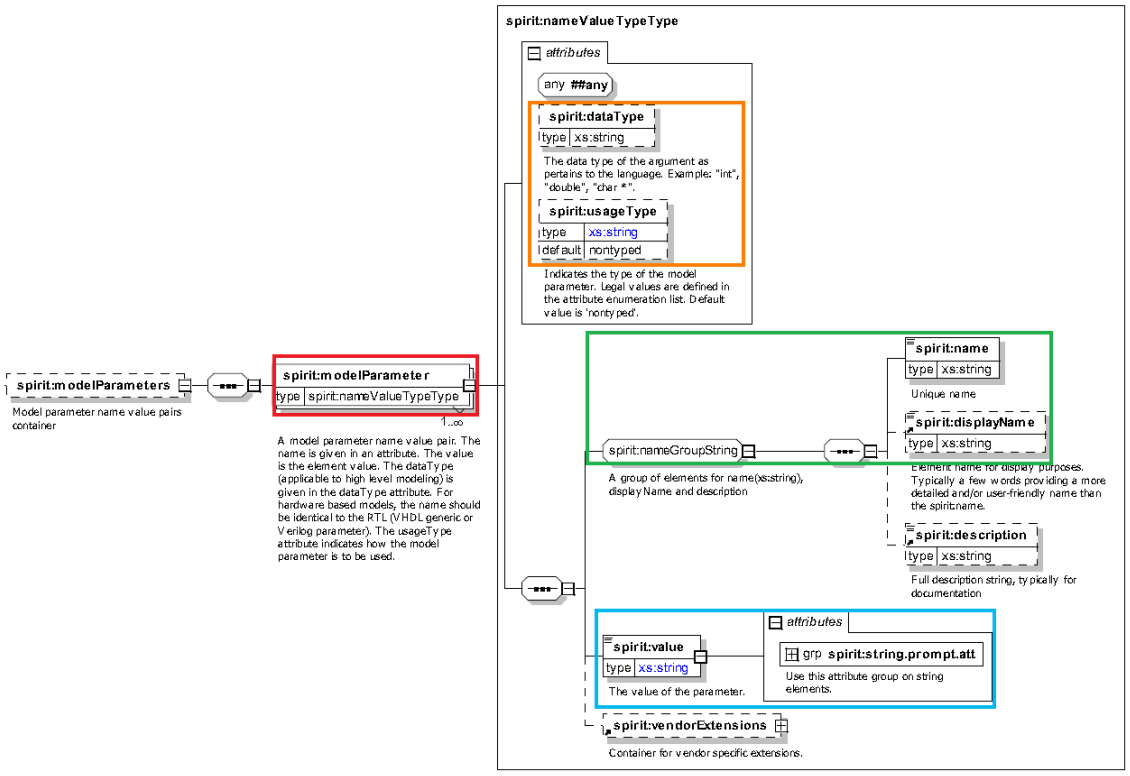


Figure 5.14: IP-XACT sub-schema for the parameters in a component description

urable ⟨*parameter*⟩ is customizable in IP-XACT by attaching a set of attributes, contained the so-called ⟨*attributeGroup*⟩ and that permits the CAD tool designer to specify how a ⟨*parameter*⟩ is to be set within a tool flow. This classification capabilities provide a great degree of flexibility, an important aspect in EDA, by defining a large set of attributes within a group; there are four kinds of attribute groups in IP-XACT, depending on the ⟨*dataType*⟩ hold by the ⟨*parameter*⟩: boolean, float, long and string. In Figure 5.15, we depict the boolean attribute group, along the attributes it contains. It must be noted however that the other groups contain similar attributes, changing only the associated ⟨*dataType*⟩. As with many other elements in the IP-XACT standard, the attributes described here are optional, and we have made use of those highlighted in the figure, for reasons that will be explained as follows.

1. ⟨*format*⟩: The first attribute in the groups is the ⟨*format*⟩ of the value to be held by the parameter. The type of value is defined upon the used attribute group: boolean, float, long and string being the four classes of formats

2. ⟨*id*⟩: When a component is instantiated onto a ⟨*design*⟩ description using an IP-XACT compliant design environment, a set of the ⟨*parameters*⟩ in the IP-XACT component description is available for the user to set. The configuration of the so-called ⟨*configurableElements*⟩ is done via a reference to an ⟨*ID*⟩ attribute contained in the element, which belongs to the ⟨*attributeGroup*⟩ sub-element (and which is stored into an

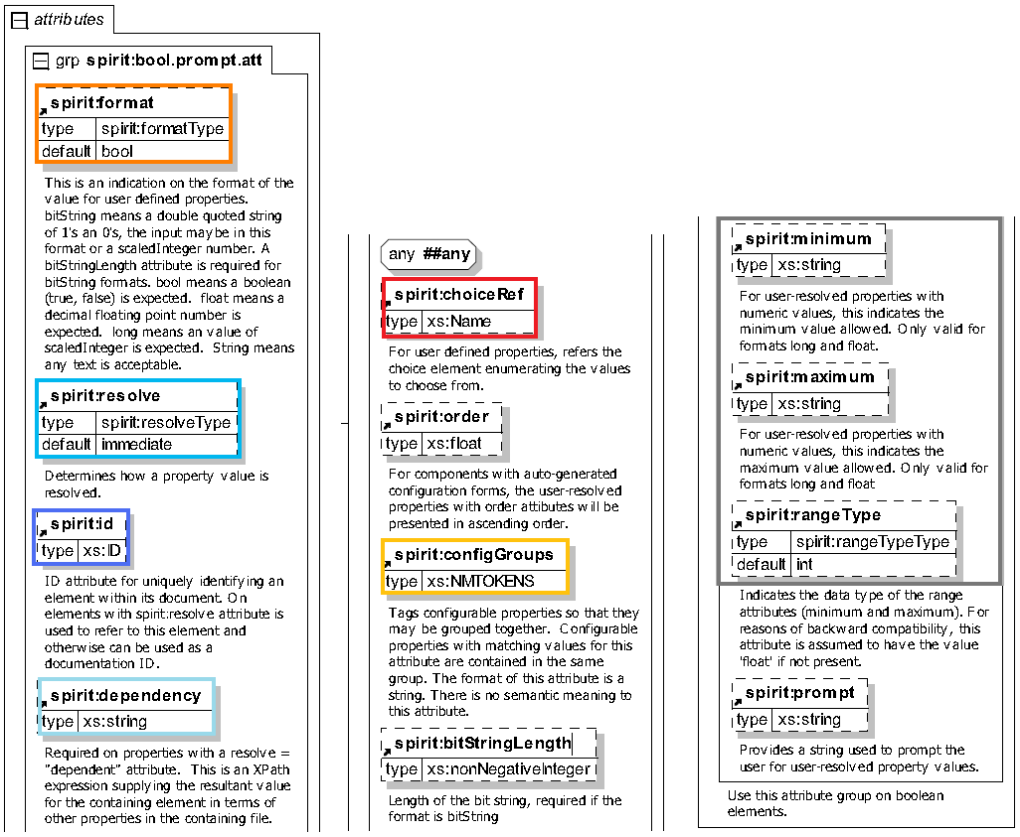


Figure 5.15: IP-XACT attribute group for boolean parameters

string). However, not all the ⟨*parameters*⟩ are directly configurable; many ⟨*parameters*⟩ are static, and not directly accessible to the user. On the other hand, the value of the configurable ⟨*parameters*⟩ can be set in different manners.

3. ⟨*configGroups*⟩: The ⟨*parameters*⟩ in the component description can be classified into different groups for facilitating their processing by the generators or configuration tools. For instance, a set of ⟨*parameters*⟩ can be associated to a ⟨*busInterface*⟩; if this interface is not used in the final implementation (due to customization process), then the ⟨*parameters*⟩ under that ⟨*configGroup*⟩ are not used for configuring the component.

4. ⟨*minimum*⟩, ⟨*maximum*⟩ attributes are used for delimiting the minimum and maximum allowed values for the associated parameter. Similarly, if the other attributes are not used, the ⟨*rangeType*⟩ defines maximum allowed range of values for the parameter, and its type (e.g. integer, float, etc.)

5. ⟨*resolve*⟩: This attribute defines how the value for the containing element is to be configured. The value shall be one from the enumerated list of immediate, user, dependent, or generated. This attribute is probably the most important for the EDA intentions of the IP-XACT standard, since specifies how the parameters it to be used within a tool flow; basically, the resolve element tells the generators (or parser in the case of our approach) how to treat a parameter, as detailed as follows:

- The ⟨*immediate*⟩ configuration (the default) in the ⟨*resolve*⟩ attribute indicates the value shall be specified in the containing element. This implies that these elements cannot be changed, and that the default values will immediately resolved (used as is) for any generation purposes. Static ⟨*parameters*⟩ are typically categorized using this attribute.

- The ⟨*user*⟩ configuration indicates the value shall be specified by user input and the new value stored in a ⟨*componentInstance*⟩ (at the platform level or ⟨*design*⟩ under the ⟨*configurableElement*⟩ element referenced with the associated ⟨*ID*⟩

- The ⟨*dependent*⟩ configuration indicates the parameter value shall be defined by an XPATH equation defined in the ⟨*dependency*⟩ attribute. The values assigned to the parameters are inferred (resolved) from the result of the equation.

- ⟨*generated*⟩ configuration indicates the value shall be set by a generator (written automatically by CAD tools) and the new value stored under the ⟨*configuranleElement*⟩ sub-element of a component instance. Examples of these parameters are the base and high addresses of the IP (the memory range assigned for the registers in an IP), that are only assigned when the total memory map has been computed by the SoC design CAD tools.

- The ⟨*dependency*⟩ attribute requires the ⟨*resolve*⟩ attribute to be equal to dependent. It is typically used for describing ⟨*parameters*⟩ that depend on other, usually top-level parameters (i.e. generics), but it can also be modified by user defined configuration values of other ⟨*parameters*⟩ in the component description.This equation takes the other parameters ⟨*IDs*⟩ as operands, and the resolved value is assigned to the corresponding parameter value.

6. The ⟨*choiceRef*⟩ attribute, though separated from the resolve element in the standard, is very much related to the ⟨*dependent*⟩ configuration, since the value of a ⟨*parameter*⟩ might depend on the selection of a value from an enumeration list (defined by the ⟨*choices*⟩ element of the component description, which has been tied to the ⟨*parameters*⟩ in this discussion for convenience reasons). For instance, a parameter can take a value from a predefined set of possibilities (i.e. the baud rate values for an UART component); then, the ID is tied to not to a ⟨*parameter*⟩ but to a ⟨*choice*⟩, and the values is taken from the available values described in that enumeration. Examples will be provided in a subsequent section, when dealing with the transformation rules between the MPD file and IP-XACT component description.

The packaging of the parameters from the MPD file has been done in such a way the each of them fits in one of the aforementioned categories. In some cases, there is an overlap between the two IP component representations, and when not, some vendor extensions have been introduced in some elements of the schema.

### 5.4.3/ UML MARTE Proposed IP Deployment Package

In order to promote IP reuse in our approach, we have introduced the Deploy package as an extension to the MARTE Profile at the Deployed Allocation level, in which we use two kinds of UML diagrams. First, we have to identify which elements can be deployed in the logical view (behaviour or structure), and what can serve as a target of a deployment, the physical view (a resource or a service). The stereotype ⟨⟨*deployed*⟩⟩ (which belongs to the extended Deploy package in MARTE) is used for this matter in the Deployed Component diagram (or Platform View). Each elementary component corresponds to an IP implementation in the IP-XACT library. In order to enable this, we have defined a new stereotype, labelled as ⟨⟨*IP*⟩⟩, which makes use of several resources from the Generic Resource Modeling (GRM) package in MARTE.

This stereotype is applied on classes in the Diagram Class of IPs (or Parameterization View), which is the second kind of UML diagram used at this level, and which contains a set of components instances used for configuring top-level parameters, which control the creation of the MHS file from the IP-XACT design. The ⟨⟨*IP*⟩⟩ stereotype contains a set of attributes used to describe basic information of the IP at high-levels of abstraction, as depicted on Figure 5.16. We have decided to keep these attributes to a minimum, since the designer does not need to know all the parameters of the IP. We have defined two ⟨⟨*DataType*⟩⟩ to provide a means to deploy the IP.

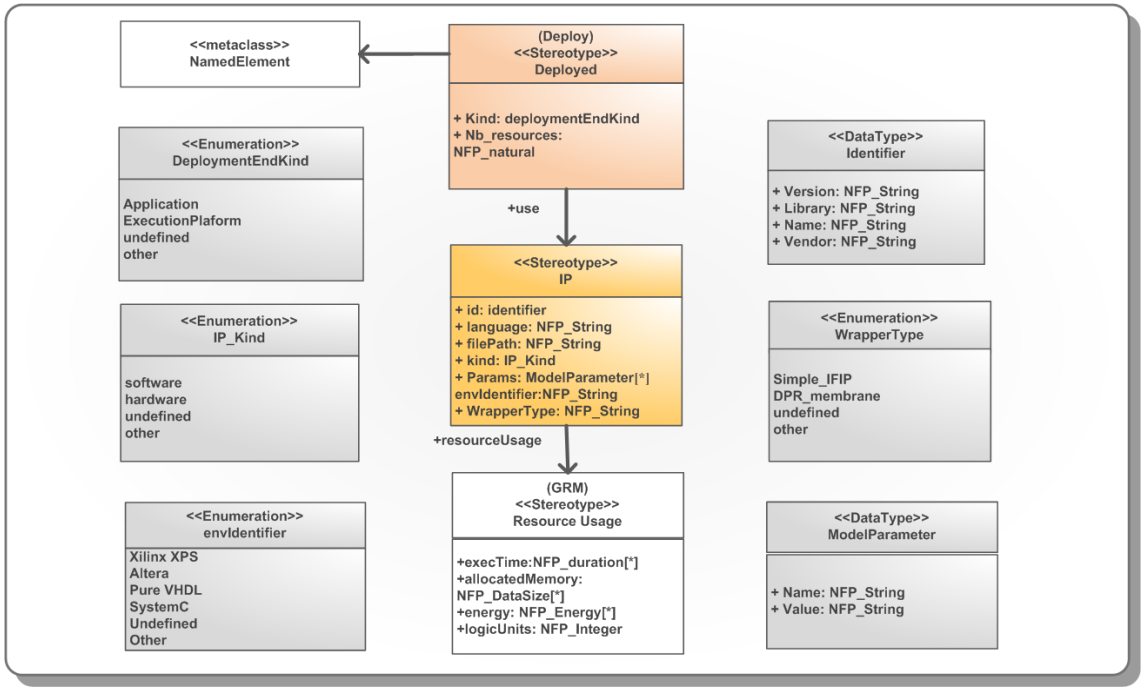


Figure 5.16: IP-XACT component block representation showing the main concepts for IP parameterization and configuration

These data types are explained in more detail as follows. The ⟨⟨*IP_Kind*⟩⟩ enumeration is

used to identify the type of implementation of the IP core. This provides a mechanism for identifying which parameters should be used in the flow, since the kind of implementation determines their configuration. In this work, we assume that all the IPs are implemented as HW components (hence, the IP language attribute should be VHDL or Verilog). However, this can be extended for software implementation functionalities. The flow specific information is obtained from the IP-XACT component description, particularly from the ⟨*View*⟩ element), by assigning a value from the ⟨⟨*envIdentifier*⟩⟩ enumeration (e.g. Xilinx XPS, Pure VHDL, SystemC); in this way, we provide an easy way to select which back-end will be used in the system generation phase.

The ID element types ⟨⟨*identifier*⟩⟩ which is a ⟨⟨*DataType*⟩⟩ in MARTE and contains a set of attributes to link the high-level descriptions to their IP-XACT counterparts. This type provides a means to unequivocally identify a component in the library by associating a ⟨*vlnv*⟩ (Version, Library, Name, and Version) tuple to a component instance in UML MARTE. For Xilinx EDK IPs, the Name and Version values are obtained from the MPD file (Name and HW_Ver). Nonetheless, we have decided to keep the ⟨⟨*filePath*⟩⟩ attribute for cases in which the designer wants to point the location of the implementation files; this information can be retrieved from ⟨*fileSet*⟩ element in the IP-XACT component description, under the ⟨*Dependency*⟩ element.

Finally, the most important aspect of our approach for parameterization (and eventually customization) of a component instance in the hardware platform, is the capability to import the most relevant parameters of the IP-XACT component into the UML MARTE component template. We perform model transformations from the Xilinx MPD files to obtain our IP-XACT library; the components in the library contain a set of elements described in the standard. The ⟨*Model*⟩ element contains, among other features, a description of the ⟨*Parameters*⟩ of the IP, typically implementation dependent information. We store the parameters information from the MPD file into the ⟨*view:Parameters*⟩ section of the component description; we have extended the definition of parameters with Xilinx specific attributes, through vendor extensions. These extensions will be described in more detail the next section.

In order to illustrate the concepts discussed above, Figure 5.17 a) illustrates a snapshot of a deployed component view diagram in which we make use of stereotypes from MARTE HRM package (e.g. HwComponent) in order to describe the logical architecture. We also use ⟨⟨*deployed*⟩⟩ stereotype from Deploy package to match each component to its respective IP defined in a class diagram. This is the so-called Platform View; using a CSD, the designer is interested in describing the way the system is to be connected, not concerned to the low level aspects of the design. The MARTE extensions discussed in the previous section allow us to import the IP description to generate the views used for parameterization and integration, as depicted in Figure b). We promote IP reuse in our approach by importing important parameters into a ⟨⟨*IP*⟩⟩ instance in MARTE; the creation of both views is done automatically by models transformations.

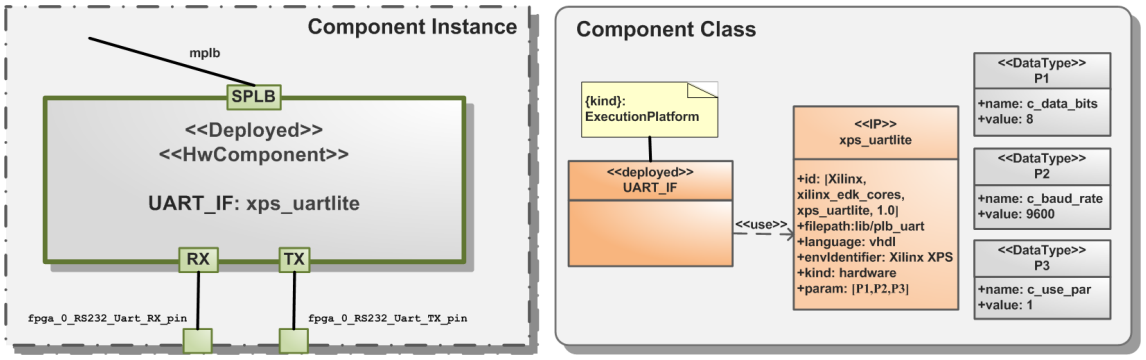


Figure 5.17: a) Snapshot of the MARTE architecture view. b) Example of an IP instantiation with parameters

In order to accomplish this, we need first to define a transformation from MPD to an IP-XACT component description; the transformations are defined in the next section, but the basic principle is to categorize important features in the MPD model into meaningful groups in order to obtain only the required information for the high level models. For instance, parameters can be categorized as ⟨*visible*⟩, ⟨*visible_when_valid*⟩,⟨*optional*⟩ , and ⟨*constant*⟩; these attributes are defined using the ⟨*configGroups*⟩ tag under the ⟨*view:Parameters*⟩ elements.

By separating the parameters into different groups, we can define which sets can be imported into the MARTE component description; only ⟨*visible*⟩ and ⟨*visible_when_valid*⟩ parameters are imported into MARTE component template by an IP-XACT to MARTE transformation. The reason for this is that ⟨*optional*⟩ and ⟨*constant*⟩ parameters are typically part of the IP description, but not used in the parameterization phase. Another aspect in the reuse of IPs is the customization of different components of the implementation; the VHDL underlying components can be designed in such a way that code templates can be added or removed from in the synthesis phase by controlling parameters in the MPD description. Therefore, these parameters control the inclusion of other parameters, ports and bus interfaces into the final IP description; the ⟨*visible_when_valid*⟩ tag is thus applied to all these three groups when converting the MPD description into an IP-XACT description.

When performing the conversion into MARTE models, the parser looks for the ⟨*visible*⟩ and ⟨*visible_when_valid*⟩ tags under the ⟨*parameters*⟩, ⟨*ports*⟩ and ⟨*busInterfaces*⟩ sections of the IP-XACT component description, and creates a MARTE template accordingly. However, dependencies on other parameters are not supported by the current specification of IP-XACT (v1.5.); the standard specifies ways to control the values of certain parameters and choices dependent on equations involving other parameters, but no methods to control the inclusion or not of certain elements in the final generation phase. In order to support this, we have defined ⟨*vendorExtensions*⟩ in the aforementioned elements, which are parsed to resolve this attribute. The generation of the structural information of

the components is more straightforward: bus interfaces and ports are converted to UML ports and named after the IP-XACT description; similarly as in the case of parameters, only those with the visible and visible when valid tags are used to generate the MARTE component. The components labelled as ⟨⟨*deployed*⟩⟩ in the deployed architecture diagram, are linked to the ⟨⟨*IP*⟩⟩ stereotypes in the class diagram of IPs. Each deployed component corresponds to an IP class and stored in the MML MARTE Library as a template that can be subsequently used as the building block of the Platform View, as shown in Figure b). As mentioned before, both views are parsed and used to create an IP-XACT design description exploited for system generation.

## 5.5/ Proposed Model Transformations for creating he multi-level IP library

In this section will be further describe how the mapping between the Xilinx EDK files and their IP-XACT counterparts is performed. In the previous section we described how the MPD file is composed and the elements of the IP it describes. As mentioned previously, the MPD file contains all the parameters, ports and bus interfaces of an IP; however, the MHS file has precedence over the MPD file. This means that the values set at the top-level change which elements of the MPD/IP core will be implemented.

### 5.5.1/ MPD → IP-XACT component transformation rules.

We start with the BUS INTERFACES and IO INTERFACES of the IP. This concept exists in IP-XACT in the ⟨*busInterfaces*⟩ element; however, the nested PARAMETERS of the MPD file are not part of the standard. For this reason, we have decided to store these nested parameters as ⟨*vendorExtensions*⟩ in each of the ⟨*busInterfaces*⟩ elements of a component description, as shown in Figure 5.18. We have defined an ⟨*attribute*⟩, INTERFACE_TYPE, to differentiate normal BUS INTERFACES from IO INTERFACES in the IP-XACT description. Depending on the value of these parameters, the subsequent elements in the interfaces descriptions are used to create the MPD description, if this is the objective of the transformation phase. An important PARAMETER for the bus interfaces description is the IS_VALID option, which controls if the interface is included or not in the design.

The second element of the MPD file to be mapped is the PARAMETERS, which are used for different purposes. IP-XACT provides the possibility to store parameters in different elements of the component description. Since we are targeting Xilinx EDK cores, we have decided to store these parameters in the ⟨*parameters*⟩ section of a Xilinx XPS ⟨*view*⟩ element; in this way, the rest of the IP description can be more generic and not tool dependent. Figure 5.18 shows how the PARAMETERS have been mapped to IP-XACT. In

| MPD Command | IP-XACT Component Counterpart |
|---|---|
| *a) BUS_INTERFACE* | spirit:busInterface<br>if (interface_type) = *bus_interface*<br>spirit:parameters:spirit: |
| INTERFACE_TYPE | parameter:Interface_Type |
| BUS | parameter:bus:value |
| BUS_STD | parameter:bus_std:value |
| BUS_TYPE | parameter:bus_type:value |
| GENERATE_BURSTS | parameter:generate_bursts:value |
| ISVALID | parameter: value =<br>((Depedency(param_id)) ? 0:1) |
| SHARES_ADDR | parameter:share_addr:value |

| *b) IO_INTERFACE* | spirit:busInterface<br>if (interface_type) = *hier_interface*<br>spirit:parameters:spirit: |
|---|---|
| INTERFACE_TYPE | parameter:Interface_Type |
| *IO_IF* | parameter:IO_IF |
| *IO_TYPE* | parameter:IO_TYPE |

| c) PARAMETERS | *model:modelParameters*<br>modelParameter:spirit: |
|---|---|
| PARAM_i = VALUE_i | modelParameter : value |
| ASSIGNMENT | (constant, optional, require, update) |
| DT | dataType |
| ISVALID | vendorExtensions:parameters:<br>isValid:((Depedency(param_id)) ? 0:1) |
| MIN_SIZE | value:minimum |
| PERMIT | value:configGroups<br>isValid:((Depedency(param_id)) ? 0:1)<br>and resolve = "user" |
| RANGE | value:minimum - value:maximum |
| VALUES | if value:spirit:format = "choice"<br>then VALUES = (enum1,...,enumn) |

| MPDt Command | MPD Asignment Command |
|---|---|
| d) Ports | *model:Ports*<br>spirit:port:spirit |
| NAME | name |
| DIR | wire:spirit:direction |
| VEC | vector:left - vector:right *or*<br>if resove = dep right((Depedency(param_id)) |
| *e) PARAMS* | port:vendorExtensions:Parameters |
| PORT_TYPE | parameter:portType<br>(bus_signal, hier_signal, ah-hoc_signal) |
| PORT_GROUP | parameter:portGroup |
| SIGNAL_TYPE | parameter:signalType<br>(clk, interrupt, bus, io, three_st) |
| ISVALID | parameter:isValid<br>((Depedency(param_id)) ? 0:1) |
| ASSIGNMENT | (constant, optional, require, update) |
| PERMIT | if (port_type = "hier_signal" andpermit = user)<br>PERMIT = base_user<br>otherwise PERMIT = advanced_user |
| BUFFER_TYPE | parameter:bufferType |
| BUS | if (portType = "bus_signal")<br>parameter:bus_name |
| IO_IF | if (portType = "hier_signal" or "ad-hoc_signal")<br>parameter:bundle_name |
| IO_IS | if (portType = "hier_signal")<br>parameter:bundle_name |
| INT_PRIOR | if (signalType = "interrupt")<br>parameter:interruptPriority |
| SENSITIVITY | if (signalType = "interrupt")<br>parameter:sensitivity |
| SIGIS | parameter: SIGIS<br>(clk, rst, interrupt) |
| THREE_ST | if (signalType = "three_st")<br>parameter:threeState |
| TRI_I | if (signalType = "three_st")<br>parameter:tri_i |
| TRI_O | if (signalType = "three_st")<br>parameter:tri_o |
| TRI_T | if (signalType = "three_st")<br>parameter:tri_t |

Figure 5.18: MPD to IP-XACT mappings for different elements

some cases new ⟨*parameters*⟩ have been created, whereas in others cases the mapping can be performed through already existing concepts.

For the PORTS section of the MPD description, we have used the ports description in the ⟨*model*⟩ element of the IP-XACT component description. As with the ⟨*parameters*⟩, setting the configuration of the ⟨*ports*⟩ in this section, helps in keeping the component description independent of the intended flow. For a VHDL implementation, only the name, direction and size of the port are required; this information is used for generating the entity to be connected in the top-level description.

However, ports in a SoC flow are more complex. For instance, in Xilinx EDK, each of the ports in the MPD description has a set of nested PARAMETERS or attributes. These attributes depend on the port type we are dealing with; we have defined three parameters to identify the type of port: PORT_TYPE (bus signal, external, internal), PORT_GROUP (to associate a port with a bus interface or IO interface), and SIGNAL_TYPE (clk, interrupt, bus, io, three_st). This has been necessary since IP-XACT provides very limited port classification capabilities. Depending on the values of these signals in the MPD file or in the design entry phase, the rest of the PARAMETERS in the port description will be created or parsed, depending on the scenario. The figure also shows these PARAME-

TERS and how certain among them depend on the values of the first three signal types, which are not part of the MPD description.

Finally, the options in the MPD file deal with technological/flow related aspects. We have decided to store these values in the normal ⟨*parameters*⟩ of the component description, since they can be parsed first and affect, in some cases, the ⟨*parameters*⟩ located in other sections of the component description. Examples of these parameters/OPTIONS are Arch_Support (the FPGAs in which certain IPs can be used), CORE_STATE (active, deprecated, development, obsolete), HDL (BOTH, MIXED, VERILOG, VHDL), IMP_NETLIST, IP_GROUP (this parameters can be used as well in the IP-XACT description to improve the way components are searched), IP_TYPE.

### 5.5.2/ IP-XACT → MARTE COMPONENT TRANSFORMATION RULES

Once the IP-XACT component library has been populated, the following step is to perform the creation of the UML MARTE templates library, which have been depicted on Figure 5.17. This IP models conform to the IP Deployment metamodel introduced in the previous section, and the basic idea is to provide the high-level user in MARTE with a sub-set of abstract elements in the IP-XACT component descriptions; in contrast with previous approaches, that have made use of metamodels comprising the totality of the IP-XACT component sub-elements, we have favoured the use of a simpler IP metamodel that allow us to import just the most basic information for the user at high-levels of abstraction. The mappings between the IP-XACT component metamodel and the IP Deployment Metamodel are depicted on Figure 5.19; as described before, the IP deployment template contains an attribute to set the corresponding VLNV value, and thus, this information is directly mapped. The second element to be mapped to the IP template are a list of high-level parameters; these parameters are only applicable for configurable soft IPs, and have been labelled in the IP-XACT description so they belong to the visible ⟨*ConfigGroups*⟩ (top-level configuration parameters) and which need to be set by the defined as ⟨*user*⟩ under the ⟨*resolve*⟩ attribute of the ⟨*parameter*⟩ element, so that a correct kernel of the IP is instantiated, and the corresponding high-level parameters are propagated in the generation phase, serving as the controlling parameters of the introspective architecture.

Then, the ad-Hoc ⟨*ports*⟩ (those not belonging to predefined bus interfaces) are associated to the IP template. These ports need to be connected in the Platform View of the UML MARTE models (using a PortKind = adHoc_Port_Int or adHoc_Port_Ext tag for differentiating between adHoc ports and hierarchical ports ). The bus interfaces are much simpler to map, since they only refer to a PortKind = busIF that require a name (i.e. SPLB for slave interfaces) and a bus type (PLB, FSL, AXI). By providing these simple mechanisms, the IP templates (or kernels) can be instantiated onto the UML MARTE models, and then parameterized and interconnected for obtaining an abstract representation of the hardware component of a SoC platform. This description contains the minimum-required

information necessary for generating an IP-XACT design object, which is used by MD-Workbench for the model transformations in order to obtained enriched models such as the Microprocessor Hardware Specification (MHS) or the top-level VHDL description. This process will be the subject of the next section.

## 5.6/ DESIGN BY REUSE IN THE FAMOUS METHODOLOGY

In this section, we deal with the second part of a design by reuse methodology: the composition of a SoC platform by reusing IP components from the multi-level library. Furthermore, the system description composed in such a way must be used for performing IP parameterization and the generation of the back-end platform model through metadata introspection. This process, based on the MCF paradigm, is depicted once again on Figure 5.20; we will focus our attention on the dashed square of the figure, which implies how the modeling of the platform in performed in UML MARTE, and how this description in converted into our chosen XML intermediate representation (an IP-XACT design description object). This IP-XACT description is used with two purposes: parsing the IP component descriptions in the IP library for obtaining relevant configuration and interconnection information and, once this information is available, for generating the back-end platform description (e.g. RTL or Xilinx Platform Studio MHS file). Of course, the Partial Reconfiguration Support must be taken into account as well during the modeling phase, but being this a component-based approach, this requirement does not impact the way the platform is created, but the generation phase.

Consequently, we engage in a thoroughly discussion of the platform modeling in UML MARTE and how these concepts are related to their IP-XACT counterparts in the design object description. Some of the UML MARTE extensions are introduced to provide the link between these two levels, as well as the required model transformations to perform their mapping. However, before engaging in a discussion of all the high-level modeling aspects, we first analyze how the Xilinx Platform Studio back-end platform models are

| IP - XACT | UML MARTE |
|---|---|
| Component | *Class Instance* |
| spirit: version: library: name: vendor | id: identifier<br>*(VLNV tuple)* |
| spirit:busInterfaces:busInterface:name | Port-BusIF = name<br>port kind = busIF |
| spirit:Model:ports:port:name | Port-adHoc = name<br>port kind = ad-hoc port |
| (if <user> and <configGroups> = visible)<br>spirit:view:Parameters | Parameter_n = name |

Figure 5.19: IP-XACT to MARTE templates mappings

used in the context of the flow, and from there, we outline the requirements in terms of the modeling of the encompassing objects at high-levels of abstraction. In this manner, we introduce the proposed transformation rules to move from MARTE to Xilinx XPS, passing through the IP-XACT introspective architecture. We provide several examples of how these concepts have been developed, and some case studies are provided in the next chapter.

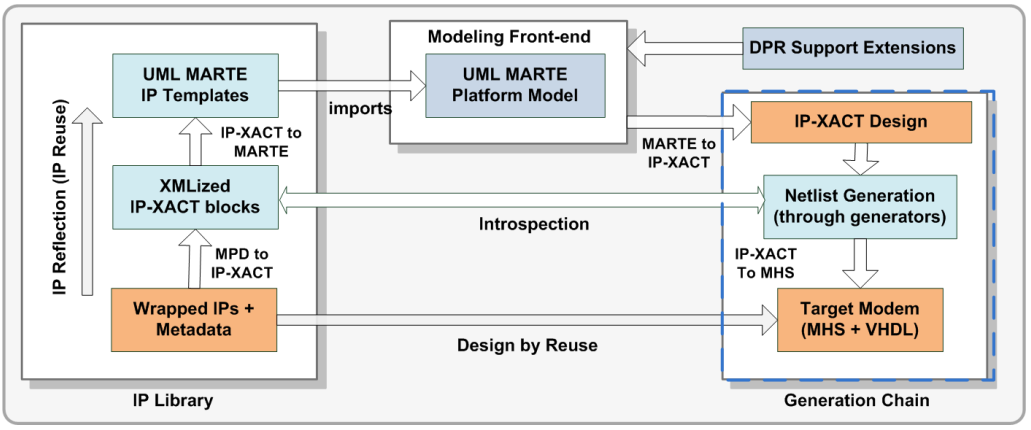


Figure 5.20: System generation phase of the MCF FAMOUS methodology

### 5.6.1/ Xilinx Platform Studio Back-end System Generation

In Chapter 4 we have introduced the Xilinx Platform Studio tool, and its use in the conception of complex FPGA-based SoC systems. The SoC hardware platform consists of one or more processors and peripherals connected to the processor buses. The description of these components is expressed via a top-level description, usually in VHDL, which consists in a set of component instances (with the associated generic and ports), and the necessary interconnections to define the desired functionality of the platform. As described in Chapter 2, the user of VHDL does not lend itself very well for IP reuse and design by reuse: VHDL is difficult to parse and usually does not contain design flow information, since it is intended as a flow agnostic HDL, which can be targeted to different back-ends (i.e. synthesis for generating a netlist for a particular FPGA technology). Therefore, metadata driven approaches (making use of intermediate IP and platform representations) have been put into practice by CAD vendors for automating the retrieval, instantiation, parameterization and interconnection of IP (the IP design reuse cycle); as with the IP-XACT standard, the rationale behind these representations is to ease the task of the designer by automating many of the burdensome steps, with the aid of Graphical User Interfaces (GUIs), which make use of the metadata contained in the intermediate representations.

Xilinx Platform Studio deploys the Microprocessor Hardware Specification (MHS) file for capturing the hardware platform description. This intermediate platform description is

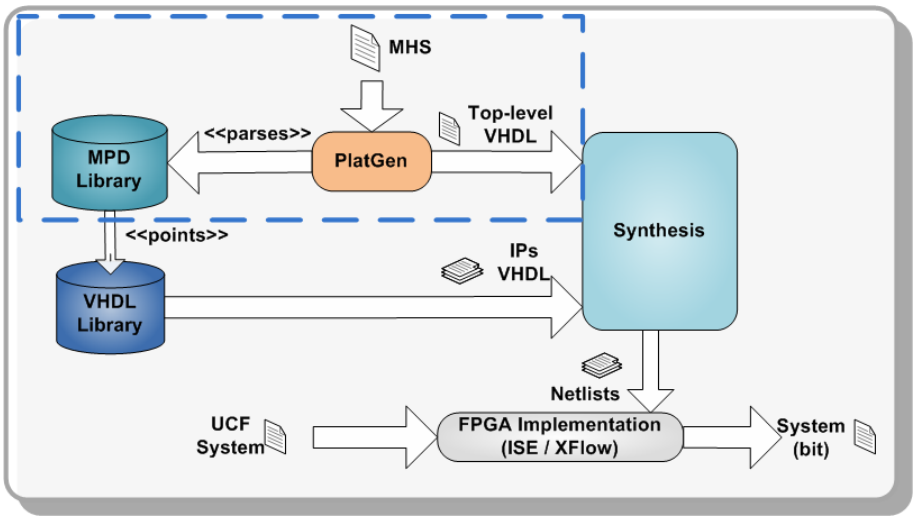


Figure 5.21: Role of the MHS file in the Xilinx Platform Studio Back-end

written in an ASCII textual format, with an associated syntax, which helps XPS tools to parse the associated files to mine data corresponding to several elements in underlying IP and platform implementations. Xilinx Platform Studio provides a design environment in which a platform can be created through the use of GUIs that help the designer in accessing the IP metadata in an efficient manner, without actually having to deal with the underlying IP VHDL/Verilog code. The role of the MHS file in the Xilinx Platform Studio design flow is depicted again on Figure 5.21, but here we focus more in the system composition aspects of the design cycle.

As explained in the previous section, the Platgen tool is at the heart of the generation phase of Xilinx Platform Studio. Platgen compiles the intermediate MHS description and those corresponding to the instantiated modules in a processor-based system, and translates them into HDL netlists that can be implemented in a target FPGA device. Some of the tasks performed by Platgen are described as follows.

1. Reads the MHS file as its primary design input.
2. Reads various IP blocks (pcore) hardware description files (MPD, PAO) from the XPS project and any user IP repository.
3. Produces the top-level HDL design file for the embedded system that stitches together all the instances of parameterized pcores contained in the system. In the process, it resolves the high-level bus connections in the MHS into the actual signals required to interconnect the processors, peripherals and on-chip memories. (The system-level HDL netlist produced by Platgen is used as part of the FPGA implementation process.). This process is shown on Figure 5.22, where an instance on the MHS is compared with the underlying component instantiation in the top-level VHDL description
4. The, Platgen invokes the XST (Xilinx Synthesis Technology) compiler to synthesize each of the instantiated components.

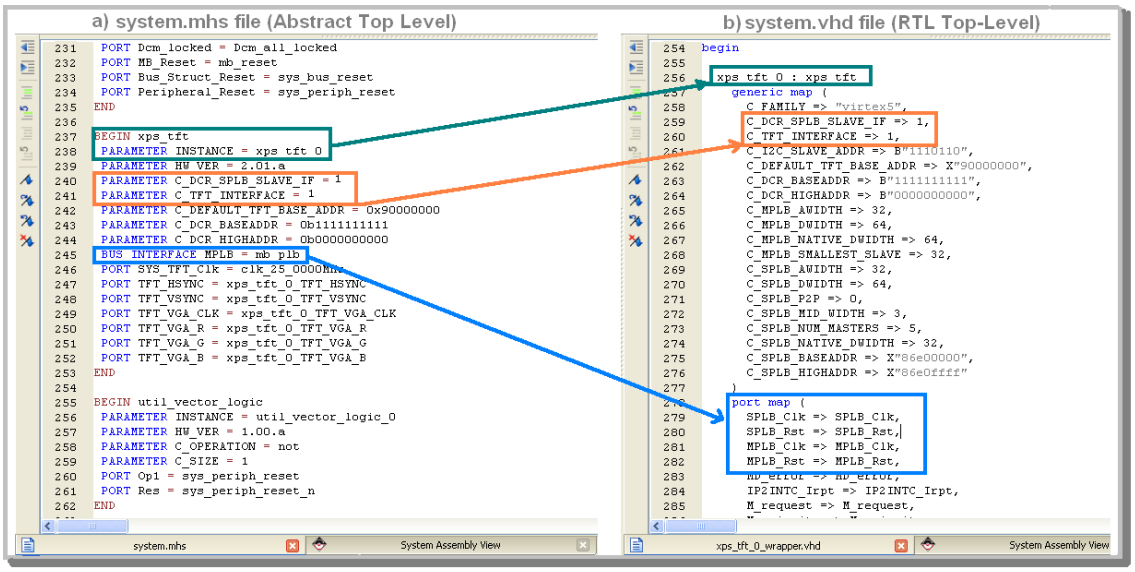


Figure 5.22: Relationship between the MHS model and the RTL top-level description

5. Finally, it generates the block RAM Memory Map (BMM) file which contains addresses and configuration of on-chip block RAM memories. This file is used later for initializing the block RAMs with software.

In the subsections that follow, we will discuss the role of the MHS file in more depth. In particular, we are interested in how this intermediate model abstracts the instantiation (parameterization and interconnection) of a set of IPs, and produces and synthesizes automatically a top-level VHDL description. As discussed in the previous chapters, this top-level netlist, along those of the PRM comprise the primary inputs of the Xilinx DPR design flow, hence our interest in generating the MHS file from the UML MARTE. In order to do so, we have already defined an IP reflection mechanism that enables us to access to the IPs metadata contained in the MPD files. This metadata is then used for generating the MHS file from UML MARTE models, while abstracting the use of heterogeneous and complex tools for the user.

### 5.6.2/ Platform Generation Chain: from MARTE to Xilinx XPS

In this section, we embark in a brief description of the platform creation branch of the MDE based FAMOUS methodology; a more in-depth discussion of each of the steps will be introduced in subsequent sections. This stage of the FAMOUS flow represents the design by reuse phase of any IP reuse methodology, and in the particular context of the FAMOUS approach, permits the composition of a DPR-based SoC platform in high-levels of abstraction, and its subsequent transformation into a Xilinx Platform Studio platform model. We make special emphasis on how the different tools required for the MDE development of our methodology are integrated into the DPR design flow. Furthermore, we

discuss the role of the platform models at each stage, as well as the associated model transformations. The complete component-based methodology for the generation of DPR platforms is depicted on Figure 5.23. The four libraries created in the first phase, and introduced in the previous chapter are coupled with Sodius MDWorkbench, but their role in this stage differs from Figure 5.4.

Once the IP libraries in different levels have been obtained, the MARTE Model Library (MML) can be used by UML Modeling Environment (which contains the associated MARTE extensions) to enable the creation of a platform using Composite Structure Diagram, in which the deployment phase takes place, as depicted on Figure 5.23 a). The deployed platform model created in the UML Modeling Environment is exported as an XMI file and imported into Sodius MDWorkbench (label b) on the figure). First, the MDE tool parses the XMI files using a model parser, and employing the associated IP deployment metamodel, along the IP-XACT design metamodel, produces a design object through a MARTE2IP-XACT model transformation.

As described in the previous sections, the IP-XACT design description obtained in this manner contains a set of elements (stored in the metadata schema) that completely and unequivocally describes the intended platform. Such elements are the component instances, their parameterized configurable elements, and the interconnections between the components. This is attained by integrating an IP deployment package in MARTE (a

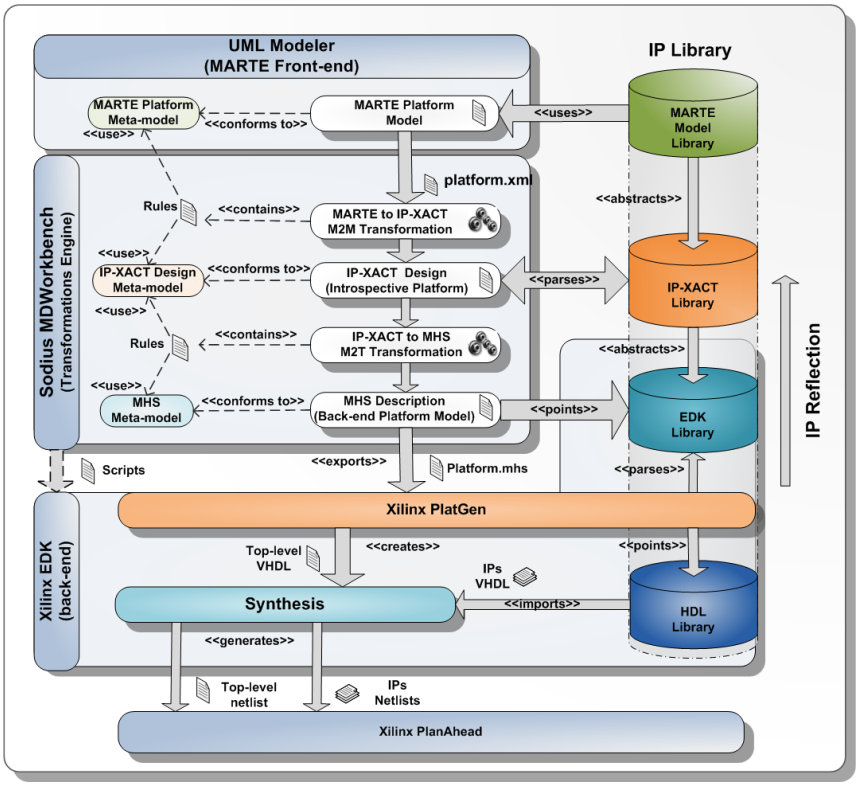


Figure 5.23: Metamodeling-driven approach for the creation of the platform

platform metamodel), which permits to import the minimum required information for composing the platform at high-levels of abstraction, and more prominently for performing the required model transformations. The obtained IP-XACT XML intermediate representation is flow agnostic, and can be used for targeting different back-ends (i.e. purely VHDL, Xilinx EDK). The selection of the desired back-end depends upon on the chosen generators (or in the context of an MDE approach, on the target metamodel); therefore, it is important to be able to select which implementation back-end is to be used. This is achieved in IP-XACT by choosing the adequate ⟨*view*⟩ in the IP-XACT component description, which allows to access the corresponding ⟨*modelParameters*⟩ of the instantiated components in the design description; to make this process easier, this selection is done in the UML model and propagated to the IP-XACT design.

Therefore, the design description is used as an introspective architecture in Sodius MDWorkbench to parse the IP-XACT component library. The parser seeks for parameters, ports and bus interfaces that are controlled by the ⟨*configurableElements*⟩ of the corresponding components instances; the retrieved information is then exploited, in tandem with the MHS meta-model, for generating the MHS file. This Xilinx-specific model invokes the encompassing IPs from the MPD and HDL libraries for their parameterization and integration at the XPS level. A top-level VHDL file is obtained by PlatGen, which retrieves the constituent static IP implementations for synthesis, as described in Section 5.6.1. The DPR IP tasks to be mapped in the reconfigurable partitions are retrieved from the VHDL library and synthesized independently (for clarity, this path is not shown in the diagram), as demanded by the Xilinx DPR design flow. For this, the IP classification provided in the previous chapter helps the parser in MDWorkbench which IPs are to be part of the top-level MHS description and which IPs need to be only parameterized and synthesized. Once the top-level and DPR IPs implementation have been synthesized, the obtained netlists are fed to Xilinx PlanAhead to generate the physical implementation of the DPR system, and subsequently the bitstreams necessary to configure the FPGA. A detailed discussion of each of the steps, the associated metamodels, and the implemented model transformations to move from MARTE to Xilinx Platform Studio MHS model will be discussed in the next sections.

## 5.7/ METAMODELS FOR PLATFORM GENERATION

In this section we discuss the different metamodels used in each of the phases of the FAMOUS system composition and generation flow. We start from the lowest abstract platform description, namely the Microprocessor Hardware Specification (MHS) listing its more important features and how the metamodel has been created and used in our methodology. Then, we analyze more in detail the IP-XACT design description and we delineate its relation with the MHS file. Finally, we introduce some extensions to MARTE

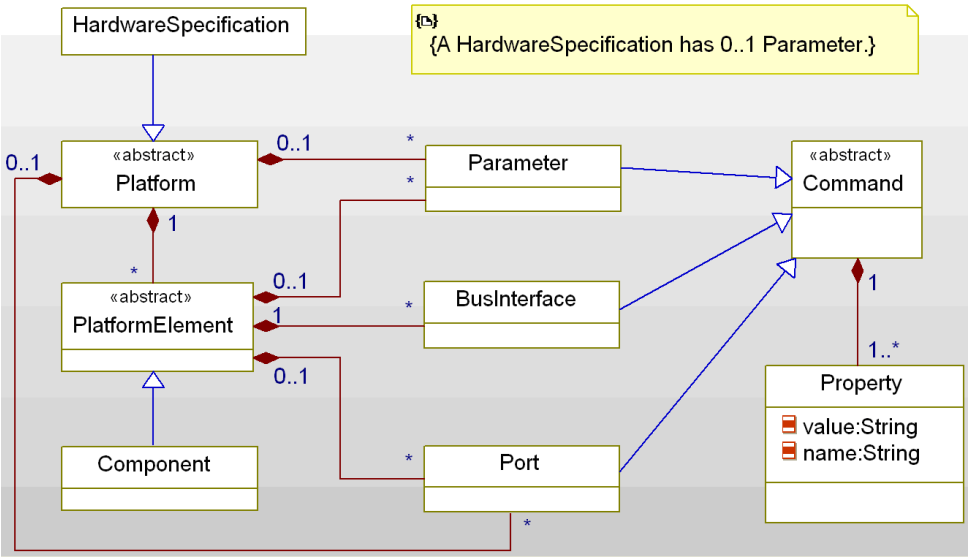


Figure 5.24: UML Model of Xilinx PSF Meta-model - Microprocessor Hardware Specification (MHS)

the have been defined to handle the modeling of SoC platform in UML, and how the IP deployment package allows to have all the required information for generating the lowest systems descriptions through model transformations. The latter are which are then discussed in the last section.

### 5.7.1/ Microprocessor Hardware Definition metamodel

As discussed previously, the MHS file is used by Xilinx XPS to store the information of the platform (in terms of component instances, their parameters, ports and bus interfaces). As with the MPD file, the information is this model is encoded in ASCII. The model is used as a Xilinx specific platform representation, and exploited by Xilinx tools such as Platgen to parse the IP representations to generate the top-level RTL description of the platform (acting as a netlister).

As with the MPD files, the metamodel for the Xilinx platform was not available, and thus, the MHS metamodel has been created using the Rhapsody UML Modeler and imported into MDWorkbench. The so-obtained metamodel is depicted on Figure 5.24. As for the MPD metamodel introduced in the previous chapter, a set of Platform Elements (belonging to a Platform, in this case, a Hardware Specification) can be present in a platform model; in a MHS file, these Platform Elements are labelled as Components (or Component instances). Each Platform Element in a Hardware Specification contains a set of attributes (Parameters, Bus Interfaces and Ports) which contain as well a chain of commands (a Value:Name pair, separated by commas). Similarly, a Platform instance can contain a set of Parameters and Ports (with their respective commands), which typically represent the generics and entity ports of the underlying top-level VHDL architecture.

An example of a section of the MHS file is provided in Figure 5.25, which shows some of the commands associated with a component instance. Xilinx XPS tools provide Graphical User Interfaces (GUIs) that deal with the parameters contained in the component instances, and if any change is performed through the GUIs, the associated tools parse the MPD files searching for dependent parameters, ports and bus interfaces. As mentioned in the previous chapter, the commands in the MPD files are defined in such a way that some parameters, ports and bus interfaces can be made dependent upon the user of other elements in the IP description. If we take again the example of the TFT controller introduced in that chapter, the type of control interface can be customized through the use of the C_DCR_SPLB_SLAVE_IF parameter, while the output interface (VGA or DVI) is controlled via the C_TFT_INTERFACE parameter. The effects of changing these two parameters are then reflected in the MHS file, as depicted on Figures 5.25 a) and b), in which new parameters and ports are updated depending upon the users choice.

An in-depth description of the MHS commands for each of the aforementioned attributes is outside of the scope of this manuscript (full details can be found in Xilinx Platform Specification Format Reference Manual [42]). However, their role will become more apparent when discussing the IP-XACT design metamodel, and in particular, when discussing how these two models are abstracted in UML MARTE.

In Chapter 4, we have described how Xilinx Platform studio automates the creation of FPGA-based SoC systems through the use of proprietary IP and system intermediate descriptions, the MPD and MHS files, respectively. As mentioned in the previous section, the MHS file abstracts the RTL top-level description of a SoC platform by simplifying the interconnection of relatively complex interfaces (by bundling the several IP ports into bus interfaces) and by providing the user with an easy way to modify IP parameters, which can be propagated during the system generation phase. Therefore, the MHS file represents a top-level platform with a set of IP instances, along with interconnection and parameterization information; this information is sufficient to parse the MPD files of the

```
239 BEGIN xps_tft
240  PARAMETER INSTANCE = xps_tft_0
241  PARAMETER HW_VER = 2.01.a
242  PARAMETER C_DCR_SPLB_SLAVE_IF = 1
243  PARAMETER C_TFT_INTERFACE = 1
244  PARAMETER C_I2C_SLAVE_ADDR = 0b1110110
245  PARAMETER C_DEFAULT_TFT_BASE_ADDR = 0x90000000
246  PARAMETER C_SPLB_BASEADDR = 0x86e00000
247  PARAMETER C_SPLB_HIGHADDR = 0x86e0ffff
249  BUS_INTERFACE SPLB = mb_plb
250  PORT SYS_TFT_Clk = clk_25_0000MHz
251  PORT TFT_HSYNC = xps_tft_0_TFT_HSYNC
252  PORT TFT_VSYNC = xps_tft_0_TFT_VSYNC
253  PORT TFT_DE = xps_tft_0_TFT_DE
254  PORT TFT_DVI_CLK_P = xps_tft_0_TFT_DVI_CLK_P
255  PORT TFT_DVI_CLK_N = xps_tft_0_TFT_DVI_CLK_N
256  PORT TFT_DVI_DATA = xps_tft_0_TFT_DVI_DATA
259 END
```

a) DVI Interface

```
237 BEGIN xps_tft
238  PARAMETER INSTANCE = xps_tft_0
239  PARAMETER HW_VER = 2.01.a
240  PARAMETER C_DCR_SPLB_SLAVE_IF = 0
241  PARAMETER C_TFT_INTERFACE = 0
242  PARAMETER C_DEFAULT_TFT_BASE_ADDR = 0x90000000
243  PARAMETER C_DCR_BASEADDR = 0b1111111111
244  PARAMETER C_DCR_HIGHADDR = 0b0000000000
245  BUS_INTERFACE MPLB = mb_plb
246  PORT SYS_TFT_Clk = clk_25_0000MHz
247  PORT TFT_HSYNC = xps_tft_0_TFT_HSYNC
248  PORT TFT_VSYNC = xps_tft_0_TFT_VSYNC
249  PORT TFT_VGA_CLK = xps_tft_0_TFT_VGA_CLK
250  PORT TFT_VGA_R = xps_tft_0_TFT_VGA_R
251  PORT TFT_VGA_G = xps_tft_0_TFT_VGA_G
252  PORT TFT_VGA_B = xps_tft_0_TFT_VGA_B
253 END
```

b) VGA Interface

Figure 5.25: Two examples of a component instance in an MHS file by modifying controlling parameters

instantiated components and generate a top-level VHDL file that can then be synthesized. Nevertheless, as discussed in Chapter 2, the SoC industry has been fostering the use of flow-agnostic metadata-based IP and system representations, such as IP-XACT. The consortium behind the standard have invested a great deal of work in defining a set of concepts that can be used for almost any purpose in the development of SoC platforms; from the description of the MHS file in the previous section, and of the IP-XACT description, it should be clear by now that the intentions of both descriptions overlap.

The MHS file could be used as the intermediate representation in a MDE-based composition framework for DPR systems by defining a MHS metamodel but, as mentioned in Section2.4.2, this would severely tie the high-level models to the chosen back-end. We have then opted for an approach that makes use of the IP-XACT design as the intermediate metamodel; using IP-XACT decouples the methodology from the back-end by providing a standard XML representation that can then be used for generating the necessary back-end representation. In our approach, RTL could be generated directly from the IP-XACT description by using a tool such as Magillem Studio, but due to certain methodological constraints (i.e. the use of MDE-based techniques and of Xilinx Platform Studio), we have opted to feed the IP-XACT description to MDWorkbench to, through model transformations, generate the MHS file used by Xilinx XPS. The transformations are relatively straightforward, given that most of the elements in both models are relatively the same; however, this was only possible through the creation of the multi-level library introduced in the previous chapter.

As described before, MDWorkbench contains the metamodels for all the Xilinx XPS and IP-XACT objects. Therefore, it contains as well a metamodel for the IP-XACT design, which will be introduced in the next section. However, the IP-XACT design file, which is a model that conforms to the IP-XACT design metamodel, has to be supplied. This is done by transforming the UML MARTE platform description into the necessary IP-XACT design file; in order to make this possible, a metamodel containing certain extensions must be defined in MARTE, along the necessary transformation rules. In this way, a complete generation chain from UML MARTE to Xilinx XPS is obtained, which will be the subject of the next sections.

### 5.7.2/ IP-XACT Design Metamodel

An IP-XACT design is the central placeholder for the assembly of component objects metadata. As with other schemas in the IP-XACT standard, a design object is stored in an arborescent XML stack, which can be archived in the IP library as an hierarchical IP and referenced via a ⟨*VLNV*⟩ id. A design artifact describes a list of components instances referenced by their ⟨*vlnv*⟩ tags, along configuration information, and their interconnections to each other, as depicted in the schema of Figure 5.26. There are three types of connections in a design: interconnections, ad-hoc connections and hierarchical connections. In

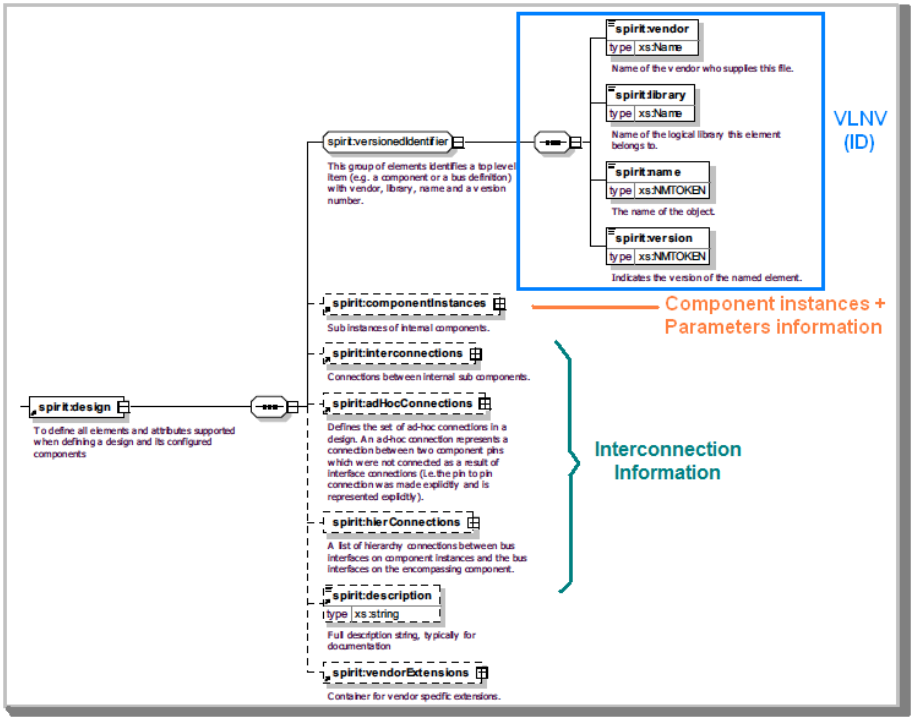


Figure 5.26: IP-XACT Schema for an IP-XACT design object

Systems-on-Chip, the sub-elements in a platform are interconnected using bus interfaces (that conform to predefined bus definitions). Interconnections represent a convenient and abstract way of describing the relationship between two component interfaces with a single connection element. On the other hand, ad-hoc connections describe point-to-point links between component instances in a design (e.g. clocks, interrupts), for cases in which such signals cannot be bundled in a bus definition. Finally, hierarchical connections refer to bonds between component instances in the design description and ports/interfaces in the encompassing top-level component. Thus, a hierarchical connection is analogous to the mapping of an internal signal and the top-level port of the encompassing VHDL top-level architecture.

The discussion above clearly shows that a design description represents in fact a schematic of components, as depicted on Figure 5.27. Many P-XACT enabled design tools support visual design capture capabilities. The difference between these tools and the MDE approaches proposed in the literature is that the latter make use of UML MARTE as a modelling front-end and of the IP-XACT design as an intermediate model. This is analogous to the Xilinx Platform Studio use of the MHS file and its somewhat reduced visual composition capabilities, and the underlying top-level VHDL description.

Similarly to the MHS file, the IP-XACT design also contains a set of ⟨*componentInstances*⟩ (referenced via the VLNV id), as depicted on the sub-schema of Figure 5.28. Each ⟨*componentInstance*⟩ is accompanied by a set ⟨*configurableElements*⟩ (a Name:Value pair, which are analogous to parameters in the MHS file and to generics in a VHDL

component) that correspond to certain ⟨*modelParameters*⟩ in the corresponding IP-XACT component descriptions. As mentioned in the previous chapter, we have classified the ⟨*modelParameters*⟩ into several ⟨*configGroups*⟩ in order to facilitate the creation of the IP-XACT design; thus, only those parameters defined as ⟨*immediate*⟩ in the ⟨*view:modelParameters*⟩ section of the component description appear in the design XML.

In order to make the connection between the parameters in the MHS file and the ⟨*configurableElements*⟩ in the design description, Figure 5.29 depicts the component instance for the TFT_Ctrl presented in the previous subsection. However, unlike the MHS file, that defines the interconnections only in terms of labels under the component instances (using ports and bus interfaces), the IP-XACT design separates the different kinds of connections under several elements in the XML stack, as depicted on Figure 5.27. The connections are referenced using the information of the component instances they attach; this facilitates the classification (and subsequent processing of the XML design file during generation) of the connections, given that bus interconnections are inherently different from ad-hoc and hierarchical connections.

As an example, Figure 5.29 shows a design description in the Eclipse IP-XACT Editor, encompassing a set of component instances and their interconnections. As it can be observed, in an IP-XACT description, the component instances are described in a relatively abstract manner, leaving the components low-level details (e.g. signal names, directions, widths) stored in the corresponding IP-XACT descriptions. The component instances contain a set of configurable elements that are propagated as controlling parameters during the design build phase. After the design description has been created during the design capture stage, it is used as an introspective architecture for accessing the components information and, through generators, for creating the chosen back-end

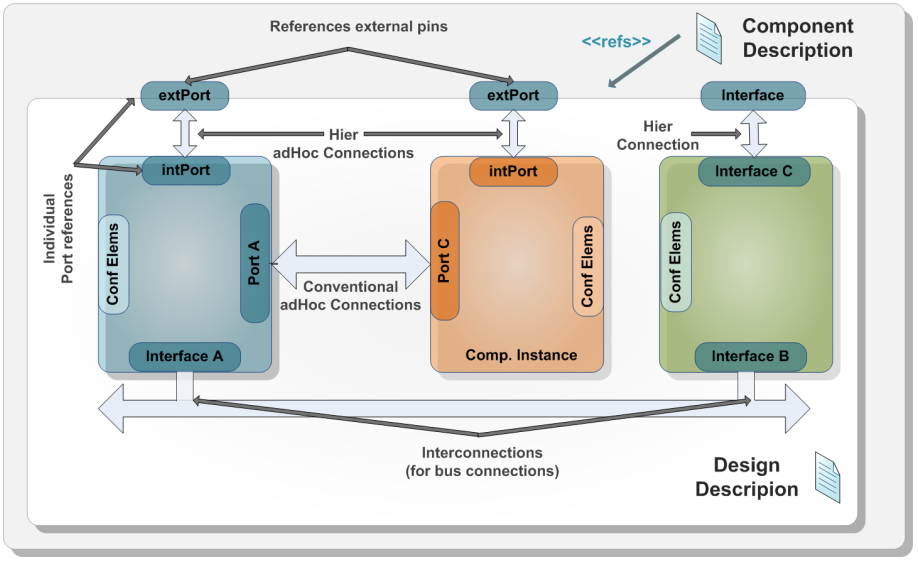


Figure 5.27: IP-XACT design block representation showing the main concepts

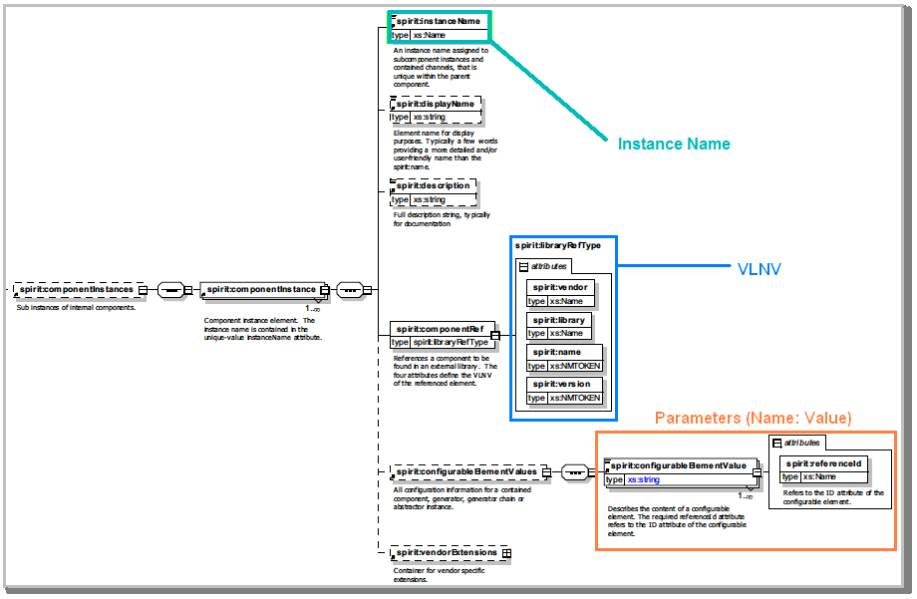


Figure 5.28: IP-XACT Schema the component instances on a design description

top-level input. For instance, in an RTL or TLM based approach, synthesizable netlists, as depicted on Figure 5.20 can be obtained. If the chosen back-end is an RTL representation, the complete information about the entity ports and generics is retrieved, and a top-level VHDL containing the detailed component instances is created. The signals interconnecting the component instances are obtained from the interconnections section of the design XML file, and used for defining the relations between the component instances in the VHDL file; in this manner, one of the most prone to errors phases of the SoC created can be easily automated.

While a design description describes most of the information for a design, some information is missing, such as the exact port names used by a hierarchical (i.e. external) bus interface. To resolve this, a component description (referred to as a hierarchical com-

topLevel_Design.spirit

```
<spirit:componentInstance>
   <spirit:instanceName>xps_tft_0</spirit:instanceName>
   <spirit:componentRef spirit:version="2.01.a"
       spirit:library="xilinx_EDK_cores" spirit:name="xps_tft" spirit:vendor="Xilinx" />
   <spirit:configurableElementValues>
       <spirit:configurableElementValue
           spirit:referenceId="C_DCR_SPLB_SLAVE_IF">1</spirit:configurableElementValue>
       <spirit:configurableElementValue
           spirit:referenceId="C_TFT_INTERFACE">1</spirit:configurableElementValue>
       <spirit:configurableElementValue
           spirit:referenceId="C_I2C_SLAVE_ADDR">0b1110110</spirit:configurableElementValue>
       <spirit:configurableElementValue
           spirit:referenceId="C_DEFAULT_TFT_BASE_ADDR">0x10000000</spirit:configurableElementValue>
           <spirit:configurableElementValue
           spirit:referenceId="C_SPLB_BASEADDR">0x86e00000</spirit:configurableElementValue>
       <spirit:configurableElementValue
           spirit:referenceId="C_SPLB_HIGHADDR">0x86e0ffff</spirit:configurableElementValue>
   </spirit:configurableElementValues>
</spirit:componentInstance>
```

Figure 5.29: Snapshot of a component instance in an IP-XACT design description

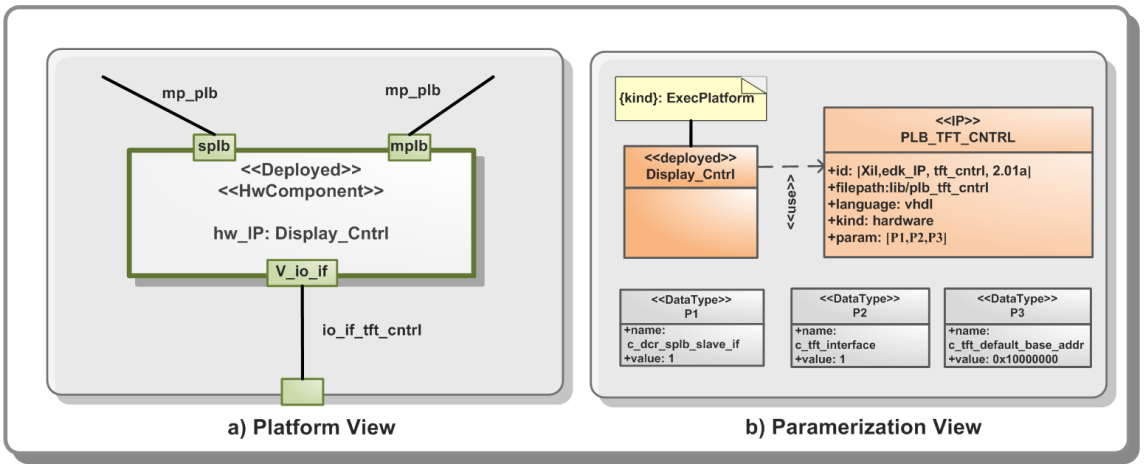


Figure 5.30: Snapshot of a platform model in MARTE, showing the a) Platform and b) Parameterization views

ponent) is used. This component description contains a ⟨*view*⟩ with a reference to the design description. Together, the component and referenced design description form a complete single-level hierarchical description, as depicted in the envelope of Figure 5.27.

## 5.8/ UML MARTE Proposed Modeling of the Platform

As depicted in Figure 5.23, the designer of a high-level platform imports a set of components from the MARTE Library into the chosen Modeling Environment. This library has been previously populated by transforming the IP-XACT components into MARTE deployment templates, as described in the previous chapter. We make use of the proposed deployment IP metamodel that allows us to perform this mapping and to promote IP reuse in our methodology by linking the high-level components to their IP-XACT counterparts. Moreover, it enables the creation of a *Parameterization View*, which contains predefined parameters that control the inclusion of ports, bus interfaces and other parameters in the final MHS model (and subsequently in the underlying VHDL IP implementations). Instead of having a complete IP-XACT meta-model in MARTE, our simplified IP meta-model allows us to have a reduced set of elements, which can be easily transformed in both directions, as depicted on Figure 5.30. This fact could be exploited for importing complex platforms which have been created using other tools, which may help the designer in UML to concentrate solely in the allocation of the DPR modules onto the available DPR placeholders, further simplifying the proceedings and promoting design by reuse; in such scenario, the platform to be used does not need to be created, but only reused and imported into UML MARTE.

| UML MARTE Model | IP - XACT Design |
|---|---|
| *Param View (for each IP class)* | **spirit:componentsIntances** |
| *IP_n = name*<br>**id: identifier**<br>**Parameter_n = name** | *create componentInstance = name*<br>**spirit: version: library: name: vendor**<br>**configurableElement_n:value** |
| *Platform View (for each instance)* | **spirit:Interconnections** |
| *if (port kind = busIF)*<br>**Connector = name**<br>**Port = name** | *create interconnection*<br>**interconnection:name**<br>**spirit:activeInteface:name** |
| *Platform View (for each instance)* | **spirit:adHocConnections** |
| *if (conn = internal)*<br>**Connector = name**<br>**Port = name** | *create adHocConnection*<br>**adHocConnection:name**<br>**internalPortReference:name** |
| *if (conn = external)*<br>**Connector = name**<br>**Port_int = name**<br>**endpoint = name** | *create adHocConnection*<br>**adHocConnection:name**<br>**internalPortReference:name**<br>**externalPortReference:name** |

Figure 5.31: MARTE_2_IP-XACT Transformation rules

## 5.9/ Proposed Model Transformations for creating the hardware platform

In this section we provide an in-depth description of the implemented model transformations required to move from the high-level models in UML MARTE (a platform description in a deployment model) to an intermediate representation via an IP-XACT design and, ultimately, to the MHS model used by Xilinx tools to generate the VHDL top-level design of the SoC platform.

### 5.9.1/ MARTE ↔ IP-XACT Transformation Rules.

Once the designer has parameterized and composed its platform in a MARTE-compliant modeling environment, the UML platform model is fed to Sodius MDWorkbench and parsed in order to generate an IP-XACT design description from the *Parameterization* and *Platform* views (Figure 5.30). The obtained XML file contains a ⟨*spirit:design*⟩ entry, which identifies it as the top level element in a SoC design or hierarchical component. The first UML diagram, the *Platform* view contains a set of ⟨⟨*parts*⟩⟩, connected to other components via ⟨⟨*ports*⟩⟩ through the so-called ⟨⟨*connectors*⟩⟩.

The UML MARTE extensions introduced in the previous section permit to completely specify the required information for generating the IP-XACT design (e.g. names of the connections, instance names and parameters/choices for configuration); the transformation rules for generating the IP-XACT design are shown in Figure 5.31, and discussed as follows.

For the purpose of clarity, an example is provided on Figure 5.32. As the design file is comprised of four main sections, each one is created sequentially; the first one corresponds to the ⟨*componentInstances*⟩ section, which is mostly inferred from the *Parameterization* view, as depicted on the left side of the figure. For each ⟨⟨*deployed*⟩⟨ component in the MARTE model, an ⟨*instanceName*⟩ element is created in the ⟨*spirit:componentInstances*⟩ section of the IP-XACT design file, along the corresponding ⟨*configurableElements*⟩.

On the other hand, the system connectivity information is obtained from the MARTE *Platform View* diagram, which contains ⟨⟨*parts*⟩⟨ instances with the associated ⟨⟨*ports*⟩⟨. The ports definitions have been extended in MARTE to classify them into three groups: bus interfaces, internal ports, and external ports; this enables the mapping to the corresponding IP-XACT elements (bus interfaces in the interconnections section, and internal and external ports in the ad-hoc connections section) Thus, ⟨⟨connectors⟩⟨ elements in MARTE are mapped to IP-XACT ⟨*interconnection*⟩ or ⟨*adHocConnection*⟩ depending upon the kind of ports they link. Finally, the external pins information is obtained from the ⟨*ports*⟩ labels in the enclosing IP-XACT component description, and written into the

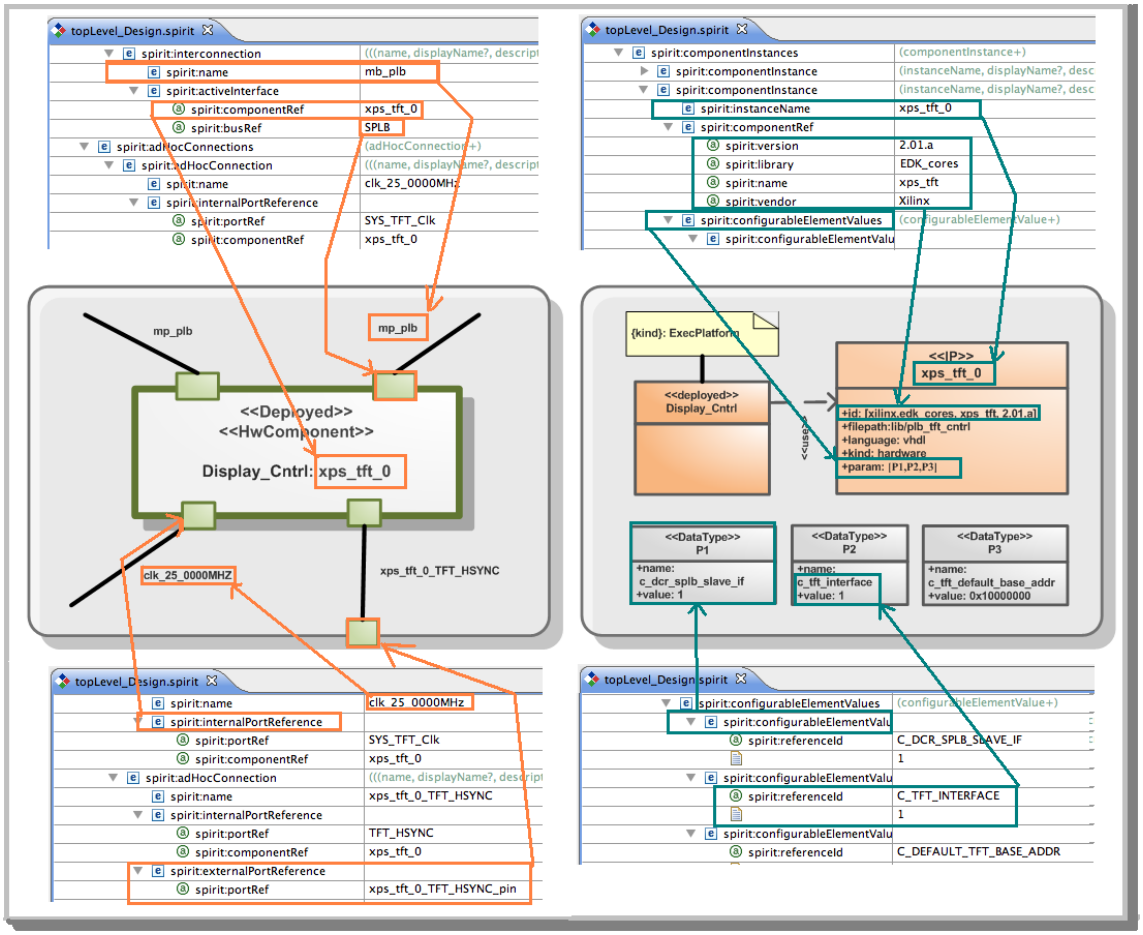


Figure 5.32: MARTE_2_IP-XACT mapping example

| MHS Command | IP-XACT Designt Counterpart |
|---|---|
| *PARAMETERS* | **CONFIGURABLE ELEMENTS**<br>Desing:componentInstances:ComponentInstance |
| INSTANCE | spirit:InstanceName |
| HW_VER | spirit:componentRef_spirit:Version |
| PARAM_i = VALUE_i | spirit:configurableElementValue<br>spirit:referenceId = value |
| *BUS INTERFACES* | **ACTIVE INTERFACES( Design:interconnections)** |
| BUS INTERFACE i<br>INTERFACE = bus_std | for each(interconnection)<br>if (componentRef = "instanceName")<br>activeInterface:busRef = interconnection:name |
| *PORTS* | **INTERNAL PORT REFERENCES (Design:adHocConnections)** |
| internal ports<br>PORT_i = VALUE_i | for each(adHocConnection)<br>if (componentRef = "instanceName")<br>internalPortRef:portRef = adHocConnection:name |
| external ports<br>PORT_i = VALUE_i | for each(adHocConnection)<br>if (componentRef = "instanceName")<br>externalPortRef:portRef = adHocConnection:name |

Figure 5.33: IP-XACT to MHS mappings for the top-level hardware description

⟨*spirit:adHocConnections*⟩ elements attached to an external port reference.

### 5.9.2/ IP-XACT ↔ MHS TRANSFORMATION RULES.

The second step in the compilation chain is the generation of the back-end (Xilinx Platform studio) MHS model, as depicted on Figure 5.23. The MHS file is created from an IP-XACT design description; this is achieved via a Model-to-Text transformation using the associated IP-XACT and MHS metamodels in Sodius MDWorkbench and a set of transformation rules (depicted on Figure 5.33), which will be described in the next subsections.

#### 5.9.2.1/ MAPPING OF THE BUS PARAMETERS

The MHS file contains a set of component instances that are linked to the components in the EDK MPD library via two elements: the NAME and the HW_VER tags (i.e. xps_tft and 2.01.a), that are used by Xilinx tools to locate the components implementations in the Xilinx XPS IP core directories. These elements are inferred from the ⟨*name:value* ⟩ pair in the ⟨*VLNV*⟩ element of the corresponding ⟨*componentInstance*⟩ in the input design description, as depicted on the top of Figure 5.34. Then, the ⟨*configurableElements*⟩ in the IP-XACT design are directly converted into PARAMETERS in the component instance of the MHS file. As mentioned previously, some parameters in the component description might depend on these controlling parameters/constraints and therefore, the IP-XACT design is used as an introspective architecture; the referenced IP-XACT components descriptions are parsed for retrieving any dependent ⟨*modelParameters*⟩, which are then written to the MHS file as PARAMETERS.

There are two scenarios in which controlling parameters play a crucial role: the first, for parameters which are only visible in the MHS file if the controlling parameter is TRUE and on the other hand, when the value of a parameter depend on the value of another via a logical or algebraic expression, which must be resolved during the parsing phase.

#### 5.9.2.2/ MAPPING OF THE BUS INTERFACES

The second kind of element to generate in the component instances section of the MHS file are the bus interfaces information, as depicted on the bottom of Figure 5.34. This is relatively simple due to the use of bus definitions (a name:value pair that abstracts and references the low-level details such as ports, along their direction and width). The bus interfaces information (e.g. master or slave, PLB or AXI) are inferred from the ⟨*interconnections*⟩ elements, using ⟨*busRef:name*⟩ tag for the given interface (and ⟨*activeInterface*⟩). All the interfaces connected to the same bus will thus have the same name for the bus connection, which is retrieved from the UML MARTE model when creating the IP-XACT design description.

#### 5.9.2.3/ MAPPING OF THE INTERNAL AND EXTERNAL CONNECTIONS AND PORTS

As mentioned in the previous section, there are two types of point-to-point connections from a component instance in a platform description. The first are connections from/to an internal signal to a component. Examples of these connections are clocks, interrupts and reset signals; this information is written into the MHS file under the component instance section, and retrieved from the ⟨*adHocConnections*⟩ elements of the IP-

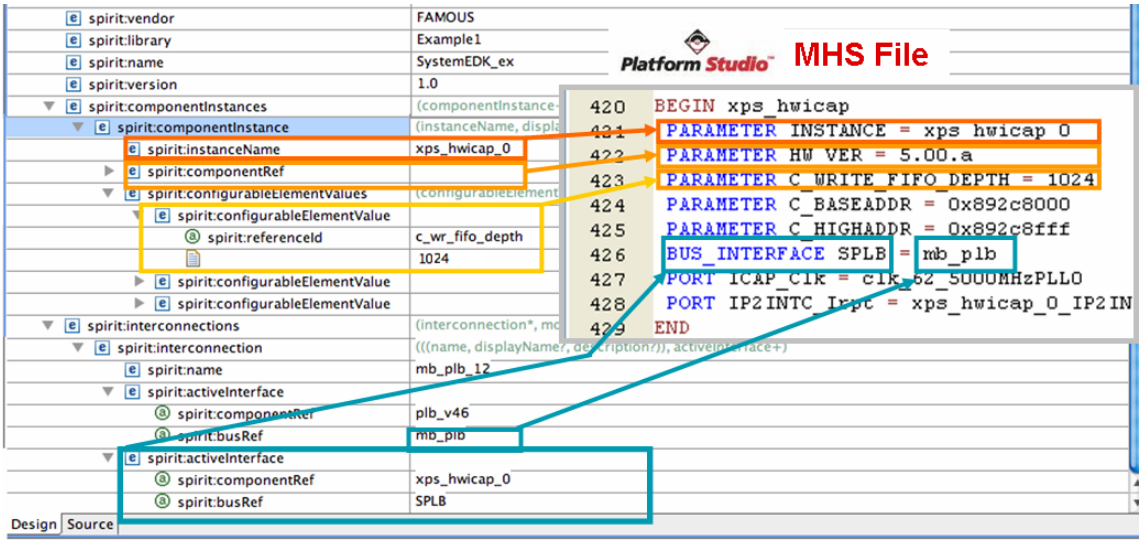


Figure 5.34: Example of mapping between parameters and bus interfaces in the MHS file and the IP-XACT description

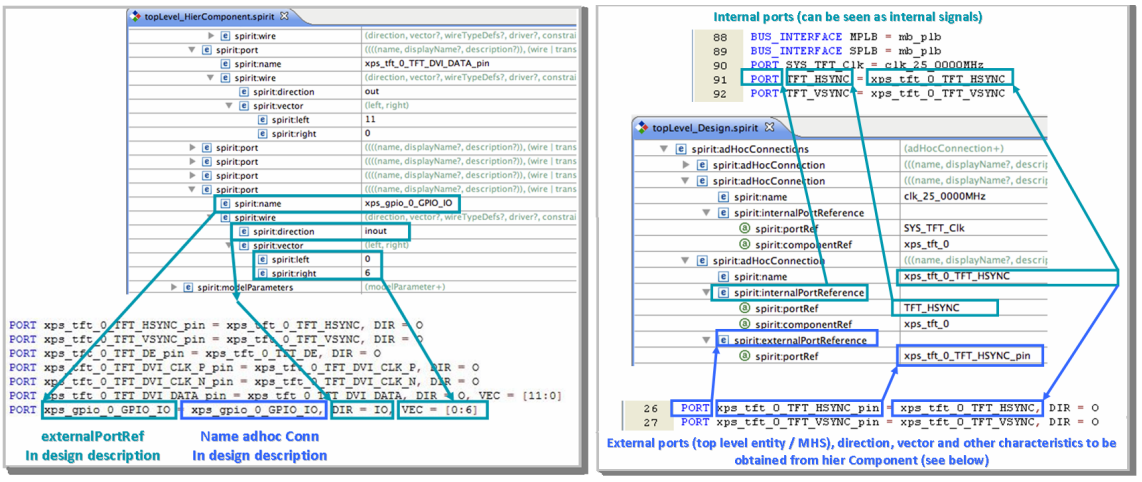


Figure 5.35: Mapping between external ports in the MHS and IP-XACT descriptions

XACT design file. External connections represent links between internal ports in the component instances, and hierarchical ports in the encompassing top-level architecture; these connections usually represent I/O data signals, input clocks for subsystems in the FPGA fabric, among others. In order to identify internal ad-hoc connections from ad-hoc hierarchical (external point-to-point connections), the parser must check whether or not both an ⟨*internalPortReference*⟩ and an ⟨*externalPortReference*⟩ are present in the ⟨*adHocConnection*⟩ element. If only one label is present, a normal PORT, along the signal that connects to the other component, are written into the MHS file, as depicted on 5.35 a)

On the other hand, if both tags are used, the ⟨*internalPortReference*⟩ is written under the component instance as a PORT = NAME pair (where the name represents the label of the adHoc connection), while the ⟨*externalPortReference*⟩ is written as a separate port on the top of the MHS file (i.e. xps_tft_0_TFT_HSYNC_pin on the figure). These individual ports correspond to those of the encompassing VHDL top-level entity, and correspond to the labels assigned to a given pin in the User Constraint File; the name of the ad-hoc connection corresponds then to the signal used to interconnect the internal port in the component instance and the external port of the platform.

As mentioned before, the IP-XACT design does not contain all the required information of a top-level VHDL design. Metadata such as top-level parameters and ports must be stored into a hierarchical component description that references the underlying design description. For instance, Figure 5.35 b) shows an example of how the information of the top-level PORTS in the MHS is obtained from the IP-XACT component description of the platform; examples of the associated commands for an external PORT in the MHS file are the direction DIR and the width of the port encoded in a vector VEC.

### 5.9.3/ ROLE OF THE PARTIALLY RECONFIGURABLE MODULES IN THE FLOW

The transformation rules presented in the previous section are used for generating the top-level static design, containing a set of IP blocks wrapped by the IPIF interface, and therefore, leading to a scenario in which they can be seen as black-boxes exposing a set of parameters to be resolved and interfaces to be connected by the designer. These black-boxes are in fact UML templates which are based on deployment extensions introduced in the previous chapter, and that help to map several concepts to elements in the IP-XACT design description, and subsequently, to the Xilinx MHS platform description. An example for the top-level generation was provided on Figure 5.32; however, that example assumes that all parameters are to be set by the designer (or defined as user in the configuration groups of the IP-XACT component description). Nonetheless, inn Chapter 4 we have described how the DPR wrappers (with and without context support) are to be parameterized: the information of the Reconfigurable Partition (RP) interfaces (i.e. data widths, burst support, among other parameters) has to be accessible in the component instance as with all the other top-level components.

The manual configuration of the interface width information is not a very effective method: a RP can hold many PRMs, containing many different widths, whose UML representations (as it will be illustrated in the next chapter) are split in several diagrams. Therefore, the configuration of the DPR Wrappers RP interfaces is done automatically by retrieving the interface information of the PRMs (tasks IPs) from the IP-XACT component library and using it to customize the interface of the DPR wrapper, as depicted on Figure 5.36 b) and c), where the d_out_width parameter for two different PRM configurations are retrieved for setting the corresponding element in the DPR_Wrapper_Membrane. The configured

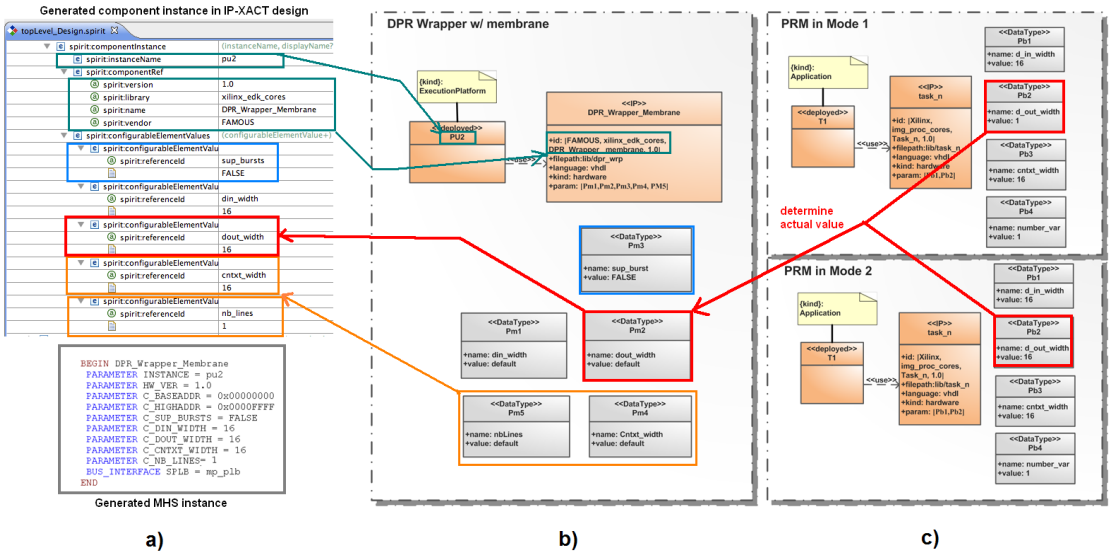


Figure 5.36: The role of the PRM metadata in the system generation flow

DPR Wrapper, along the rest of the static components is used for generating the top-level IP-XACT ⟨*design*⟩; the membrane dependent parameters just described are converted into ⟨*configurableElements*⟩ (which have been classified with a ⟨*generated*⟩ attribute under the ⟨*configGroups*⟩ of the associated element, indicating that such parameters are set by external generators) under the corresponding ⟨*componentInstance*⟩, as depicted on Figure 5.36 a). As described in the IP-XACT to MHS transformation section, the ⟨*configurableElements*⟩ are transformed into PARAMETERS in the MHS description (as depicted on the figure as well), which represent generics associated with the RP in the underlying DPR Wrapper HDL implementation.

As mentioned before, the goal of this thesis work was to foster IP reuse and design by reuse capabilities to the FAMOUS framework, whilst providing a means to facilitate the design entry phase of the Xilinx DPR design flow. As depicted on Figure 5.37 a), the top-level description, containing static modules and DPR wrappers (encompassing customizable Reconfigurable Partitions) is obtained from the UML MARTE deployment level, which is converted into an IP-XACT description and subsequently into an Xilinx XPS MHS file. The advantage of using IP-XACT as and XMI-IR are manifold, but in this context, enables to generate a handful of back-end representations; then, the HDL top-level description could be generated directly from the IP-XACT description, since it already contains all the necessary information (e.g. generics, ports, file paths and build commands information, among others). This branch of the design flow generates an ensemble of modules that are converted into netlists/bitstreams through synthesis and the DPR flow in PlanAhead, respectively (as depicted on the top of Figure 5.37 b)). The static modules are obtained

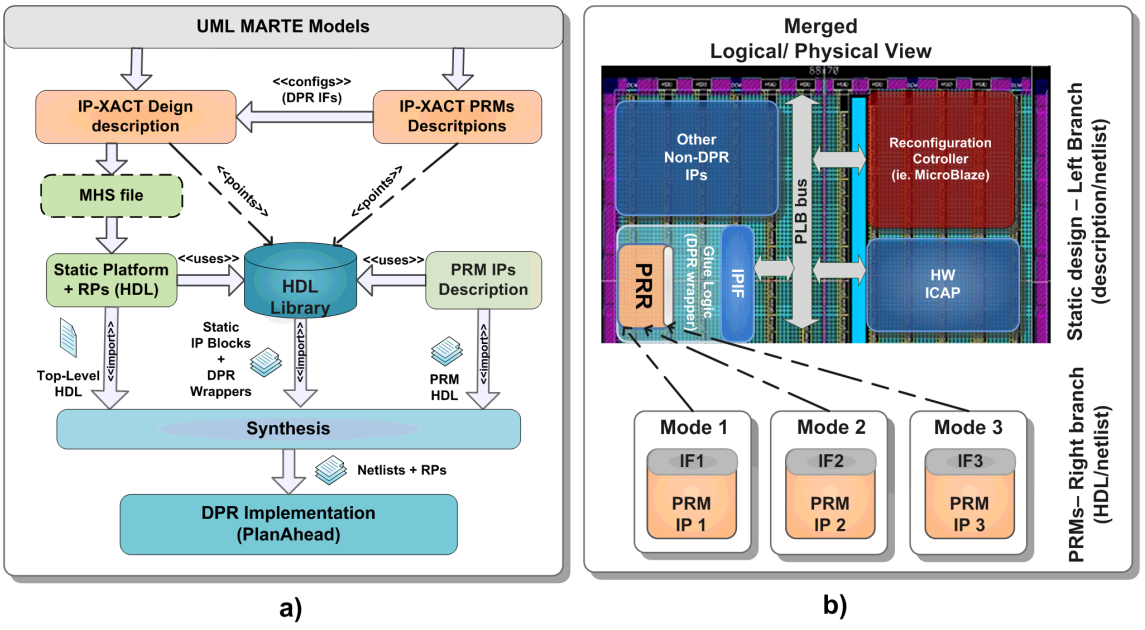


Figure 5.37: Relationship between the proposed flow (a) and the DPR system implementation (b)

from the left branch of the flow (i.e. MHS). In parallel, and as described in detail in Chapter 2, the Partially Reconfigurable Modules (PRMs) are synthesized independently and fed to PlanAhead (once more, by using the file path information stored in the XML description), in which they are assigned to Partially Reconfigurable Regions (PRRs), corresponding to the RPs defined in the logical description. However, before synthesizing the DPR Wrappers, their interfaces and other parameters must be configured in advance by retrieving metadata from the PRM IP-XACT component descriptions, which are used for setting the configurable elements in the corresponding component instances in the IP-XACT design description. These tasks have been automated by generators in MDWorkbench, as described previously, easing one of the most prone to errors phases of the DPR design flow and overall facilitating the design process.

## 5.10/ Discussion and Conclusions

In this chapter we have described in detail our approach for providing IP reuse capabilities to the FAMOUS framework, a feature lacking in many MDE-based methodologies, and which is attained by reflecting IP metadata from Xilinx XPS Peripheral Descriptions into UML MARTE templates. The rationales behind this strategy are manifold. First, since we have made use of a component-based approach, the entire IP functionality is encompassed by a single, top-level IP description, in which the information of the ports is abstracted by bus interfaces, and the (eventual) configuration of the IP is contained in few configurable parameters, that are propagated during the compilation chain without the intervention of the high-level modeler. This is only possible due to the way the IP blocks have been coded, as introduced in the previous chapter, making them as modular and generic as possible. This schema enables to reflect a reduced set of features into the UML MARTE XML templates; thus, the proposed IP Deployment metamodel provides the designer at high-levels of abstraction with a visual front-end in which the low-level features of the components (e.g. parameters and connectivity information) are deployed in a very abstract manner. The use of IP-XACT as an intermediate representation has been already discussed in great detail: enables IP and system representation tool interoperability, fostering the research in the SoC community; moreover, having such standard XML-IR permits to easily adapt the metadata reflection mechanisms (for targeting new front or back-end), by solely modifying the transformation rules.

Furthermore, the IP-XACT component description contains rich information about the IP implementation artifacts, and the associated build commands for obtaining concrete results in a given design flow. This feature provides two benefits: the first one being able to easily associate an UML component instance to a very often complex IP implementation (i.e. a DDR controller code can be comprised of hundreds of HDL files). Second, the rich EDA information contained within a single IP-XACT component description enables

the creation of tool chains, which can target multiple back-ends; once again, this can be customized by using a different set of generators (e.g. transformation rules in MDE). This capability abstracts the eccentricities of the IP implementation, the design tools and the tools flow options away from the designer. This is in fact one of the contributions of this work: facilitating the design entry phase of DPR design flow. This aspect will be discussed in more detail subsequently, when addressing the design by reuse (system composition and generation) components of this work.

Subsequnetly, in the second part of this chapter we have described in great detail the second step in a component-based design methodology for SoC: the design by reuse stage, in which IP components are retrieved from the library and instantiated in a top-level description. This instantiation step implies the interconnection of the components and basic parameterization for the configurable (usually soft) IPs. As with other Computer-Aided Design approaches, this step done at a high-level of abstraction, dealing not with the low-level component implementations, but with an abstract representation more amenable for being processed automatically by a machine, through a metamodel. At this level, a high-level platform representation containing enough information for creating the so-called introspective architecture is obtained though a MARTE to IP-XACT model transformation. Then, this introspective architecture (an IP-XACT design object) is used for parsing the IP-XACT component descriptions, seeking for dependent parameters, ports and bus interfaces, which if valid, are used along those contained in the design description for generating the Xilinx-specific Microprocessor Hardware Specification (MHS) file. This representation parses the MPD files (gathering the underlying HDL representations), for generating the top-level system description.

Nonetheless, we must emphasize an important point: the proposed methodology is not only valid for Xilinx tools. We have decided to tackle this particular IP representation since the tool enables the creation of complex embedded systems in which the hardware and software components of the SoC platform can be jointly developed. In order to plug the FAMOUS framework with Xilinx Platform Studio, the definition of the corresponding MHS and MPD metamodels and transformation rules was a necessity. For any MDE approach to be applicable, most of the modeling must be technologic independent: but this is only true before the deployment level. The work presented in this manuscript address this modeling phase, in which the high-level models are allocated to back-end specific artifacts (i.e. the IP components in VHDL) with design flow EDA information (e.g. a tool flow and its associated metadata). Regardless of the chosen back-end, an IP-XACT design is created from UML MARTE, and the back-end generation phase depends only on the used generators (i.e. in MDE, model transformations); this is achieved by storing IP-XACT components with back-end views and the associated parameters and vendor extensions, which are only parsed on the context of a particular phase. Thus, the FAMOUS approach should be easily adapted to other design flows which make use or pure VHDL representations, and where the C code to be run on the processor is

generated independently.

Until know, we have only discussed the generation of the top-level, static architecture. This is because the DPR Wrappers IPs introduced in Chapter 4 are stored and processed as any other component in the library, due to the interface and metadata standardization process they all underwent. However, the dynamicity of the DPR platform, and the configuration of the DPR Wrappers are obtained from the Partially Reconfigurable Modules to be assigned to these IP place-holders, and this information is also in the deployment model. These concepts will be further explored in the next chapter, where a case study will be analyzed. Then, the reader will have a bigger picture of how the concepts discussed in the second part of the manuscript are used for modeling DPR systems. We will also discuss how the proposed MDE design by reuse approach eases the design process, providing design efforts comparisons and implementation results.

# 6

# CASE STUDY: DPR ARCHITECTURE FOR IMAGE PROCESSING

## Contents

## 6.1/ INTRODUCTION

In this chapter we intend to provide the reader with a glimpse on how a complete DPR system can be created at high-levels of abstraction by using UML MARTE. As described in Chapters 4 and 5, several components descriptions (i.e. Microprocessor Peripheral Definitions or MPD files) in the Xilinx Platform Studio design suite (e.g. processor, communication and video IPs, among others) have been processed, along those the DPR Wrapper IP containers, in our chosen MDE tool (Sodius MDWorkbench), to produce a multi-level IP library. The most upward library in the stack contains a set of IP templates which provide the user in UML MARTE with a modeling front-end in which information for interconnection and parameterization is readily available (promoting IP reuse); however, this information has been split into two diagrams for facilitating the task of a designer: the interconnection of the components to the bus, using component instances (and seen as black-boxes exposing the encompassing ports) is performed using a composite structural diagram, whereas the parameterization is carried out using a class diagram.

This approach follows the same philosophy of the IP-XACT design schema, in which ⟨*componentInstances*⟩ and their ⟨*interconnections*⟩ are divided into different sub-elements of a ⟨*design*⟩ description. Moreover, such an approach fosters the separation of concerns paradigm looked after by the Model-driven Engineering approach, in which different aspects of a system are clearly separated, either following a purely modeling (and thus model comprehension) reasoning or for facilitating the model transformations. This chapter does not intend to shed any light onto the UML MARTE extensions or packages to create the RecoMARTE profile, since these endeavours have been carried out by other partners in the FAMOUS project; however, more details can be found in [150].

Nonetheless, sufficient elements for understanding the advantages of the proposed approach will be demonstrated through the use of a couple of case studies in which we have created complete DPR platforms from UML MARTE. The main goal of the chapter is then to demonstrate how these UML MARTE platforms are modeled, concentrating our discussion in the hardware aspects of the methodology, since in this work we are mainly interested in providing the MDE-based FAMOUS approach with IP reuse and design by reuse capabilities. Therefore, our main goal is the creation of DPR systems from a component library with abstracted blocks and their use for composing a SoC platform (design by reuse), for constructing a set of models at the deployment level.

These models are generative in the sense that they contain enough information (due to their association to IP-XACT components containing rich metadata about the low-level components) for performing the model transformations described in detail in Chapter 6, namely, the MARTE to IP-XACT Design and the IP-XACT to MHS transformations. Furthermore, the allocation of the hardware modules in the application models onto the DPR Wrappers in the deployed architecture model, only outlined in the previous chapter, are

explained in more detail, and their role in the modeling process are highlighted. When discussing the different modeling aspects and how they relate to the Dynamic Partial Reconfiguration design flow, we will also discuss the advantages of using an MDE framework over the traditional purely HDL or Xilinx EDK-based approaches, both in terms of easiness of use, but also in terms of updatability and scalability, and more importantly, in how they can eventually speed up the design process.

The chapter ends with several implementation details. First, we discuss the hardware implementation of the proposed architectures, in terms of the consumed resources and the number of implementation components and files. These metrics will provide the user with a general picture of the complexity of creating a DPR system by hand, adding the additional effort of performing several repetitive and prone to error modifications, specifically during the design entry phase of the DPR design flow. Subsequently, we provide more information on the implemented model transformations, which where not detailed in the previous chapters, but more importantly, we discuss how the proposed approach speeds up the design process by providing several metrics (in particular, we discuss the design effort required to build the DPR platform, but also for configuring and generating the static and PRM netlists). In the discussion section, we analyze how this work is to be integrated with the rest of the FAMOUS framework, specifically regarding the application and reconfiguration control generation chains, but also with the reconfiguration services described in the previous chapters.

## 6.2/ Used DPR image processing architecture and implemented applications

In this section, we present a couple of system implementations to demonstrate how the hardware branch of a DPR system can be created using the IP reuse and design by reuse capabilities introduced in the second part of this thesis manuscript. First, we will introduce the base DPR system, in terms of the encompassing hardware components, and then we present two relatively simple applications, which have been developed to demonstrate the approach without compromising the comprehensibility of the modeling concepts discussed in a subsequent section. Both applications are based on image processing IP tasks, which have been designed in such a way that the reconfiguration process can be observed during the application execution.

### 6.2.1/ Utilized System-on-Chip architecture

In order to validate the correctness of the proposed approach, but also the feasibility of the FAMOUS framework as a whole, we made use of a common platform, depicted on Figure 6.1. The platform represents an on-demand DPR system, in which different Partially

Reconfigurable Regions can be reconfigured via external commands (i.e. user inputs provided by the hyperterminal and UART controller) and monitored by a soft-processor. The architecture is based on a MicroBlaze embedded soft-processor and it has been implemented in a set of platforms, targeting Virtex 5 and 6 FPGAs, but here we concentrate on the former. The introduction of an embedded processor in the DPR design responds to two reasons. Firstly, the DPR design flow produces the static configuration file and several partial bitstreams. The static design is programmed in the FPGA during the initialization of the board, but the partial bitstreams have to be loaded on run time and the inclusion of a processor enables the system with a mechanism to efficiently perform this task. Secondly, the processor facilitates the creation of systems in which the dynamic reconfiguration process is performed on-demand. This means that the user can program the processor to perform the reconfiguration under certain predefined circumstances.

In the proposed architecture, the static design consists of a MicroBlaze processor and its associated data and program memories (BRAM modules), connected to a series of IP peripherals via the PLB bus. These peripherals give support to several of the tasks of the system. We have divided the modules for clarity reasons; first, the processor, along the SystemACE and ICAP controllers, encompass the Partial Reconfiguration specific modules of the system. The System ACE controller manages the read operations from the non-volatile flash external memory, and charges the full configuration and the partial bitstreams (the static in the form of an ACE configuration file which is automatically loaded onto the FPGA at power-on). The ICAP controller module receives the partial bitstreams

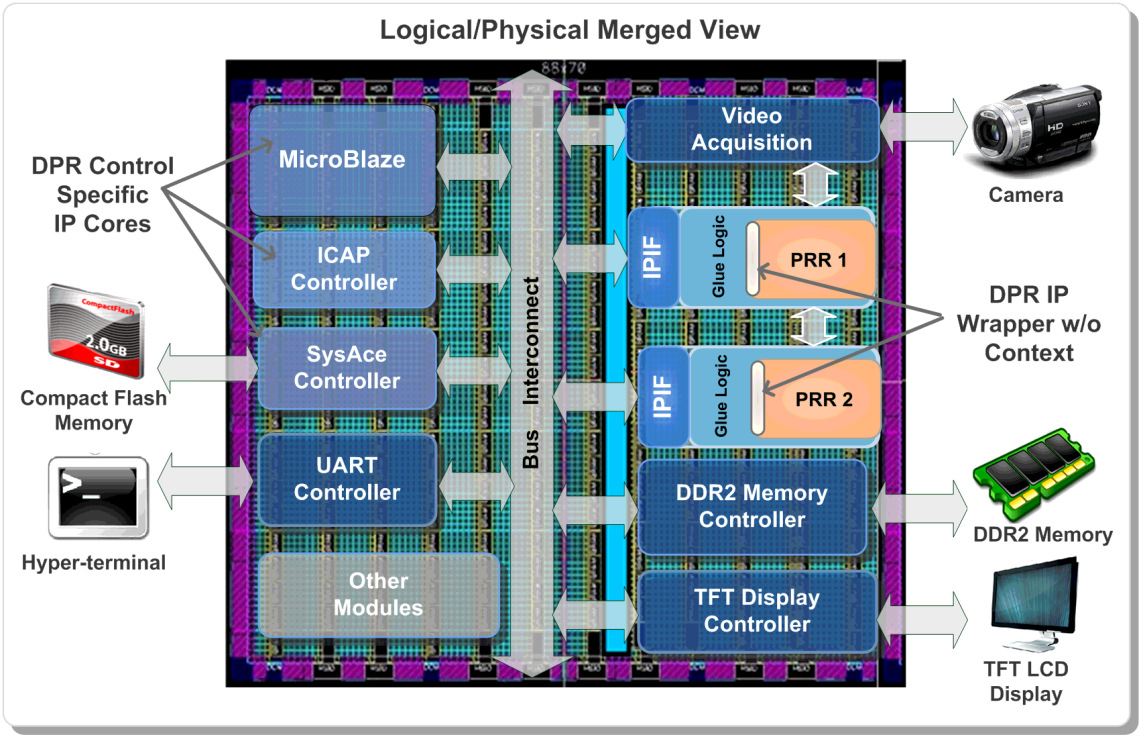


Figure 6.1: First Case Study, a simple image processing DPR system

(corresponding to each Partial Reconfiguration Module or PRM) from memory (via an IPIF wrapper) and uses this configuration data to modify the behaviour of the PRR by means of partial reconfiguration.

The modules in darker blue represent the set of static components used for supporting the proposed applications. The Multi-Port Memory Controller (MPMC) manages the read/write operations of an external DDR2 memory; it is in this memory where the modified partial bitstreams are stored to speed up the reconfiguration process, but also to store intermediate results in the video processing applications presented here, and from where the TFT controller module can read pixel information, gathered by the Video Acquisition IP, and then feed this pixel data to the external chip that managed the video output. As mentioned previously, an UART module provides a means of communication between the application running in the embedded processor and the user, via the hyper-terminal.

### 6.2.2/ TARGET APPLICATIONS

Having shown the proposed architecture, in this section we proceed to describe which reconfigurability scenarios have been implemented for validating the generation approach from UML MARTE. As discussed in the previous section, two Partially Reconfigurable Regions (PRRs) have been used in the architecture for this case study, in which a set of simple image processing IPs are mapped during the DPR floorplanning phase, and onto which the partial bitstreams implement the specified functionalities. We concentrate

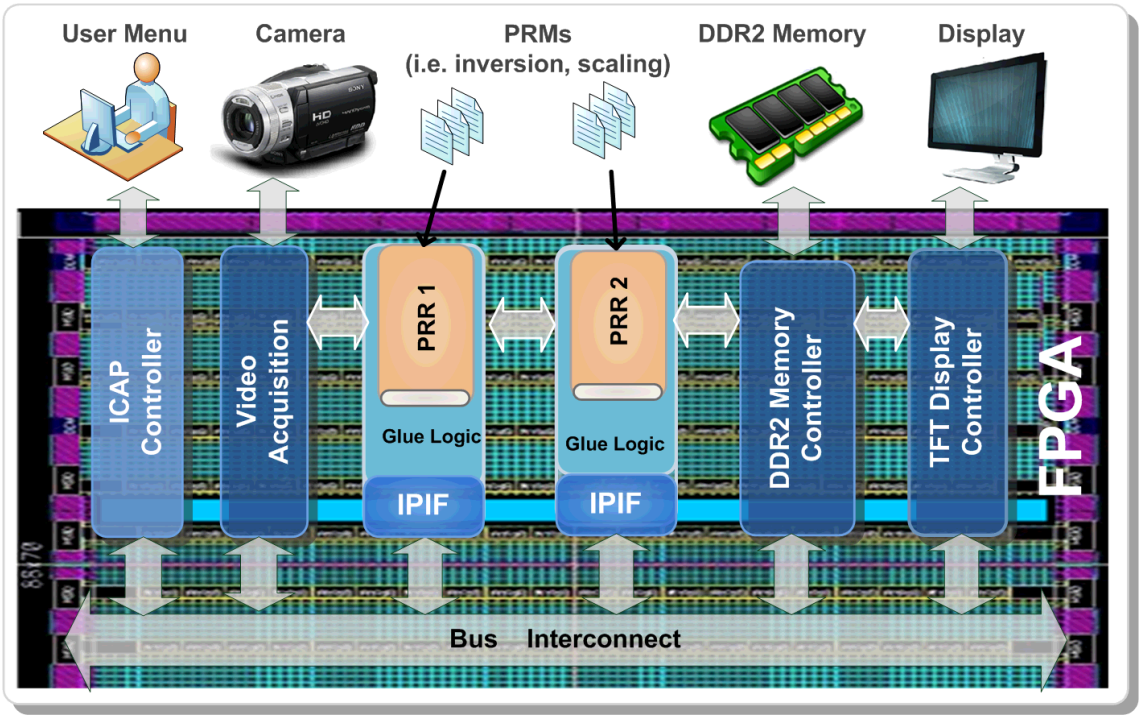


Figure 6.2: General scenario for the application: a reconfigurable video pipeline

this case study in video streaming applications, as depicted on Figure 6.2, in which the PRRs form part of a video pipeline between the image acquisition module and the TFT controller, passing by the MPMC, which manages the access to the memory to write the incoming video stream and the output video data to be displayed by the TFT controller. Furthermore, we have implemented a system in which the video camera is substituted by a set of images stored in the SD flash memory along the partial bitstreams, making for a more portable version. In the first case, the video image goes though the video pipelines, while the IPIF attachment are only used for setting registers in charge of fixing the height of the image allocated to each PRR; in the second case, the PRMs are fed with image data, which is written into the DDR after it has been processed, and finally, displayed in a continuous manner by the TFT controller.

As depicted on the figure, several bitstreams can be mapped to each of the reconfigurable regions; for the example presented in the rest of the chapter, the partially reconfigurable modules correspond to single pixel image processing operations (inversion, binarization, gray-scale reduction and pass through) which are loaded on the FPGA on-demand. In order to show the dynamicity of the application, each PRM processes only a part of the incoming image (for instance, half of the image, as depicted on Figure 6.3 a)): in this manner, we can show the reconfiguration process for one of the modules while the second maintains the same functionality. The partial bitstreams are loaded onto the FPGA via an external command provided by the user, which is presented with a menu describing which operations can be performed, and to which part of the incoming stream they can affect. A second architecture, not described in full details in this chapter, has been implemented: it basically carries out the same functionalities, with the added capability of readjusting the image resolution in parallel with the image processing operations.

The rest of the chapter demonstrates how the hardware component system (netlists before implementation) has been created from a set of UML MARTE models. We have purposely targeted a simple application (and architecture) to make it easier for the reader to understand the modeling concepts presented from here onwards.

Figure 6.3: Examples of implemented PRMs in the case study: a) Pixel image processing operators b) Scaling and image processing

## 6.3/ CASE STUDY: A DPR IMAGE PROCESSING ARCHITECTURE

In this section, we demonstrate how the system architecture introduced in the previous section has been modeled using UML MARTE. Then, we show how the hardware tasks to be allocated to the reconfigurable regions are used for describing the dynamic aspects of the application, and in general, how these models are then used as input to the proposed compilation chain to obtain the static top-level design on one hand, and the netlists for the PRMs on the other, in a parallel branch of the generation phase.

### 6.3.1/ MODELING OF THE APPLICATION AND ITS ALLOCATION ONTO THE ARCHITECTURE

The objective of an ⟨⟨*application*⟩⟩ model in UML MARTE is to define the functionalities (tasks) of a system, along the communication mechanisms used to express the overall system behaviour. In the particular context of this thesis work, we do not concentrate on the models of computation and the communication aspects of the application (which are described using separate models and metamodels), but purely in the allocation of hardware tasks into architecture, and on their subsequent deployment (IP reuse for system generation), in order to generate the input netlists of the Xilinx Partition Partial Reconfiguration flow. Therefore, in the discussion that follows, we concentrate on those modules that represent HW Task IPs to be executed by the DPR Wrappers. As discussed all along this manuscript, they represent functions to be used by the application dynamically, and at the lowest level, they are seen as partial bitstreams to be obtained from a set of synthesized PRMs, allocated to the Reconfigurable Partitions of the static architecture.

In fact, the application diagram created in this manner expresses all the possible configurations of a given module. For instance, a video distribution system could have multiple implementations of the front-end and back-ends, for processing various types of video inputs and display resolutions, respectively. Therefore, the application would require having two main placeholders, in which multiple implementations of the task are to be assigned using a different diagram, which combined with a state machine diagram, express the actual dynamicity of the application (a configuration diagram, controlled by a set of state machines, which are to be described in a subsequent section). In this sense, the reconfigurable tasks are analogous to the reconfigurable partitions (RP) to be defined in the top-level HDL code in the design entry phase, with the difference that the designer in UML MARTE is not concerned about the low-level implementation details: the blocks in the application model represent simple containers which can host different HW tasks seamlessly (due to the standardization process applied upon the PRM IPs). In the two examples presented here, we make use of two sets of image processing tasks, which are represented by the blocks Task1 and Task2 on Figure 6.4.

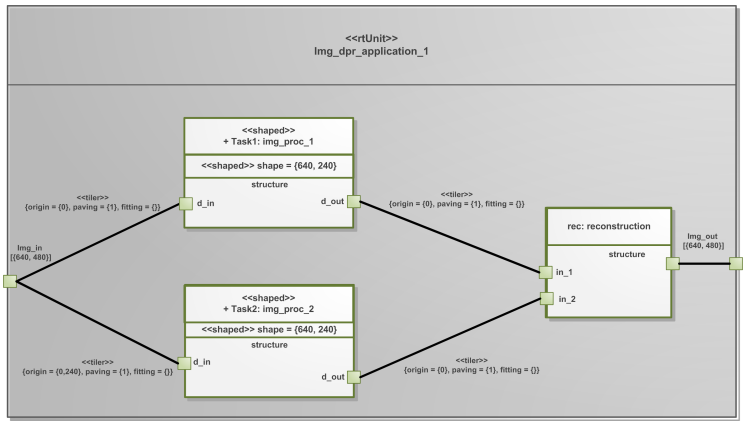


Figure 6.4: Modelling of the application using the ArrayOL model of computation

These blocks are to be allocated to DRP Wrappers in a subsequent stage of the FAMOUS design process but first, the logical architecture to express the construction of the system depicted on Figure 6.1, needs to be defined. This is done via the construction of an architecture diagram using a Composite Structure Diagram, as depicted on Figure 6.5; the difference of this platform description with the deployment description counterpart, examples of which have been used before, is that the logical architecture diagram represents the first step of a co-design methodology (using the Y-chart), before the association and deployment phases. Therefore, at this stage, the blocks in the model do not represent specific IP blocks or any particular back-end technology, but the intention of the designer in terms of elementary components such as processors, memories, and IO devices for communications, and in the case of our case study, video input and output devices. The DPR Wrappers, at this stage, are not yet explicitly defined as such, given that the tasks in the application can be implemented as software functions, and then allocated to processors; it is only during the deployed allocation phase (an extension to the MARTE profile proposed [150]) that tasks in the application diagram, allocated to processing units in the architecture diagram (pu1 and pu2 on Figure 6.5) are deployed (linked to their actual hardware implementations) as DPR Wrappers of certain kind (with or without context saving services). The rest of the components in the ⟨⟨*logicalArchitecture*⟩⟩, being static components, need merely to be linked to their actual implementation IPs, as it will be described in a subsequent section.

### 6.3.2/ Modeling of the architecture at the deployment level

The deployed allocation level contains a set of views for executable models generation; in this thesis we deal with the generation of the logical/architectural netlist of the top-level of a given platform specification, and of the IP descriptions that compose it (static and dynamically reconfigurable). As mentioned before, the objective is the generation of the structural and functional information that is used as input for the DPR design flow. Thus,

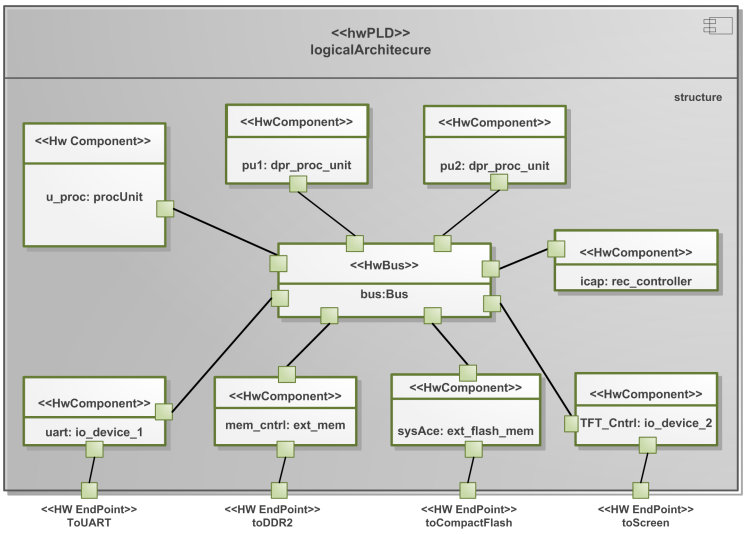


Figure 6.5: Modeling of the architecture before the deployed allocation phase

Figure 6.6 shows the modeling phase of a reconfigurable SoC platform, targeted to an embedded architecture to be exploited by the Xilinx Platform Studio tool. This diagram represents a ⟨⟨*deployed*⟩⟩ logical architecture where each elementary component (gathered from the IP library and instantiated into the diagram) is labeled as ⟨⟨*HwComponent*⟩⟩ and ⟨⟨*deployed*⟩⟩; the first one is used to define the component as hardware module from MARTE HRM package, while ⟨⟨*deployed*⟩⟩ stereotype belongs the Deploy package, and implies that each elementary component is to be associated with a component class in the *Parameterization view* for carrying out its configuration.

The ⟨⟨*deployed*⟩⟩ stereotype functions in fact as a means to enable the FAMOUS methodology with the requirements of the deployment level discussed extensively in Chapters 2 and 5 (associating the IP with properties, Component re-utilization, abstraction of the associated functionality, and system composition and generation). As discussed in the latter chapter, this is attained by linking the UML MARTE templates to IP-XACT components, which contain rich information about the IP (for parameterization, re-use, interconnection and design by reuse).

The diagram depicted on Figure 6.5 the so-called *Platform View*; using a CSD, the designer is interested in describing the way the system is to be connected, not concerned to the low level aspects of the design. However, and as described in Chapter 5, each of the ports in the elementary component corresponds to a port, bus interface or IO interface in the underlying IP-XACT and MPD descriptions; therefore, the ports carry name and port type information for distinguishing among the types of connections, information which is subsequently used for transformation purposes. Similarly, the connection between ports (using UML connectors) is to be associated with ⟨*interconnections*⟩ and ⟨*adHocConnections*⟩ in the generated IP-XACT design description, just as described in the previous chapter.

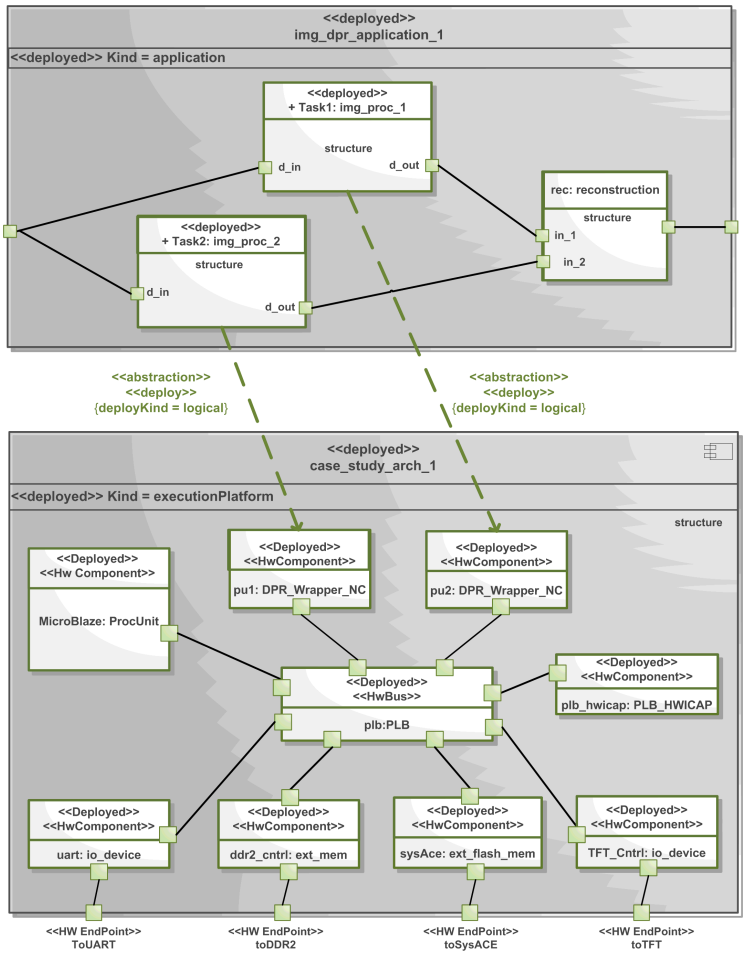


Figure 6.6: Allocation of the application onto the deployed architecture

The elementary components on the figure correspond with those in the schematic system description of Figure 6.6. The DPR Wrappers (labelled on the figure as pu1 and pu2, standing for processing units) need to be associated with tasks in the ⟨⟨*application*⟩⟩ model, showing the different configurations (or modes) that they can implement. However, as with all the other components in the ⟨⟨*deployedArchitecture*⟩⟩ model, they are converted to simple component instances in the IP-XACT design description, and propagated in the transformation chain (e.g. MHS and top-level HDL description), as static components.

Regarding the correctness of the so-obtained platform description, this is ensured by the fact that the port descriptions contain information about the name and type of the interface (i.e. PLB, VGA), so checkers can be implemented to verify the consistency of the interconnection performed by the user before proceeding to the generation of the back-end models; the same applies for the external ports (labelled as ⟨*HwEndPoints*⟩ on the figure) which are assigned to IO interfaces in the hierarchical component associated with the design description, and which corresponds to the encompassing ports of the static top-level system description (and thus assigned to external FPGA pins).

### 6.3.3/ ASSIGNING CONFIGURATIONS TO THE RECONFIGURABLE PARTITIONS

Once the model for the ⟨⟨*deployedArchitecture*⟩⟩ has been created from the IP library, the next step is to actually define which tasks in the ⟨⟨*application*⟩⟩ are to be allocated onto the DPR Wrappers (processing units in UML MARTE jargon). As described in Chapter 4, the DPR wrappers have been architected in such a way that they contain the Reconfigurable Partitions (RP) to accommodate the different configurations of the Partially Reconfigurable Modules (PRMs); the parameters information of the PRMs customize the RP internal interfaces through dependent parameters. This process is analogous to the PRM allocation in the design entry phase: however, unlike the typical DPR design flow, in which this step is completely manual, in the case of our methodology, this allocation is entirely specified in a model and derived automatically though model transformations. Furthermore, in the typical design flow, the task of manually modifying the PRMs to have the same interface of the RP is carried manually in an individual basis, as well as the synthesis of each PRM, for which a Xilinx ISE project must be created. In order to exemplify how the FAMOUS approach eases this task, let us consider the simple image processing IPs allocated to a non-context aware DPR Wrapper depicted on Figure 6.7 a); the DPR wrapper can hold different PRMs (although we have only chosen two for simplicity reasons) as depicted on Figure 6.7 b): image binarization and inversion.

The architecture of our example contains two DPR wrappers, which can hold both configurations of the IP for treating a part of the incoming image. Therefore, the chosen tasks must be ⟨⟨*deployed*⟩⟩ individually onto each of the wrappers using a separate ⟨⟨*configuration*⟩⟩ MARTE block, as depicted on Figure 6.8. Each ⟨⟨*configuration*⟩⟩ block corresponds to the mode of a reconfigurable region (the physical counterparts of the reconfigurable partitions when the system is implemented and running on the FPGA) defined in the ⟨⟨*application*⟩⟩ model; as can be seen on the figure, the type of allocation (as discussed before, a combined deployed allocation) is defined as *logical*, using the

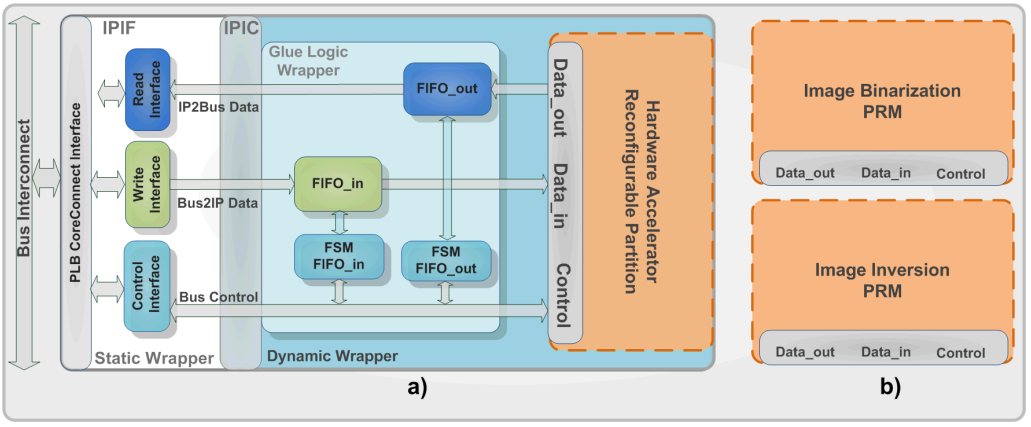


Figure 6.7: a) DPR Wrapper IP containing the customizable reconfigurable partition. b) The different task IPs allocated to each partition (binarization, inversion)

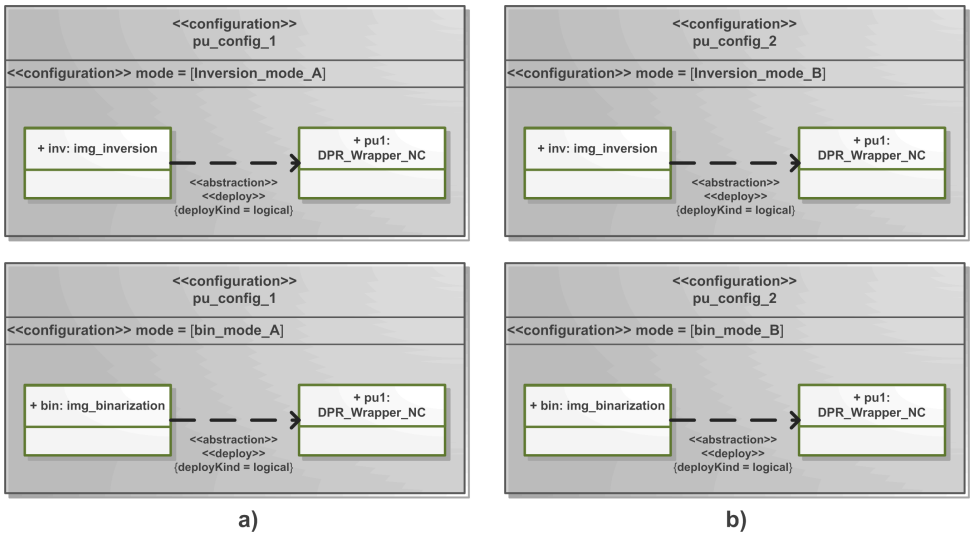


Figure 6.8: Examples of two possible configurations for the pair of reconfigurable wrappers defined in the deployed diagram

⟨⟨*deployKind*⟩⟩ attribute, which unequivocally specifies the allocation of a task in the application onto a component in the ⟨⟨*logicalArchitecture*⟩⟩ model. This attribute is used in order to differentiate it from the physical deployment of a DPR wrapper onto a reconfigurable region of the FPGA that will be described shortly after.

The ⟨⟨*modes*⟩⟩ of the reconfigurable wrappers are controlled by state machines defining the application behavior, which are described using a different model, as depicted on Figure 6.9. These state machines are used, along other models, to obtain a correct by construction Reconfiguration Controller, another axis of the FAMOUS project, as described in [151].

The use of different modeling diagrams for the configurations and the mode behaviour has several advantages. By independently modifying those models, completely new applications can be created from the same architecture, a feature the many DPR methodologies foster for scenarios in which hardware tasks virtualization is a necessity. In this manner, the co-design of DPR platform is significantly leveraged, even if new static modules are

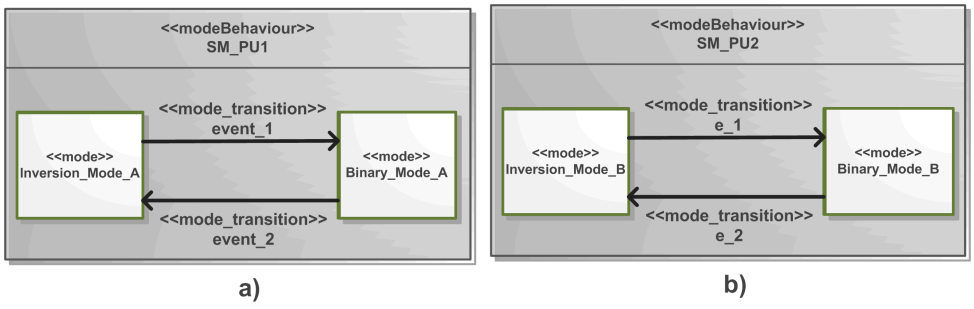


Figure 6.9: Example of the state machines for the two reconfigurable partitions/areas

added to the static architecture; if new DPR wrappers are added to the system, the modularization nature of the configurations and state machine models enables to easily update and scale the application, and the model transformations remain unchanged. Therefore, the remaining work consists on re-defining the application code to run on the processor, a task that can be greatly simplified by using a component-based approach in tandem with an operating system.

### 6.3.4/ PARAMETERIZATION OF THE APPLICATION HARDWARE IP BLOCKS

As described in the previous sections, each ⟨⟨*deployed*⟩⟩ component, both in the ⟨⟨*logicalArchitecture*⟩⟩ and in the ⟨⟨*configurations*⟩⟩ models corresponds to a class instances in the *Parameterization view*. As discussed in Chapter 5, the UML MARTE components have been classified into two groups, by using the ⟨⟨*deployedEndKind*⟩⟩ attribute in the deployment package extension in the RecoMARTE profile; therefore, hardware components in the ⟨⟨*configurations*⟩⟩ model become ⟨⟨*deployed*⟩⟩ components with the {kind=application attribute} in the *Parameterization view* , whereas those in the ⟨⟨*logicalArchitecture*⟩⟩ are associated with components whose attribute is {kind=executionPlatform}. This classification has important implications in the FAMOUS chain generation, given that they are processed in a different manner to produce the PRM netlists and the top-level description, respectively.

As an example, Figure 6.10 shows a snapshot of the *Parameterization view* for the reconfigurable modules (HW IP tasks, as defined in our IP taxonomy); it must be noted that each PRM might need to be configured independently, but in certain applications (such as large scalable systems) in which several copies of the same reconfigurable are used, the model could be compacted by using OCL constraints. However, this discussion is outside of the scope of this thesis, since it corresponds more to modeling strategies in UML MARTE, and not directly to the hardware generation of the PRMs. Nonetheless, if the same PRM is to be used for several DPR wrappers, the consistency in the interfaces

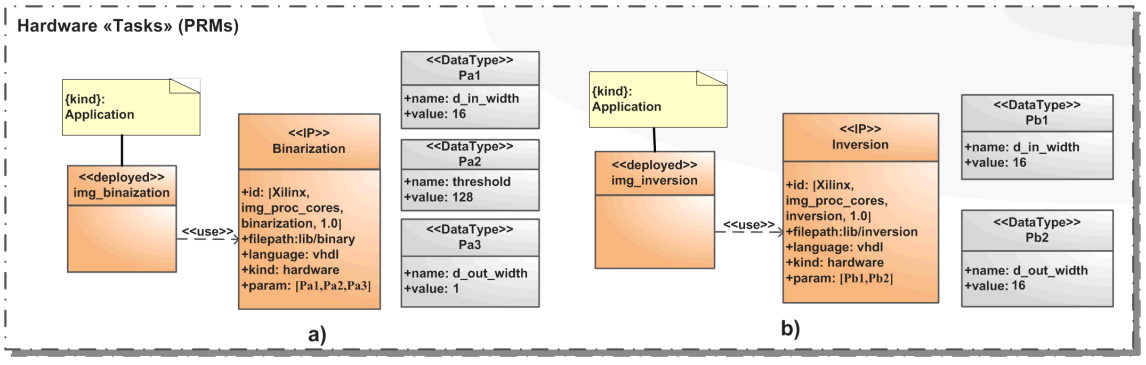


Figure 6.10: A snapshot of the parameterization view containing the configuration for the PMRs to be assigned to one of the DPR Wrappers

enables to map the same netlist to multiple Partially Reconfigurable Regions (e.g. for applications such as bitstream relocation or in hardware virtualization).

As described in Chapter 2, PRM modules usually require to be parameterized to meet the requirements of the application. This configuration can be performed automatically or manually by the designer; for instance, the threshold of the binarization IP module can be set to different values, as well as the data_in_width. The data_out_width, however, does not need to be configured, given that a binarization modules produces an output of only one bit; nonetheless, the information of the interface needs to be present in the model given that is to be used to configure the interfaces of the DPR Wrappers, as discussed in the previous chapter. Regarding the inversion IP module, no configuration parameters are defined apart from those of the interfaces; as described in Chapter 5, the ⟨⟨*parameters*⟩⟩ to be configured by the user are tagged in the IP-XACT component description, and imported on the UML MARTE template when defined as ⟨*user*⟩ attribute under the ⟨*resolve*⟩ element of the ⟨*parameter*⟩ description.

The second role of the deployment of the PRM modules is associating the abstract blocks with the underlying HDL implementations. This is seamlessly achieved by first linking the UML MARTE template with an IP-XACT component description of the IP implementation. As discussed in Chapter 5, this approach has many advantages for building SoC platforms, but in the case of the PRMs, this binding has a different purpose, since the reconfigurable modules are not in fact directly integrated in the platform. Nonetheless, the IP-XACT component descriptions of the PRMs, which are retrieved by associating the ⟨⟨*deployed*⟩⟩ MARTE component with an IP-XACT ⟨⟨*vlnv*⟩⟩ value, are used for two purposes: automatically retrieving the interface information for configuring the RP interface and executing the synthesis procedures on the HDL files specified in the IP-XACT component description. In this manner, all the netlists for the PRMs can be automatically

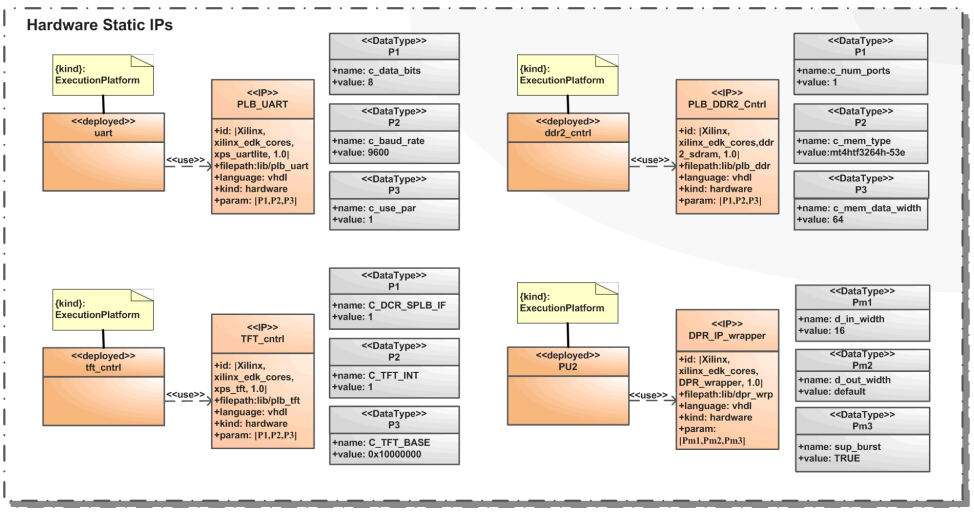


Figure 6.11: A snapshot of the parameterization view containing the configuration for several of the static IP blocks in the platform

obtained from the UML MARTE modules, without dealing directly with the Xilinx tools.

### 6.3.5/ PARAMETERIZATION OF THE STATIC PLATFORM HARDWARE IP BLOCKS

The discussion in the previous section for the reconfigurable modules applies in a very similar vein for those deployed components whose attribute is {kind=executionPlatform}. The main difference lies in their role in the FAMOUS flow, as described before: the parameterization information is used for generating the IP-XACT ⟨*design*⟩ description, in which they are transformed into ⟨*configurableElements*⟩ under each ⟨*componentInstance*⟩. In this context, their role is in many instances that one of controlling configuration attributes for the inclusion of dependent parameters, ports and interfaces in the underlying MHS, and therefore, top-level HDL description of the static platform. Many of the components in the static platform need to be configured, while others (such as the ICAP component) are used as black-boxes hardware-wise. Figure 6.11 shows some of the static components of the implemented DPR image processing architecture, namely the UART component, and the DDR2 and TFT controllers, along one of the DPR Wrappers.

### 6.3.6/ GENERATION OF THE XILINX PLATFORM STUDIO DESIGN DESCRIPTION

The ⟨⟨*deployed*⟩⟩ components in those diagrams are used, as described in Chapter 6, for obtaining the IP-XACT design description, from which the top-level static platform is obtained. For reasons explained in detail in previous chapters, we target the creation of the Xilinx Platform Studio system hardware specification, encoded in the MHS file, which is obtained from an IP-XACT to MHS model-to-text transformation. The so-obtained MHS file is used by MDWorkbench, along a set of scripts for generating a system which can be

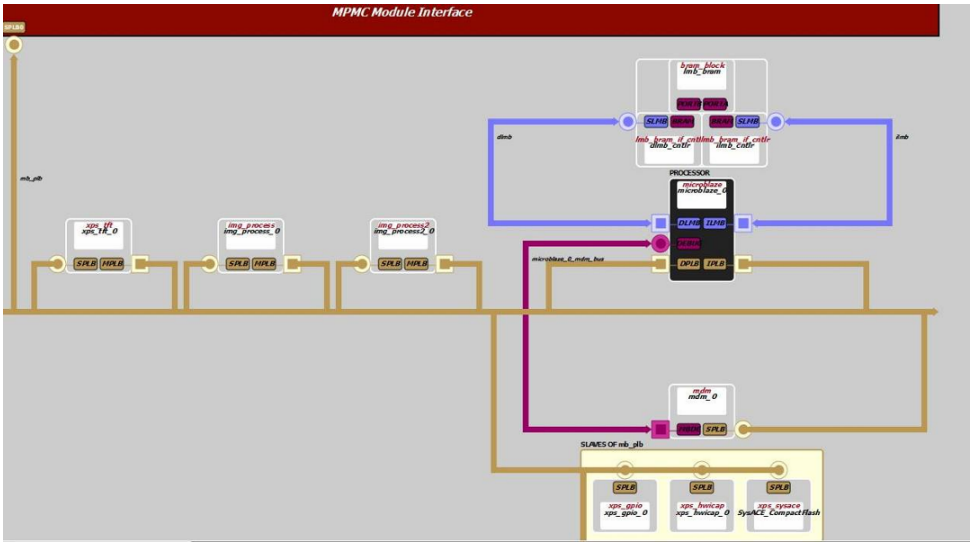


Figure 6.12: A snapshot of the obtained Xilinx Platform Studio SoC description

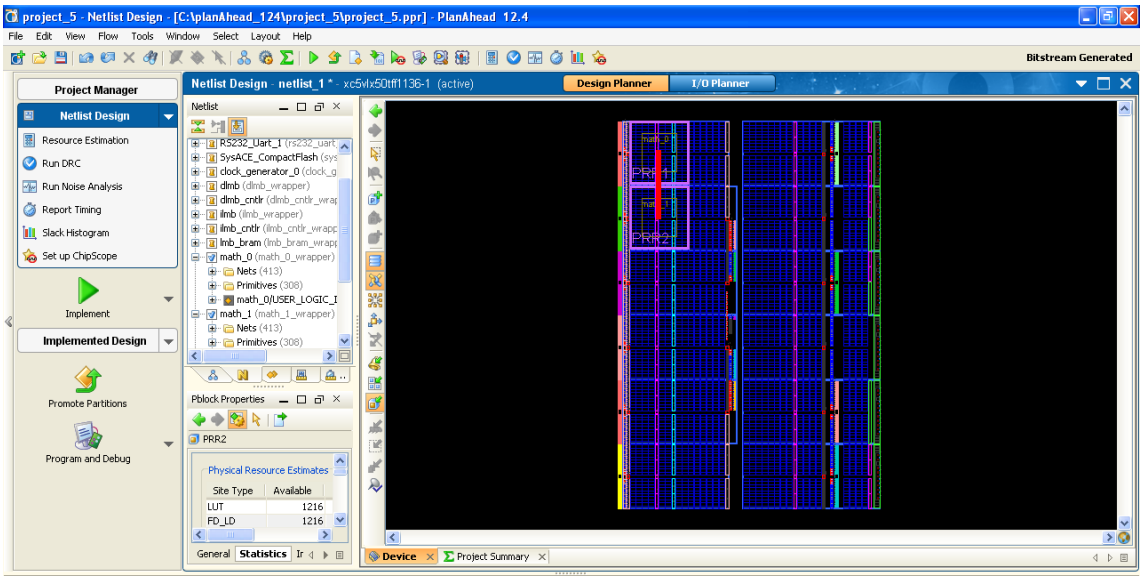


Figure 6.13: Floorplanning phase of the proposed case study

used in Xilinx EDK for creating the application code and for testing purposes, as depicted on Figure 6.12. As with any other design approach, this intermediate hardware platform can be used for testing the non-DPR functionalities of the system, or alternatively, a non-DPR system containing the intended functionalities can be created first, adding subsequently the DPR wrappers for generating the system with the Reconfigurable Partitions which can be synthesized, but not placed and routed by the conventional FPGA design flow.

Once the non-DPR system has been validated, the synthesis procedures can be executed for generating the netlists to be fed to PlanAhead for implementing the DPR system and obtaining the total and partial bitstreams for configuring the FPGA. Figure 6.13 shows the floorplanning phase of the system in PlanAhead.

## 6.4/ Generation and System Implementation Results

In this section we embark in a succinct discussion of the implementation details of the compilation chain used to generate the top-level and DPR netlists for the design entry of the DPR flow from the UML MARTE models described in the previous section. First, we describe how we have created the different models (e.g. used tools and golden models) and then briefly describe the transformations rules. Then, we analyze the use of the compilation chain in the context of the design entry phase of the DPR design flow, and how can facilitate the creation of such systems for non-experts, or simply by abstracting the use of several complex tools away from the designer at high-levels of abstraction. Furthermore, we discuss some preliminary experiences in design effort, mostly in a qualitative

| Used Languages | Purpose |
|---|---|
| **MQL** | Ruleset definition |
| **Java** | Transformations |
| **IP-XACT** | Intermediate Model |
| **VHDL** | IP Implementations |
| **Used Tools** | |
| **Papyrus** | Modelling |
| **Rhapsody** | EDK Meta-models |
| **MDWorkbench** | Model Transformations |
| **IP-XACT Editor** | Golden Rule Models |
| **Xilinx EDK** | DPR SoC implementation |
| **Xilinx ISE** | IP development |

a)

| Transformation | Nb of Lines | Purpose |
|---|---|---|
| **MPD to IPXACT** | 113 | Obtain IP-XACT library from Xilinx EDK This transformation requires all IPs to be wrapped by the CoreConnect IPIF module |
| **IPXACT component to MARTE** | 98 | Obtain MARTE Library templates which can be used for creating high-level platform |
| **MARTE model to IP-XACT design** | 150 | An standard IP-XACT design object XML-IR is obtained from the UML MARTE XMI-IR |
| **IPXACT to MHS** | 87 | Generate top-level Xilinx EDK description used by Xilinx PlatGen netlister and sythesis |
| **IPXACT to MSS** | 47 | Generate software drivers description of IPs |

b)

Figure 6.14: Modeling tools and languages used in the hardware branch of the FAMOUS framework b) Transformation chain implementation details

manner, given that we do not count with many design examples to carry out an extensive study in this regard. Finally, we provide implementations results of the proposed architecture and of an extension and we outline future work.

### 6.4.1/ IMPLEMENTATION OF THE TRANSFORMATION RULES AND USED TOOLS

As depicted on Figure 6.14 a), the different models in the FAMOUS framework were specified using the Papyrus Modeling Framework, whereas the Xilinx Platform Studio metamodels where created using Rhapsody and then imported to MDWokbench. The MDWorkbench model-driven platform has been specifically designed to support the creation of meta-models from various formats, and includes several post-processing tools to improve the Ecore formalization of non-ecore meta-models. We make use of the Sodius MDWorkbench as a means of developing meta-models specifications for the different models in our design chain. Furthermore, the tool also provides means to describe transformation rules and to perform model transformations; this implies the use of the tool as a backbone for federating the heterogeneous data manipulated in our design flow.

As discussed in Chapters 2 and 5, we use IP-XACT as a means to share the same information between all the actors, using a common way to describe this information, and to automate the generation of multiple formats depending on the task needs and to perform checks between steps. We also promote IP reuse by providing IP descriptions that remain interchangeable regardless of the added vendor extensions. The IP-XACT specifications provide a set of XML schemas (.xsd) for representing different concepts in SoC design. This set of XML schemas has been processed by the improved XSD/Ecore meta-model importer in MDWorkbench, which leads to a Java/EMF implementation of the IP-XACT meta-model. An example, Figure 6.15 shows a snapshot of the design meta-model in MDWorkbench.

Before performing the MPD to IP-XACT and IP-XACT to UML MARTE model transformations, a set of IP-XACT components where created using the Eclipse IP-XACT editor, which helped us in creating a set of golden IP-XACT models for testing the validity of

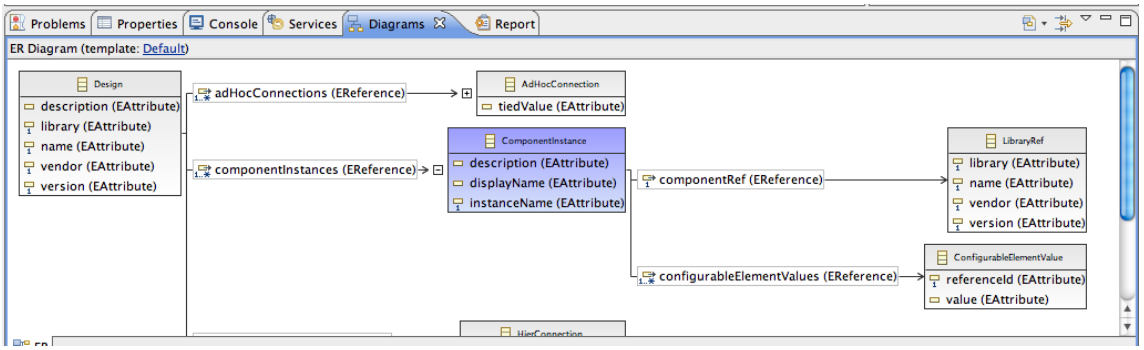


Figure 6.15: Snapshopt of the design meta-model in Sodius MDWorkbench

the proposed transformations. The IP-XACT editor also provided support for creating IP-XACT designs, so we could also test the design by reuse branch of the flow, and therefore, the UML to IP-XACT model transformations. The MHS files were first obtained by creating several simple Xilinx Platform Studio designs; then, the same systems have been created using the proposed UML MARTE composition models, and fed to Sodius MDWorkbench to obtain the corresponding MHS file, which it is used for creating a XSP project from scratch by running some scripts.

The rules sets for all the transformation were specified using the Model Query Language (MQL). The transformations were implemented in Sodius Workbench using Java, Figure 6.14 b) shows a summary of the implemented transformations, the number of code lines taken per each one and the purpose in the flow.

The trasformations have been defined as a set of services which can take UML MARTE and IP-XACT designs as inputs (in the case that the latter was avaiable from another party) and to produce the MHS files, as depicted on Figure 6.17. These services can of course be defined as a compilation chain, in which the input is a UML MARTE model and the ouput is a complete Xilinx Platform Studio project contaning an VHDL sub-folder and a netlist sub-directory. These files are obtained from scripts provided by Xilinx tools

| File Type | Nb of Lines | Exec. time | Description |
|---|---|---|---|
| ***XMI*** | 654 | 50 sec | MARTE model in XMI for transformation purposes into IP-XACT in MDE Workbench |
| **IP-XACT *Design XML*** | 998 | 30 sec | Intermmediate model used in the transformation phase to obtain the MHS file |
| **MHS File Top Level** | 455 | 20 sec | This file is used by EDK to instantiating the IP blocks, parameterisation and interconnection |
| **VHDL Top Level** | 7986 | 30 sec | HDL description of the system. Obtained by feeding Xilinx Platgen with the MHS and MPD files |
| **Netlist Top Level** | N/A | 12 min | The system VHDL top level file is used to obtain a NGC for the static part of the platform |

Figure 6.16: Lines and execution times per transformation

and could enable the generation of a complete non-DPR platform from UML MARTE to the total bitstream generation. This can be useful, for instance, when a non-DPR system needs to be created for validating some static functionalities, and then to produce the DPR top-level netlist with the corresponding black-boxes.

### 6.4.2/ THE PLATFORM TRANSFORMATION CHAIN AND ITS BENEFITS OVER THE TRADITIONAL DESIGN FLOW

In this sub-section we elaborate on the design effort required to implement the system detailed in Figure 6.1, especially if we compare it with a purely VHDL approach and, as in the case of the generation back end of this methodology, using Xilinx Platform Studio. Let us consider for instance the obtained VHDL top level design, which is generated in around 30 seconds by PlatGen, as depicted on Figure 6.16; the top-level VHDL description contains 7986 lines of code, and mainly contains components instantiation, parameterization and signals declarations for interconnection. The system presented in this case study is relatively simple and yet, its creation would represent a sheer amount of work if had to be performed manually; in contrast, by using the proposed approach, the system designer in UML MARTE can create the platform in a much more straightforward manner, and obtain a synthesizable RTL description automatically. The figure below presents the number of lines per each intermediate system representation (in order to provide a glimpse of the complexity of the models), and the amount of time required to obtain the top-level netlists (the PRMs netlist generation time is not detailed, but represents a fraction of the total time).

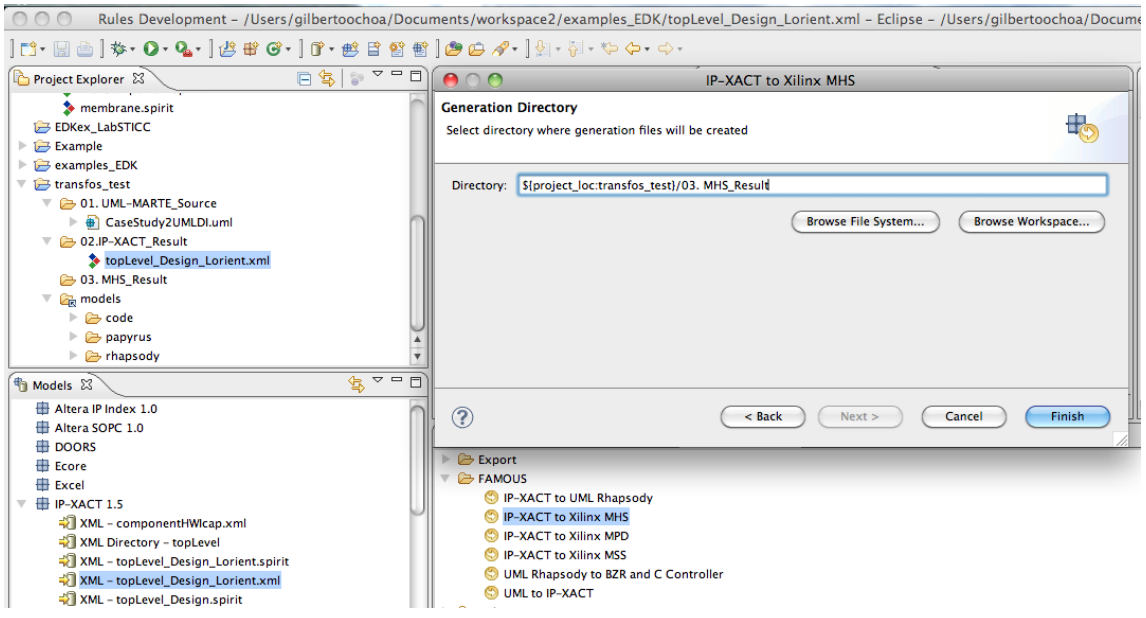


Figure 6.17: Example of the compilation chain in Sodius MDWorkbench

| Type of design capture | Time Used | Description |
|---|---|---|
| ***Pure VHDL Approach*** | | |
| Manually integrating the platform | Days | This method is the less reliable, long and prone to error. Good for small systems |
| Manually modifying DPR Ips | Hours | Not support for DPR management |
| ***Using Xilinx EDK*** | | |
| Platform Integration in XPS | Hours | EDK is justitfiable for systems containing at least one processor (DPR manager) |
| Manually modifying DPR IP+ Synthesis | Hours | IP blocks need to be processed separately |
| ***Proposed Approach*** | | |
| Platform Integration | Hours | Platform creation simpler and more visual than using EDK, description independant from back-end |
| Modifying DPR + Synthesis (automated through IP-XACT) | Mins | time, but integration is improved DPR system and IP syntheis is automated |

Figure 6.18: Design efforts using VHDL, EDK and our approach

Regarding the actual design effort to create the platform, it is evident that manually instantiating and integrating the components of such a design (composed dozens of components, and multiple sub-components) would take not only hours, but days, in a very prone to errors process, as depicted in Figure 6.18. Xilinx XPS, using the Platform Specification Format (PFS, notably MHS and MPD files), makes the design process more amenable: the designer can start creating a design through an easy to use Graphical User Interface (GUI), and then parameterize the design by choosing different options through IP specific TCL files and GUIs. These changes are automatically updated in the MHS files by parsing the corresponding MPD file and checking for any dependencies on parameters. However, the creation of the platform in XPS is not completely automated, and a lot of steps still need to be performed manually; for instance, importing IPs into the platform, their interconnection and parameterization.

All these steps require a great deal of design effort and expertise of the tools and this is precisely one of the advantages of used the proposed MDE methodology: by using a high-level description, the designer does not to know all the specifics of the used tools, which often are difficult to grasp by people who are not proficient into FPGA design and VHDL. The DPR aspects of the flow further complicate the proceedings, since more tools need to be used for generating the DPR design. For instance, the design has to be generated in XPS, with black boxes for the reconfigurable modules (PRMs); the PRMs need to be synthesized as independent projects, in Xilinx ISE, and then imported along the top level into PlanAhead.

A comparison of the generation PRM netlists is also provided on the figure. The typical design approach would require manually inserting (and configuring) the black boxes in the reconfigurable partitions of the PRM HDL descriptions, and in parallel, to synthesize each of the IPs to be mapped in these reconfigurable partitions; the same applies for an EDK based approach. If we consider the number of files to be integrated, as shown on the table, it can be observed how rapidly the design effort can explode. In our approach,

these descriptions are available in the library, and their synthesis is automated through TCL scripts that access the IP-XACT components description, and stores the netlists in a new project folder. Along with the top-level netlist, they comprise the necessary inputs for the DPR floorplanning phase.

Further advantages of using UML and MARTE is the maintainability and improved updatability of the models; this means that, contrarily to purely VHDL or EDK flows, a change in the platform requires much less effort: since every step of the design flow is automated, the designer does not even need to make use Xilinx EDK or ISE. The IP-XACT descriptions also facilitate the updatability of the approach by changing the vendor extensions or the target meta-models, but not the implementation files. It must be noted that we consider design capture times by non experts.

In Figure 6.19 we provide a summary of the number of files used for the implementation of the platform, note that an IP block can be composed by a few or even hundreds (260 VHDL files for the Multi-Port Memory Controller) of implementation files. The figure shows the implementation details for the system originally presented in the case study, but also for its extension outlined in the last section of the first section of this chapter. Here, we would like to emphasize that the capability of linking the UML MARTE with IP-XACT components which contain the complete implementation information, significantly reducinh information clutter in the deployment models: previous approaches would write the complete file dependency path into a comment, which in our opinion is not very efficient, since it leads to poor IP reuse strategies and makes it more difficult for the designer to adapt the models to new IPs or changing environments.

### 6.4.3/ Implementation results in an actual FPGA platform

In this sub-section we briefly discuss the implementation details of the case study, both in its original and enlarged versions. Both architectures were implemented in Virtex 5 FPGAs (ML501 and ML505 platforms). The implementations results are shown on Figures 6.20 a) -c). The first table shows the implementation results for the static components

| File type | Number of files per system | |
|---|---|---|
| | Architecture 1 | Architecture 2 |
| **Comp IP-XACT** | 21 | 24 |
| **MPD** | 21 | 24 |
| **VHDL** | 21 (top-level IP)<br>700 (sub-components)<br>2 DPR wrappers w/o cntxt support | 24 (top-level IP)<br>1200 (sub-components)<br>2 DPR wrappers w/o contxt support |
| **Netlist** | 1 Top Level<br>2 DPR Modules (x4) | 1 Top Level<br>2 DPR Modules (x3) |

Figure 6.19: Number of IP-XACT, MPD and VHDL files used per architecture

**a) Static system implementation**

| Architecture | Resources Utilization | | |
|---|---|---|---|
| **Module** | **LUT** | **FF** | **BRAM** |
| Microblaze | 1445 | 1518 | 0 |
| BRAM Ctntrl | 0 | 0 | 16 |
| MB_PLB | 182 | 854 | 0 |
| RS-232 | 160 | 142 | 0 |
| DDR2 Cntrl | 3698 | 2680 | 5 |
| SysACE Cntrl | 217 | 103 | 0 |
| TFT Cntrl | 767 | 619 | 1 |
| HWICAP Cntrl | 520 | 635 | 0 |
| **System** | **9592** | **8688** | **22** |

**b) Reconfiguration Metrics for Example 1**

| Example 1 | Resources Utilization | | | | Bitstream Metrics | |
|---|---|---|---|---|---|---|
| DPR Module | LUT | FF | DSP | RAM | KB Size | Conf Time |
| IMG Proc 1 | 1008 | 847 | 0 | 0 | 5 KB | 0.05 ms |
| IMG Proc 2 | 1008 | 847 | 0 | 0 | 5 KB | 0.05 ms |

| Example 2 | Resources Utilization | | | | Bitstream Metrics | |
|---|---|---|---|---|---|---|
| DPR Module | LUT | FF | DSP | RAM | KB Size | Conf Time |
| Filter | 1419 | 1636 | 8 | 8 | 47 KB | 0.47 ms |
| Rescale | 940 | 389 | 0 | 4 | 44KB | 0.44 ms |

**c) Reconfiguration Metrics for Example 2**

Figure 6.20: Implementation results for: a) Static architecture. b) PRMs for example 1. c) PRMs for example 2

of the architectures, which are very much the same, apart from the video acquisition IP, which has not been used in the first example.

The Table b) shows the implementation results for the inversion/binarization IP cores, which consume the same resources due to the simplicity of the performed operations. The second part of the table (Bitstream metrics) shows the results of the resulting bitstreams, which are identical due to the size of the PRR. The same applies for the PRMs for the second application detailed on Table c). The two applications are shown on Figures 6.21 a) and b).

## 6.5/ Discussion and Conclusions

In this chapter, we have presented a case study that demonstrates how the proposed IP reuse and design by reuse methodology is exploited for composing DPR SoC platforms at high-levels of abstraction, and then generating the inputs of the Xilinx Partition Partial Reconfiguration Design flow (or the outputs of the design entry phase in the form of

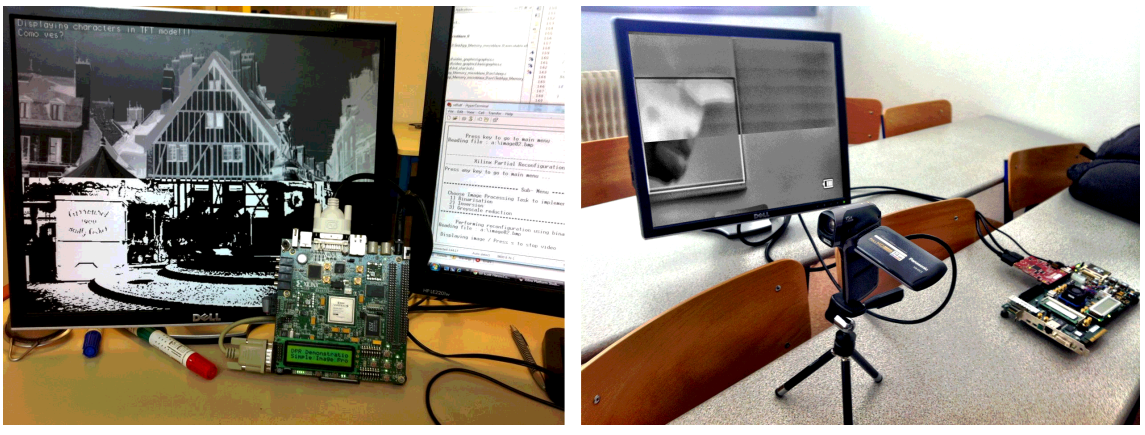

Figure 6.21: a) Implementation with static image b) Implementation with video input

netlists for the static and PRM modules). The chosen application is simple enough to understand the different modeling requirements and complex enough to be scalable and adaptable to different applications, as demonstrated by the implementation of a second system.

We must emphasize that we have not showed an example with context saving capabilities due to the fact that the such application is still under development; nonetheless, in this thesis we focus on the hardware aspects of the FAMOUS methodology, and the modeling concepts presented in this chapter apply as well to the creation of the hardware platform of context-aware DPR systems. This is possible again thanks to the use of a component-based approach, in which the DPR Wrappers with context saving capabilities become mere black-boxes onto which context-ready HW IP tasks in the application model can be allocated. Thus, the creation of the hardware platform for such a system would follow the same guidelines described in this chapter, but other model would be required to model an application with context saving (and possibly, operating system) capabilities.

The case study has also enabled us to engage in a thorough discussion of the required design effort when creating a DPR platform, using a purely VHDL approach, Xilinx Platform Studio, and the proposed methodology. We have shown that, even if the proposed framework does not significantly speed up the composition and parameterization of DPR SoC platforms, it facilitates the design entry phase of the flow (as other aspects of the FAMOUS framework aim at facilitating the configuration control, and the physical floor-planning, among others). This is due to the fact that a modeling front-end facilitates the integration and configuration of the encompassing static and DPR IPs, which along with the generation capabilities provided by the tools (i.e. MDWorkbench) enables to move from the initial specification to the SoC platform generation while avoiding many low-level details, including the necessary tools for generating the netlists.

However, as discussed before, the MCF approach proposed here belongs to an MDE-based framework, which fosters methodologies in which models can be reused and easily updated, and in which the separation of concerns permits to seamlessly modify the application. Nonetheless, another cause use could be envisaged: complete pre-created platforms could be imported via an inverse transformation; this is, from MHS to MARTE, leaving to the designer the task of allocating the DPR tasks in the available reconfigurable partitions, further easing the task of the modeler in UML MARTE and the overall design process.

# III

# General Conclusion, Perspectives and Scientific Publications

# General Conclusion

During the last decade, DPR has been widely studied as a research topic. Despite its intuitive appeal, the technique had eluded widespread adoption, particularly in industrial applications. This is due to the inherent complexity of the provided design flow and the in-depth knowledge of many low level aspects of FPGA technologies used to implement DPR systems. However, with recent developments in FPGA technologies and with the automation of many of the burdensome steps in the design flow, this trend is expected to change, and some exciting new products have already been demonstrated.

In this thesis, we have concentrated our efforts in the creation of the structural description of the system that is used as an input to the DPR design flow to facilitate the design entry phase of the DPR design flow. The presented approach is based on two widely used standards, UML MARTE and IP-XACT that until recent years had been developed in parallel. A great deal of research has been carried out to unify both standards, given the opportunities offered by the IP-XACT standard for interchanging IP descriptions among EDA tools. However, as it has been exposed in this paper, IP-XACT can also be exploited as a means for providing and intermediate system description that can be used to pass from UML MARTE models to HDL code generation. An IP-XACT compliant design environment facilitates the configuration and interconnection of complex systems, and provides mechanisms for EDA that can be used to automate many of the burdensome tasks in SoC design flows.

We have shown how IP-XACT can be used to generate the top level HDL description of the system, along with the necessary reconfigurable IPs that are gathered from a component library. We made this by separating the target back-end models from the high-level descriptions. The presented IP-XACT component descriptions contain vendor extensions that allow us to integrate our methodology in the Xilinx design ecosystem for DPR systems, but in such a ways that it does not to impact the interchangeability of the models. Therefore, the IP-XACT models can be extended for targeting different back-ends, allowing to easily evolve the methodology to changing requirements or to adapt it to other vendors. Moreover, we have presented a deployment extension to the MARTE profile that enable us to import relevant information to the UML models that are subsequently used for system generation purposes, facilitating IP reuse in the process. Then, we introduced the model transformations necessary to move from UML MARTE to IP-XACT, and from the utilized Xilinx PSF models to generate the EDK platform, our targeted back-end. Furthermore, we have demonstrated our methodology through a case study in which an

image processing IP is integrated into MicroBlaze based SoC design. Using MARTE and IP-XACT makes the design or DPR more amenable, and at the same time, helps in decoupling the high-level models from the intended back-end. This is achieved through the use of a generic IP deployment meta-model, which does not make particular assumptions on the nature of the low-level implementation details.

Compared to the relatively large number of proposals in the modeling of SoCs using UML MARTE on the one hand, and a combination of UML and IP-XACT on the other, to the best of our knowledge, approaches that use both in the specification and generation of DPR systems have not been presented by the scientific community. Moreover, just about half of the approaches described in the related works chapter target FPGA implementations. Furthermore, these methodologies do not provide clear or effective mechanisms for IP reuse, due in large part to the intermediate XML representations for the deployment phase. These XML IRs usually are not oriented for the description of SoC platforms, and thus have failed to attract the interest of the industry. We have demonstrated how the combination of UML MARTE and IP-XACT can improve the applicability of the model-driven approaches to the development of FPGA-based SoC platforms, and in particular, facilitate the conception of DPR systems. This has been achieved by combining a component-based approach and a well fixed IP taxonomy for easily associating different blocks in the UML MARTE models to their IP-XACT. However, targeting pure VHDL generation might be counter-intuitive, since it does not lend itself to further IP reuse; it is thus preferable to exploit the capabilities of IP-XACT as a standard intermediate representation for generating the desired back-end representations. We have proceeded in this manner: we use IP-XACT for promoting IP reuse (through IP reflection) and design by reuse by generation the XPS platform representation from IP-XACT. In both cases, the model transformations enable this passage seamlessly, without compromising the necessary abstraction in UML MARTE and the flow agnostic philosophy of IP-XACT regarding the target back-end.

Nonetheless, we must emphasize an important point: the proposed methodology is not only valid for Xilinx tools; for any MDE approach to be applicable, most of the modeling must be technologic independent, but only before the deployment level. The works presented in this manuscript address this modeling phase, in which the high-level models are allocated to back-end specific artifacts (i.e. the IP components in VHDL) with design flow EDA information (e.g. a tool flow and its associated metadata). Regardless of the chosen back-end, an IP-XACT design is created from UML MARTE, and the back-end generation phase depends only on the used generators (i.e. in MDE, model transformations); this is achieved by storing IP-XACT components with back-end views and the associated parameters and vendor extensions, which are only parsed on the context of a particular phase. Thus, the FAMOUS approach should be easily adapted to design flows which make use or pure VHDL representations, and where the C code to be run on the processor is generated independently.

# PERSPECTIVES

The research in dynamic partial reconfiguration is not bounded to the development of applications developed using the traditional design flow, which despite the enormous advantages that presents to the SoC and Reconfigurable Computing communities, still suffers from lack of the complexity of systems that can be created [152] by the traditional DPR flow. Many works have been proposed during the last decade that aim at extending the capabilities of the technology for creating more complex applications, some examples being the Replica approach [153]. Furthermore, DPR applications cannot only be based on bus interfaces such as CoreConnect or AXI, but must include other efforts introduced by the academy and the industry, such as the Wishbone [147] and OCP-IP protocols, or even in DPR dedicated bus architectures [154]. Thus, the work presented in this thesis manuscript can be continued in several directions:

***1. Exploring other wrapping interfaces.*** In this work, we have concentrated in wrapping the static and DPR placeholders around the CoreConnect IPIF interface. However, through the use of IP-XACT, bus and abstraction definitions for other bus protocols can be easily defined. This can prove particularly helpful in decoupling the FAMOUS framework from the Xilinx Platform Studio IP descriptions, leading to a more general ecosystem. Another possibility in this direction is to explore the IO blocks with the OCP-IP [85] protocol, which has been gaining traction in the SoC community in recent years, and that can be seen as an orthogonal effort to those carried by the Spirit Consortium with the standardization of IP-XACT: OCP-IP wrappers can be deployed to further foster IP reuse and exchange by providing a common infrastructure for heterogeneous IPs with various bus protocol interfaces. Furthermore, the IP-XACT design description of such systems, which represents a standard XML-IP system representation, would facilitate its exchange among different tools, promoting the collaboration in the academia, not only in the SoC domain, but in the Reconfigurable Computing community.

The main difference between the approach proposed in this manuscript, and the extensions described above, would lie in the automatic generation of the top-level RTL description of the hardware SoC platform, without passing through the Xilinx XPS (or any other) intermediate metadata description. However, more research needs to be carried out in the wrapping the DPR wrapper IPs and the PRM tasks, which might lead to well accepted interfaces that can be used by the Reconfigurable Computing community, fostering the development of partially reconfigurable systems and standard IPs.

***2. Extensions to support other DPR back-ends.*** In the FAMOUS approach, we have tackled the modeling and automatic generation of artifacts (static and PRM netlists, C code for the application and reconfiguration controller) that support directly the Xilinx Partition-based Partial Reconfiguration design flow. In the particular context of this thesis, we have targeted the generation of the design entry output artifacts (which serve as

inputs to the Xilinx PlanAhead tool for implementing the DPR system). Other works in the FAMOUS map those artifacts to a physical model at high-levels of abstraction, in order to automate the rest of the flow; this has been possible to the componentizing (i.e. IP taxonomy) and interface standardization introduced in this work.

Nonetheless, other Partial Reconfiguration design flows, making in many cases of Xilinx tools, exist. Some examples of these flows, developed very recently are OpenPR [155] and GoAhead [156]. Along tools such as the RapidSmith [157] framework, they have been used in recent years for overcoming several limitations inherent to the traditional Xilinx PR flow, such as the incapacity to relocate a single bitstream in multiple PRRs in the FPGA, a very important feature in applications such as fault-detection and migration or task virtualization. It most be noted, however, that all these approaches depart from a bottom-up design approach for the specification of the DPR system (the design entry phase profusely discussed in this thesis manuscript) which is not very dissimilar from what we have proposed here.

In most of these approaches, IP blocks are wrapped using well defined PRM interfaces, such as the Recobus [158] PR wrapper and design flow, upon which the GoAhead tool has been developed [159]. Making use of homogeneous and scalable bus interfaces, a set of advanced features are provided to the user, such as the capability to control the granularity of the PRMs to be placed in the FPGA, and to relocate modules in different areas of the device. These features can be coupled with the efforts carried in the FAMOUS framework on context saving for creating complex OS-like DPR applications, in which PRM effectively become services used by the application, while at the same time reducing the memory footprint required to store the partial bitstreams [160]. Furthermore, RapidSmith make use of hard macro IPs along the soft IPs discussed in this work in order to accelerate the system implementation times [161], but also to provide a framework for other DPR tools to enable capabilities not supported by the Xilinx flow. RapidSmith also departs from a design entry specification, in which hard and soft IPs are instantiated in a top-level design and compiled using a set of directives embedded in the tool.

These methodologies could be easily integrated to the FAMOUS framework, which is based on IP-XACT components, necessitating only vendor extensions and a modification in the generators used for the back-end compilation chain, which can be stored in the IP XML representation. In this manner, more complex applications can be created using the hardware branch of the FAMOUS MDE framework, proposed in this work. The use of UML MARTE would accomplish the separation of concerns in the development of the application and reconfiguration control, while at the same time, abstracting the specifics of tools that still require a great deal of expertise. However, further extensions in the UML MARTE profile, which represents one of the axes of the FAMOUS endeavour, would be required, since currently only traditional DPR platforms are supported. However, from a purely hardware perspective, the ideas discussed in this manuscript could be directly exploited to generate the artifacts (typically netlists) required by those tools.

# PUBLICATIONS

## 1. International Conferences

***IP-XACT and MARTE Based Approach for Partially Reconfigurable Systems-On-Chip***, G. Ochoa-Ruiz, K. Messaoudi, E.-B. Bourennane, O. Labbani, Forum on Specification and Design Languages (FDL), Germany, 2011

***High-Level Modeling and Automatic Generation of Dynamically Reconfigurable Systems***, G. Ochoa-Ruiz, E.-B. Bourennane, H. Rabah, O. Labbani, Design and Architectures for Signal and Image Processing (DASIP), Finland, November, 2011

***Model-Driven Approach for Automatic Dynamic Partially Reconfigurable IP Customization***, G. Ochoa-Ruiz, O. Labbani, E.-B. Bourennane, P. Soulard, IEEE International Parallel and Distributed Processing Symposium - Reconfigurable Architectures Workshop, (IPDPS - RAW), Shanghai, China, May 2012.

***Enabling Dynamic Partially Reconfigurable IP Core Parameterization and Integration using IP-XACT and MARTE***, G. Ochoa-Ruiz, O. Labbani, E.-B. Bourennane, S. Cherif, S. Meftali, J.-L. Dekeyser, IEEE International Symposium on Rapid System Prototyping (IEEE RSP), Finland, 2012.

***Facilitating IP Deployment in MARTE-based MDE Methodology using IP-XACT: a Xilinx EDK Case Study***, G. Ochoa-Ruiz, O. Labbani, E.-B. ÂšBourennane, S. Cherif, S. Meftali, J.-L. Dekeyser, International Conference in Reconfigurable Computing and FPGAs (ReConFig), Mexico, 2012

***A novel approach for Accelerating Bitstream Relocation in Many-core Partially Reconfigurable Applications***, G. Ochoa-Ruiz, M. Touiza, E.-B. Bourennane, A. Guessoum, K. Messaoudi, M.A. Hajjaji, IEEE Conference on Control, Decision and Information Technologies (CODIT), Hammamet, Tunisia, May 2013.

## 2.International Journals

***A Novel Methodology for Accelerating Bitstream Relocation in Partially Reconfigurable Systems***, M. Touiza, G. Ochoa-Ruiz, E.-B. Bourennane, A. Guessoum, K. Messaoudi, Microprocessors and Microsystems, Elsevier, 2012

***A High-level Methodology for Automatically Generating Dynamic Partially Reconfigurable Systems using IP-XACT and the UML MARTE Profile***, G. Ochoa-Ruiz, O. Labbani, E.-B. Bourennane, P. Soulard, S. Cherif, Design Automation for Embedded Systems, Springer, 2012

***An MDE-based Approach for the Automatic Integration of Flexible Hardware Support in Partially Reconfigurable Systems***, P. Wattebled, G. Ochoa-Ruiz, J.-P. Diguet, J.-L. Dekeyser, O. Labbani, E.-B. Bourennane, Journal on Systems Architecture, Elsevier (to be submitted, 2013)

## 3. National Conferences

***Dynamic Partial Reconfiguration Exploitation for Reducing the Resource Utilization of the Deblocking Filter in H.264 CODECS Implementations***, G. Ochoa-Ruiz, K. Messaoudi, E.-B. Bourennane, O. Labbani, Forum Jeunes Chercheurs, Dijon, France, July 2011.

***Model-Driven Approach for Automatic IP Integration in Dynamically Reconfigurable Designs***, G. Ochoa-Ruiz, E.-B. Bourennane, O. Labbani, Colloque GDR SoC-SiP. Lyon, France, June 2011

***Towards Dynamic Partially ReconfigurationIP Parameterisation and Integration Using IP-XACT***, G. Ochoa-Ruiz, E.-B. Bourennane, O. Labbani. Colloque GDR SoC-SiP, Paris, France, Juin 2012

***A Metadata-based Composition Framework for Dynamic Partially Reconfigurable Systems based on IP-XACT***, G. Ochoa-Ruiz, E.-B. Bourennane, O. Labbani, Colloque GDR SoC-SiP. Lyon, France, Juin 2013

# List of Figures

# List of Acronyms

**AMBA** Advanced Microcontroller Bus Architecture

**ASCII** American Standard Code for Information Interchange

**ASIC** Application Specific Integrated Circuit

**AXI** Advanced eXtensible Interface

**BRAM** Block RAM

**CAD** Computer Aided Design

**CLB** Complex Logic Block

**DCT** Discrete Cosine Transform

**DE** Design Environment

**DMA** Direct Memory Access

**DPR** Dynamic Partial Reconfiguration

**DCT** Discrete Cosine Transform

**DSP** Digital Signal Processor - Processing

**ECORE** ecore

**EDA** Electronic Design Automation

**ESL** Electronic System Level

**FAMOUS** FAst Modeling and design flOw for dynamically reconfigUrabe Systems

**FFT** Fast Fourier Transform

**FPGA** Field Programmable Gate Array

**FSM** Finite State Machine

**GUI** Graphical User Interface

**HDL** Hardware Description Language

**HWAcc** Hardware Accelerator

**IEEE** Institute of Electrical and Electronics Engineers

**IP** Intellectual Property Block

**IP-XACT** IEEE 1685-2009, IEEE Standard Structure for Packaging, Integrating, and Reusing IP within Tools Flows

**IPIC** Intellectual Property Interconnect

**IPIF** Intellectual Property Interface

**IR** Intermediate Representation

**LUT** Look-up Table

**LVDS** Low-voltage Differential Signaling

**MADES** Low-voltage Differential Signaling

**MAP** Xilinx Mapping Technology

**MARTE** Modelling and Analysis of Real Time and Embedded Systems

**MCF** Metadata-based Component Composition Framework

**MDE** Model-driven Engineering

**MTT** Model Transformations Tool

**MHS** Microprocessor Hardware Specification

**MOF** Meta-Object Facility

**MPD** Microprocessor Peripheral Definition

**NRE** Non-recursive Engineering

**OCL** Object Constraint Language

**OCP-IP** Open Core Protocol International Partnership

**OMG** Object Management Group

**PAO** Peripheral Analysis Order file

**PAR** Place and Route

**PBD** Platform Based Design

**PCI** Peripheral Controller Interface

**XPSF** Xilinx Platform Specification Format

**PIM** Platform Independent Model

**PlatGen** Xilinx XPS Plaform Generation tool

**PLB** CoreConnect Processor Local Bus

**PRM** Partially Reconfigurable Module

**PRR** Partially Reconfigurable Region

**PSF** Platform Studio Format

**PSM** Platform Specific Model

**RP** Reconfigurable Partition

**RTL** Register Transfer Level

**RTR** Run-time Reconfiguration

**SoC** System-on-Chip

**SPIRIT** Structure for Packaging, Integrating and Re-using IP within Tool-flows

**TFT** Thin-Film Transistor Display

**TUT** Technical University of Tampere

**UART** Universal Asynhcronous Receiver/Transmitter

**UCF** User Constraints File

**UML** Universal Modeling Language

**VHDL** Very High Description Lnaguage

**VLNV** Vendor, Library, Name, Version

**VSIA** Virtual Socket Interface Association

**XMI** XML Metadata Interchange

**XMI-IR** XML Metadata Interchange - Intermediate Representation

**XML** eXtensible Markup Language

**XPS** Xilinx Plaform Studio

**XPSF** Xilinx Plaform Studio Format

**XST** Xilinx Synthesis Technology

# Bibliography


[1] Mark L. Chang. Chapter 1: Device architecture. *ReconPgurable Computing: The Theory and Practice of FPGA-Based Computation*, 2007.

[2] Christophe Bobda. *Introduction to Reconfigurable Computing: Architectures, Algorithms, and Applications*. Springer Publishing Company, Incorporated, 1st edition, 2007.

[3] Katherine Compton and Scott Hauck. Reconfigurable computing: a survey of systems and software. *ACM Comput. Surv.*, 34(2):171–210, June 2002.

[4] S. Hauck. The roles of fpgas in reprogrammable systems. *Proceedings of the IEEE*, 86(4):615–638, Apr.

[5] Clive Maxfield. *FPGA:s World Class Designs*. Newnes, 1st edition, 2009.

[6] J. Becker and M. Hubner. Chapter 22: Applications, design tools and low power issues in fpga reconfiguration. In JÃűrg Henkel and Sri Parameswaran, editors, *Designing Embedded Processors*, pages 451–501. Springer Netherlands, 2007.

[7] J. Rose, R.J. Francis, D. Lewis, and P. Chow. Architecture of field-programmable gate arrays: the effect of logic block functionality on area efficiency. *Solid-State Circuits, IEEE Journal of*, 25(5):1217–1225, Oct.

[8] R.J Rose, S.D. Brown, R.J. Francis, and Z.G. Vranesic. *Field-programmable Gate Arrays*. Kluwer Academic Publishers, 1992.

[9] Ian Kuon and Jonathan Rose. Quantifying and exploring the gap between fpgas and asics. *Springer*, 2010.

[10] A. Donlin. Chapter 22: Applications, design tools and low power issues in fpga reconfiguration. In JÃűrg Henkel and Sri Parameswaran, editors, *Designing Embedded Processors*, pages 451–501. Springer Netherlands, 2007.

[11] Xilinx. http://www.xilinx.com, 2013.

[12] Xilinx. Command line tools user guide, ug628. http://www.xilinx.com, 2012.

[13] Xilinx. Embedded system tools reference manual, ug111. http://www.xilinx.com, 2012.

[14] Philippe Manet, Daniel Maufroid, Leonardo Tosi, Gregory Gailliard, Olivier Mulertt, Marco Di Ciano, Jean-Didier Legat, Denis Aulagnier, Christian Gamrat, Raffaele Liberati, Vincenzo La Barba, Pol Cuvelier, Bertrand Rousseau, and Paul Gelineau. An evaluation of dynamic partial reconfiguration for signal and image processing in professional electronics applications. *EURASIP J. Embedded Syst.*, 2008:1:1–1:11, January 2008.

[15] J. Becker, M. Hubner, G. Hettich, R. Constapel, J. Eisenmann, and J. Luka. Dynamic and partial fpga exploitation. *Proceedings of the IEEE*, 95(2):438–452, Feb.

[16] Christopher Claus, Rehan Ahmed, Florian Altenried, and Walter Stechele. Towards rapid dynamic partial reconfiguration in video-based driver assistance systems. In *Proceedings of the 6th international conference on Reconfigurable Computing: architectures, Tools and Applications*, ARC'10, pages 55–67, Berlin, Heidelberg, 2010. Springer-Verlag.

[17] J. Strunk, T. Volkmer, K. Stephan, Wolfgang Rehm, and H. Schick. Impact of run-time reconfiguration on design and speed - a case study based on a grid of run-time reconfigurable modules inside a fpga. In *Parallel Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on*, pages 1–8, May.

[18] K. Compton. *Chapter 4: Reconfiguration Management*. Morgan Kaufmann/Elsevier, 2008.

[19] Patrick Lysaght, On Blodget, and Jeff Mason. Enhanced architectures, design methodologies and cad tools for dynamic reconfiguration of xilinx fpgas, 2006.

[20] Gilberto Ochoa-Ruiz. *Master Dissertation : Dynamic Partial Reconfiguration and Video Distribution in a Reconfigurable Device*. Université de Bourgogne, 2009.

[21] Nicolas Marques. *PhD Dissertation :Methodologie et architecture adaptative pour le placement efficace de taches maternelles de tailles variables sur de partitions reconfigurables*. Université de Lorraine, 2012.

[22] P. Sedcole, B. Blodget, T. Becker, J. Anderson, and P. Lysaght. Modular dynamic reconfiguration in virtex fpgas. *Computers and Digital Techniques, IEE Proceedings* -, 153(3):157–164, May.

[23] Xilinx. Partial reconfiguration overview, wp374. http://www.xilinx.com, 2012.

[24] Xilinx. Partial reconfiguration user guide, ug207. http://www.xilinx.com, 2012.

[25] Xilinx. Planahead user guide, ug632. http://www.xilinx.com, 2012.

[26] Xilinx. Ise in-depth tutorial, ug695. http://www.xilinx.com, 2012.

[27] S. Neuendorffer. *Chapter 12: FPGA Platforms for Embedded Systems*. Morgan Kaufmann, 2010.

[28] On Blodget, Philip James-roxby, Eric Keller, Scott Mcmillan, and Prasanna Sundararajan. A self-reconfiguring platform. In *In Proceedings of Field Programmable Logic and Applications (2003*, pages 565–574, 2003.

[29] B. Blodget, S. McMillan, and P. Lysaght. A lightweight approach for embedded reconfiguration of fpgas. In *Design, Automation and Test in Europe Conference and Exhibition, 2003*, pages 399–400.

[30] H. Kwok-Hay So and R.W. Brodersen. Improving usability of fpga-based reconfigurable computers through operating system support. In *Field Programmable Logic and Applications, 2006. FPL '06. International Conference on*, pages 1–6, Aug.

[31] M.D. Santambrogio, V. Rana, and D. Sciuto. Operating system support for online partial dynamic reconfiguration management. In *Field Programmable Logic and Applications, 2008. FPL 2008. International Conference on*, pages 455–458, Sept.

[32] Xilinx. Edk concepts, tools, and techniques, ug683. http://www.xilinx.com, 2012.

[33] Pierre Bomel, Jeremie Crenne, Linfeng Ye, Jean-Philippe Diguet, and Guy Gogniat. Ultra-fast downloading of partial bitstreams through ethernet. In *Proceedings of the 22nd International Conference on Architecture of Computing Systems*, ARCS '09, pages 72–83, Berlin, Heidelberg, 2009. Springer-Verlag.

[34] Xilinx. Logicore ip xps hwicap(v5.01a). http://www.xilinx.com, 2011.

[35] Imran R. Quadri. *Phd Dissertation : MARTE based model driven design methodology for targeting dynamically reconfigurable FPGA based SoCs*. Université des Sciences et Technologies de Lille, 2011.

[36] Sana Cherif, Imran Rafiq Quadri, Samy Meftali, and Jean-Luc Dekeyser. Modeling reconfigurable systems-on-chips with uml marte profile: An exploratory analysis. In *DSD*, pages 706–713, 2010.

[37] Sébastien Guillet, Florent de Lamotte, Nicolas Le Griguer, Éric Rutten, Jean-Philippe Diguet, and Guy Gogniat. Modeling and synthesis of a dynamic and partial reconfiguration controller. In *FPL*, pages 703–706, 2012.

[38] Xin An, Sarra Boumedien, Abdoulaye Gamatié, and Éric Rutten. Classy: a clock analysis system for rapid prototyping of embedded applications on mpsocs. In *Proceedings of the 15th International Workshop on Software and Compilers for Embedded Systems*, SCOPES '12, pages 3–12, New York, NY, USA, 2012. ACM.

[39] Pamela Wattebled, Jean-Philippe Diguet, and Jean-Luc Dekeyser. Membrane-based design and management methodology for parallel dynamically reconfigurable embedded systems. In *ReCoSoC*, pages 1–8, 2012.

[40] Gilberto Ochoa, El-Bay Bourennane, Ouassila Labbani, and Kamel Messaoudi. Ip-xact and marte based approach for partially reconfigurable systems-on-chip. In *FDL*, pages 1–8, 2011.

[41] Gilberto Ochoa-Ruiz, Ouassila Labbani, El-Bay Bourennane, Philippe Soulard, and Sana Cherif. A high-level methodology for automatically generating dynamic partially reconfigurable systems using ip-xact and the uml marte profile. *Design Automation for Embedded Systems*, pages 1–36, 2012.

[42] Xilinx. Platform specification format reference manual embedded development kit (edk) 14.1, ug642. http://www.xilinx.com, 2012.

[43] A. Sangiovanni-Vincentelli. Quo vadis, sld? reasoning about the trends and challenges of system level design. *Proceedings of the IEEE*, 95(3):467 –506, march 2007.

[44] M.F. Jacome and H.P. Peixoto. A survey of digital design reuse. *Design Test of Computers, IEEE*, 18(3):98 –107, may 2001.

[45] R. Saleh, S. Wilton, S. Mirabbasi, A. Hu, M. Greenstreet, G. Lemieux, P.P. Pande, C. Grecu, and A. Ivanov. System-on-chip: Reuse and integration. *Proceedings of the IEEE*, 94(6):1050 –1069, june 2006.

[46] Rong Chen, Marco Sgroi, Luciano Lavagno, Grant Martin, Alberto Sangiovanni-Vincentelli, and Jan Rabaey. Uml and platform-based design, 2003.

[47] Luis G. Murilo, Marcello Mura, and Mauro Prevostini. MDE Support for HW/SW Codesign: A UML-Based Design Flow. pages 19–37, 2010.

[48] Yves Vanderperren, Wolfgang Mueller, and Wim Dehaene. Uml for electronic systems design: a comprehensive overview. *Design Automation for Embedded Systems*, 12:261–292, 2008.

[49] Grant Martin. Uml for embedded systems specification and design: Motivation and overview. In *Conference on Design, Automation and Test in Europe (DATE)*, pages 773–775, 2002.

[50] Elvinia Riccobene, Alberto Rosti, and Patrizia Scandurra. Improving soc design flow by means of mda and uml profiles, 2005.

[51] Grant Martin, Luciano Lavagno, and Jean Louis-Guerin. Embedded uml: a merger of real-time uml and co-design. In *CODES*, pages 23–28, 2001.

[52] Brian Bailey and Grant Martin. Ip meta-models for soc assembly and hw/sw interfaces. In *ESL Models and their Application*, Embedded Systems, pages 33–82. Springer US, 2010.

[53] Jean BÃľzivin. On the unification power of models. *Software and Systems Modeling*, 4:171–188, 2005.

[54] Object Management Group Inc. Uml 2 infrastructure (final adopted specifcation). http://www.omg.org/spec/UML/2.1.2/Infrastructure/PDF/.

[55] Eclipse. Eclipse modeling framework. http://www.eclipse.org/emf.

[56] Object Management Group Inc. Mof 2.0 core final adopted specification. http://www.omg.org/cgi-bin/doc?ptc/03-10-04, 2003.

[57] S. Sendall and W. Kozaczynski. Model transformation: the heart and soul of model-driven software development. *Software, IEEE*, 20(5):42 – 45, sept.-oct. 2003.

[58] Krzysztof Czarnecki and Simon Helsen. Classification of model transformation approaches. In *Workshop on Generative Techniques in the Context of Model-Driven Architecture*, 2003.

[59] K. Czarnecki and S. Helsen. Feature-based survey of model transformation approaches. *IBM Syst. J.*, 45(3):621–645, July 2006.

[60] Joaquin Miller and Jishnu Mukerji. Model driven architecture (MDA). Draft ormsc/2001-07-01, Architecture Board ORMSC, July 2001.

[61] Object Management Group Inc. Xml metadata interchange format. www.omg.org/spec/XMI/.

[62] Grant Martin and Wolfgang Muller. *UML for SOC Design*. Springer, 2005.

[63] Fateh Boutekkouk, Mohamed Benmohammed, Sebastien Bilavarn, and Michel Auguin. Uml2.0 profiles for embedded systems and systems on a chip (socs). *Journal of Object Technology*, 8(1):135–157, 2009.

[64] Marjan Mernik, Jan Heering, and Anthony M. Sloane. When and how to develop domain-specific languages. *ACM Comput. Surv.*, 37(4):316–344, December 2005.

[65] Fateh Boutekkouk, Mohamed Benmohammed, Sebastien Bilavarn, and Michel Auguin. Uml2.0 profiles for embedded systems and systems on a chip (socs). *Journal of Object Technology*, 8(1):135–157, 2009.

[66] Thomas Beierlein, Dominik FrŽhlich, Bernd Steinbach, Hochschule Mittweida, and Technische UniversitŁt Bergakademie Freiberg. Model-driven compilation of umlmodels for reconfigurable architectures. In *in Proceedings of the Second RTAS Workshop on Model-Driven Embedded Systems (MoDES04*, 2004.

[67] Safouan Taha, Ansgar Radermacher, Sébastien Gérard, and Jean-Luc Dekeyser. Marte: Uml-based hardware design from modeling to simulation. In *Forum on Design and Specification of Design Languages*, pages 274–279, 2007.

[68] S. Taha, A. Radermacher, S. Gerard, and J.-L. Dekeyser. An open framework for detailed hardware modeling. In *Industrial Embedded Systems, 2007. SIES '07. International Symposium on*, pages 118 –125, july 2007.

[69] Safouan Taha. *Phd Dissertation : Modélisation Cojointe Logiciele/Matériel de Systèmes temps Réel*. Université des Sciences et Technologies de Lille, 2008.

[70] OMG. Modeling and analysis of real-time and embedded systems marte), beta 3. http://www.omgwiki.org/marte-ftf2/doku.php, 2011.

[71] OMG. Model-driven engineering. http://www.ogm/org/mda, 2012.

[72] Abdoulaye Gamatié, Sébastien Le Beux, Éric Piel, Rabie Ben Atitallah, Anne Etien, Philippe Marquet, and Jean-Luc Dekeyser. A model-driven design framework for massively parallel embedded systems. *ACM Trans. Embed. Comput. Syst.*, 10(4):39:1–39:36, November 2011.

[73] D.D. Gajski, A.C.-H. Wu, V. Chaiyakul, S. Mori, T. Nukiyama, and P. Bricaud. Essential issues for ip reuse. In *Design Automation Conference, 2000. Proceedings of the ASP-DAC 2000. Asia and South Pacific*, pages 37 –42, june 2000.

[74] J. Teich. Hardware/software codesign: The past, the present, and predicting the future. *Proceedings of the IEEE*, 100(Special Centennial Issue):1411–1430, 13.

[75] M.F.S. Oliveira, L.B. de Brisolara, L. Carro, and F.R. Wagner. Early embedded software design space exploration using uml-based estimation. In *Rapid System Prototyping, 2006. Seventeenth IEEE International Workshop on*, pages 24–32, June.

[76] J. Altmeyer, S. Ohnsorge, and B. Sch§rmann. Reuse of design objects in cad frameworks. Technical report, ICCAD, 1994.

[77] Reinaldo A. Bergamaschi and William R. Lee. Designing systems-on-chip using cores. In *In the 37th Design Automation Conference*, pages 420–425, 2000.

[78] W.O Cesario, A. Baghdadi, L. Gauthier, D. Lyonnard, G. Nicolescu, Y. Paviot, S. Yoo, A.A. Jerraya, and M. Diaz-Nava. Component-based design approach for multicore socs. In *Design Automation Conference, 2002. Proceedings. 39th*, pages 789–794.

[79] W. Wolf. *Chapter 11: Intellectual Property-Based Design*. CRC Press, 2008.

[80] Sudeep Parischa and Nikil Dutt. *Chapter 3: On-Chip Communication Architecture Standards*. CRC Press, 2008.

[81] IBM. Coreconnect bus architecture. https://www-01.ibm.com/chips/techlib/techlib.nsf/productfamilies/CoreConnect_Bus_Architecture, 2013.

[82] ARM. Amba open specifications. http://www.arm.com/products/system-ip/amba/amba-open-specifications.php, 2013.

[83] W.O. Cesario, D. Lyonnard, G. Nicolescu, Y. Paviot, Sungjoo Yoo, A.A. Jerraya, L. Gauthier, and M. Diaz-Nava. Multiprocessor soc platforms: a component-based design approach. *Design Test of Computers, IEEE*, 19(6):52–63, Nov.-Dec.

[84] Virtual Socket Interface Alliance. Virtual socket interface specification. http://vsi.org/, 1996.

[85] Open Core Protocol Initiative. Open core protocol specification. http://www.ocpip.org/, 2013.

[86] A. Sangiovanni-Vincentelli, Guang Yang, S.K. Shukla, D.A. Mathaikutty, and J. Sztipanovits. Metamodeling: An emerging representation paradigm for system-level design. *Design Test of Computers, IEEE*, 26(3):54 –69, may-june 2009.

[87] V. Berman. Standards: The p1685 ip-xact ip metadata standard. *Design Test of Computers, IEEE*, 23(4):316 –317, april 2006.

[88] IEEE. Ieee 1685, Òieee standard for ip-xact, standard structure for packaging, integrating, and reusing ip within tools flowsÓ. http://standards.ieee.org/getieee/1685/download/1685-2009.pdf, 2009.

[89] James A. Rowson and Alberto Sangiovanni-Vincentelli. Interface-based design. In *Proceedings of the 34th annual Design Automation Conference*, DAC '97, pages 178–183, New York, NY, USA, 1997. ACM.

[90] Michael Keating and Pierre Bricaud. *Reuse methodology manual: for system-on-a-chip designs*. Kluwer Academic Publishers, 2003.

[91] W. Wolf. *Chapter 8: Architecture Design*. Prentice Hall, 2009.

[92] Altera. http://www.altera.com, 2013.

[93] M. Koegst, P. Conradi, D. Garte, and M. Wahl. A systematic analysis of reuse strategies for design of electronic circuits. In *Design, Automation and Test in Europe, 1998., Proceedings*, pages 292–296, Feb.

[94] Peter Conradi. *Reuse in Electronic Design: From Information Modelling to Intellectual Properties*. Wiley, 1999.

[95] Debasri Saha and Susmita Sur-Kolay. Soc: a real platform for ip reuse, ip infringement, and ip protection. *VLSI Des.*, 2011:5:1–5:10, January 2011.

[96] D. Densmore and R. Passerone. A platform-based taxonomy for esl design. *Design Test of Computers, IEEE*, 23(5):359 –374, may 2006.

[97] N. Medvidovic and R.N. Taylor. A classification and comparison framework for software architecture description languages. *Software Engineering, IEEE Transactions on*, 26(1):70 –93, jan 2000.

[98] W.O. Cesario, G. Nicolescu, L. Gauthier, D. Lyonnard, and A.A. Jerraya. Colif: A design representation for application-specific multiprocessor socs. *Design Test of Computers, IEEE*, 18(5):8 –20, sep-oct 2001.

[99] M. Wirthlin, D. Poznanovic, P. Sundararajan, A. Coppola, D. Pellerin, W. Najjar, R. Bruce, M. Babst, O. Pritchard, P. Palazzari, and G. Kuzmanov. Openfpga corelib core library interoperability effort. *Parallel Comput.*, 34(4-5):231–244, May 2008.

[100] S. Decker, S. Melnik, F. van Harmelen, D. Fensel, M. Klein, J. Broekstra, M. Erdmann, and I. Horrocks. The semantic web: the roles of xml and rdf. *Internet Computing, IEEE*, 4(5):63 –73, sep/oct 2000.

[101] A. Arnesen, K. Ellsworth, D. Gibelyou, T. Haroldsen, J. Havican, M. Padilla, B. Nelson, M. Rice, and M. Wirthlin. Increasing design productivity through core reuse, meta-data encapsulation, and synthesis. In *Field Programmable Logic and Applications (FPL), 2010 International Conference on*, pages 538 –543, 31 2010-sept. 2 2010.

[102] A. Arnesen, N. Rollins, and M. Wirthlin. A multi-layered xml schema and design tool for reusing and integrating fpga ip. In *Field Programmable Logic and Applications, 2009. FPL 2009. International Conference on*, pages 472 –475, 31 2009-sept. 2 2009.

[103] J. Lapalme, E.M. Aboulhamid, and G. Nicolescu. *Chapter 3: The Semantic Web Applied to IP-Based Design: A Discussion on IP-XACT in System Level Desgin with NET Technology*. CRC Press, 2010.

[104] Deepak A. Mathaikutty and Sandeep K. Shukla. *Metamodeling-Driven IP Reuse for SoC Integration and Microprocessor Design*. Artech House, Inc., Norwood, MA, USA, 1st edition, 2009.

[105] A. Ledeczi, A. Bakay, M. Maroti, P. Volgyesi, G. Nordstrom, J. Sprinkle, and G. Karsai. Composing domain-specific design environments. *Computer*, 34(11):44–51, Nov.

[106] OMG. Uml 2.0 ocl specification. http://www.omg.org/spec/OCL/2.3.1/PDF, 2012.

[107] M. Emerson, S. Neema, and S. Sztipanovits. Metamodeling Languages and Metaprogrammable Tools. *Handbook of Real-Time and Embedded Systems*.

[108] D.A. Mathaikutty and S.K. Shukla. Mcf: A metamodeling-based component composition framework: Composing systemc ips for executable system models. *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, 16(7):792 –805, july 2008.

[109] Joel M. Rahman, Shane P. Seaton, and Susan M. Cuddy. Making frameworks more useable: using model introspection and metadata to develop model processing tools. *Environmental Modelling and Software*, 19(3):275 – 284, 2004.

[110] F. Doucet, S. Shukla, and R. Gupta. Introspection in system-level language frameworks: meta-level vs. integrated. In *Design, Automation and Test in Europe Conference and Exhibition, 2003*, pages 382–387.

[111] Andy D. Pimentel, Todor Stefanov, Hristo Nikolov, Mark Thompson, Simon Polstra, and Ed F. Deprettere. Tool integration and interoperability challenges of a system-level design flow: A case study. In *Proceedings of the 8th international workshop on Embedded Computer Systems: Architectures, Modeling, and Simulation*, SAMOS '08, pages 167–176, Berlin, Heidelberg, 2008. Springer-Verlag.

[112] Spirit Consortium. http://www.spiritconsortium.org/home, 2008.

[113] Wido Kruijtzer, Pieter van der Wolf, Erwin de Kock, Jan Stuyt, Wolfgang Ecker, Albrecht Mayer, Serge Hustin, Christophe Amerijckx, Serge de Paoli, and Emmanuel Vaumorin. Industrial ip integration flows based on ip-xact? standards. *2008 Design, Automation and Test in Europe*, 0:32–37, 2008.

[114] C.K. Lennard, V. Berman, S. Fazzari, M. Indovina, C. Ussery, M. Strik, J. Wilson, O. Florent, F. Remond, and P. Bricaud. Industrially proving the spirit consortium specifications for design chain integration. In *Design, Automation and Test in Europe, 2006. DATE '06. Proceedings*, volume 2, pages 1 –6, march 2006.

[115] Xilinx Corp. Vivado design suite. http://www.xilinx.com/products/design-tools/vivado/index.htm, 2013.

[116] R. Nane, S. van Haastregt, T. Stefanov, B. Kienhuis, V.M. Sima, and K. Bertels. Ip-xact extensions for reconfigurable computing. In *Application-Specific Systems, Architectures and Processors (ASAP), 2011 IEEE International Conference on*, pages 215 –218, sept. 2011.

[117] Sebastien Revol, Safouan Taha, Francois Terrier, Alain Clouard, SŐastien Gerard, Ansgar Radermacher, and Jean-Luc Dekeyser. Unifying hw analysis and soc design flows by bridging two key standards: Uml and ip-xact. In *Distributed Embedded Systems: Design, Middleware and Resources*, volume 271 of *IFIP âThe International Federation for Information Processing*, pages 69–78. Springer US, 2008.

[118] Charles André, Frédéric Mallet, Aamir Mehmood Khan, and Robert De Simone. Modeling spirit ip-xact with uml marte, 2008.

[119] Andreas Vorg and Wolfgang Rosenstiel. Automation of ip qualification and ip exchange. *Integration, the VLSI Journal*, 37(4):323 – 352, 2004.

[120] Magillem Design Services. Magillem eda ip-xact tooling. http://www.magillem.com/eda/, 2013.

[121] Mentor Graphics. Hdl designer ip-xact. http://www.mentor.com/products/fpga/ip-xact/hdl-designer-ip-xact, 2013.

[122] Ali Koudri, Didier Vojtsiek, Philippe Soulard, Christophe Moy, Joel Champeau, Jorgiano Vidal, and Jean-christophe Le Lann. Using marte in the mopcom soc/sopc methodology. In *workshop MARTE*, March 2008.

[123] J. Vidal, F. de Lamotte, G. Gogniat, P. Soulard, and J.-P. Diguet. A co-design approach for embedded system modeling and code generation with uml and marte. In *Design, Automation Test in Europe Conference Exhibition, 2009. DATE '09.*, pages 226 –231, april 2009.

[124] J. Vidal, F. de Lamotte, G. Gogniat, J.-P. Diguet, and P. Soulard. Ip reuse in an mda mpsopc co-design approach. In *Microelectronics (ICM), 2009 International Conference on*, pages 256 –259, dec. 2009.

[125] Denis Aulagnier, Ali Koudri, Stephane Lecomte, Philippe Soulard, JoŚl Champeau, Jorgiano Vidal, Gilles Perrouin, and Pierre Leray. Soc/sopc development using mdd and marte profile, 2009.

[126] I.R. Quadri, S. Muller, S. Meftali, and J.-L. Dekeyser. Marte based design flow for partially reconÞgurable systems-on-chips. In *17th IFIP/IEEE International Conference on Very Large Scale Integration (VLSI-SoC 09)*, oct. 2009.

[127] OMG. Graphical array specification for parallel and distributed computing (gaspard) [online]. http://www2.lifl.fr/west/gaspard/, 2010.

[128] A. Ben Atitallah, P. Kadionik, F. Ghozzi, P. Nouel, N. Masmoudi, and H. Levi. Hw/sw codesign of the h. 263 video coder. In *Electrical and Computer Engineering, 2006. CCECE '06. Canadian Conference on*, pages 783 –787, may 2006.

[129] R. Ben Atitallah, S. Niar, S. Meftali, and J.-L. Dekeyser. An mpsoc performance estimation framework using transaction level modeling. In *Embedded and Real-Time Computing Systems and Applications, 2007. RTCSA 2007. 13th IEEE International Conference on*, pages 525 –533, aug. 2007.

[130] S. Le Beux, P. Marquet, and J.-L. Dekeyser. A design flow to map parallel applications onto fpgas. In *Field Programmable Logic and Applications, 2007. FPL 2007. International Conference on*, pages 605 –608, aug. 2007.

[131] I.R. Quadri, S. Meftali, and J.-L. Dekeyser. Designing dynamically reconfigurable socs: From uml marte models to automatic code generation. In *Design and Architectures for Signal and Image Processing (DASIP), 2010 Conference on*, pages 68 –75, oct. 2010.

[132] I.R. Quadri. Marte based model driven design methodology for targeting dynamically reconÞgurable fpga based socs. In *PhD Dissertation*, 2011.

[133] Imran Rafiq Quadri, Abdoulaye Gamatié, Pierre Boulet, Samy Meftali, and Jean-Luc Dekeyser. Expressing embedded systems configurations at high abstraction levels with uml marte profile: Advantages, limitations and alternatives. *Journal of Systems Architecture - Embedded Systems Design*, 58(5):178–194, 2012.

[134] I. Gray, N. Matragkas, N.C. Audsley, L.S. Indrusiak, D. Kolovos, and R. Paige. Model-based hardware generation and programming - the mades approach. In *Object/Component/Service-Oriented Real-Time Distributed Computing Workshops (ISORCW), 2011 14th IEEE International Symposium on*, pages 88 –96, march 2011.

[135] I.R. Quadri, E. Brosse, I. Gray, N. Matragkas, L.S. Indrusiak, M. Rossi, A. Bagnato, and A. Sadovykh. Mades fp7 eu project: Effective high level sysml/marte methodology for real-time and embedded avionics systems. In *Reconfigurable Communication-centric Systems-on-Chip (ReCoSoC), 2012 7th International Workshop on*, pages 1 –8, july 2012.

[136] T. Schattkowsky and Tao Xie. Uml and ip-xact for integrated sprint ip management. In *Proc. 5th Int. UML-SoC DAC Workshop*, 2008.

[137] T. Schattkowsky, Tao Xie, and W. Mueller. A uml frontend for ip-xact-based ip management. In *Design, Automation Test in Europe Conference Exhibition, 2009. DATE '09.*, pages 238 –243, april 2009.

[138] Andre Charles, Aamir Mehmood Khan, and Robert De Simone. Modeling spirit ip-xact with uml marte. In *DATE Workshop on Modeling and Analysis of Real-Time and Embedded Systems with the MARTE UML profile*, 2008.

[139] Jean-François Le Tallec and Robert De Simone. SCIPX: a SystemC to IP-XACT extraction tool. In *ESLsyn : Electronic System Level Synthesis Conference*, San Diego, États-Unis, June 2011.

[140] Jean-François Le Tallec, Julien Deantoni, Robert De Simone, Benoît Ferrero, Frédéric Mallet, and Laurent Maillet-Contoz. Combining SystemC, IP-XACT and

UML/MARTE in model-based SoC design. In *Workshop on Model Based Engineering for Embedded Systems Design (M-BED 2011)*, Grenoble, France, March 2011. This paper has been partially supported by the French ANR project HELP (ANR-09-SEGI-006).

[141] P. Kukkala, J. Riihimaki, M. Hannikainen, T.D. Hamalainen, and K. Kronlof. Uml 2.0 profile for embedded system design. In *Design, Automation and Test in Europe, 2005. Proceedings*, pages 710 – 715 Vol. 2, march 2005.

[142] Tero Kangas, Petri Kukkala, Heikki Orsila, Erno Salminen, Marko Hännikäinen, Timo D. Hämäläinen, Jouni Riihimäki, and Kimmo Kuusilinna. Uml-based multiprocessor soc design framework. *ACM Trans. Embed. Comput. Syst.*, 5(2):281–320, May 2006.

[143] T. Arpinen, T. Koskinen, E. Salminen, T.D. Hamalainen, and M. Hannikainen. Model-driven approach for automatic spirit ip integration, 2008.

[144] T. Arpinen, T. Koskinen, E. Salminen, T.D. Hamalainen, and M. Hannikainen. Evaluating uml2 modeling of ip-xact objects for automatic mp-soc integration onto fpga. In *Design, Automation Test in Europe Conference Exhibition, 2009. DATE '09.*, pages 244 –249, april 2009.

[145] F. Herrera and E. Villar. A framework for the generation from uml/marte models of ipxact hw platform descriptions for multi-level performance estimation. In *Specification and Design Languages (FDL), 2011 Forum on*, pages 1 –8, sept. 2011.

[146] F. Herrera, H. Posadas, E. Villar, and D. Calvo. Enhanced ip-xact platform descriptions for automatic generation from uml/marte of fast performance models for dse. In *Digital System Design (DSD), 2012 15th Euromicro Conference on*, pages 692 –699, sept. 2012.

[147] Open Cores. Wishbone protocol specification. http://cdn.opencores.org/downloads/wbspec_b4.pdf, 2010.

[148] Chun-Hsian Huang and Pao-Ann Hsiung. Dynamically swappable hardware design in partially reconfigurable systems. *IEEE Circuits and Systems*, 2008.

[149] Chun-Hsian Huang and Pao-Ann Hsiung. Software-controlled dynamically swappable hardware design in partially reconfigurable systems. *EURASIP Journal on Embedded Systems*, 2008(1):231940, 2008.

[150] Sana Cherif. *PhD Dissertation: Approche basée sur les modèles pour la conception des systèemes dynamiquement reconfigurables : de MARTE vers RecoMARTE*. Université de Bretagne Sud, 2013.

[151] Sébastien Guillet. *PhD Dissertation: Modélisation et controle formel de la reconfiguration: Application aux systèmes embarqués dynamiquement reconfigurables*. Université des Sciences et Technologies de Lille, 2012.

[152] Dirk Koch, Jim Torresen, Christian Beckhoff, Daniel Ziener, Christopher Dennl, Volker Breuer, Jürgen Teich, Michael Feilen, and Walter Stechele. Partial reconfiguration on fpgas in practice - tools and applications. In *ARCS Workshops*, pages 297–319, 2012.

[153] H. Kalte, G. Lee, M. Porrmann, and U. Ruckert. Replica: A bitstream manipulation filter for module relocation in partial reconfigurable systems. In *Parallel and Distributed Processing Symposium, 2005. Proceedings. 19th IEEE International*, pages 151b–151b, 2005.

[154] Jens Hagemeyer, Boris Kettelhoit, Markus Koester, and Mario Porrmann. Design of homogeneous communication infrastructures for partially reconfigurable fpgas. In *ERSA*, pages 238–247, 2007.

[155] Ali Asgar Sohanghpurwala, Peter Athanas, Tannous Frangieh, and Aaron Wood. Openpr: An open-source partial-reconfiguration toolkit for xilinx fpgas. In *IPDPS Workshops*, pages 228–235, 2011.

[156] Christian Beckhoff, Dirk Koch, and Jim Torresen. Go ahead: A partial reconfiguration framework. In *FCCM*, pages 37–44, 2012.

[157] Christopher Lavin, Marc Padilla, Jaren Lamprecht, Philip Lundrigan, Brent E. Nelson, and Brad L. Hutchings. Rapidsmith: Do-it-yourself cad tools for xilinx fpgas. In *FPL*, pages 349–355, 2011.

[158] Dirk Koch, Christian Beckhoff, and Jürgen Teich. Recobus-builder - a novel tool and technique to build statically and dynamically reconfigurable systems for fpgas. In *FPL*, pages 119–124, 2008.

[159] Christian Beckhoff, Dirk Koch, and Jim Torresen. Automatic floorplanning and interface synthesis of island style reconfigurable systems with goahead. In *ARCS*, pages 303–316, 2013.

[160] Maamar Touiza, Gilberto Ochoa-Ruiz, El-Bay Bourennane, Abderrezak Guessoum, and Kamel Messaoudi. A novel methodology for accelerating bitstream relocation in partially reconfigurable systems. *Microprocessors and Microsystems*, (0):–, 2012.

[161] Christopher Lavin, Marc Padilla, Jaren Lamprecht, Philip Lundrigan, Brent E. Nelson, and Brad L. Hutchings. Hmflow: Accelerating fpga compilation with hard macros for rapid prototyping. In *FCCM*, pages 117–124, 2011.

## Abstract:

The main contribution of this thesis consists on the proposition and development a Model-driven Engineering (MDE) framework, in tandem with a component-based approach, for facilitating the design and implementation of Dynamic Partially Reconfigurable (DPR) Systems-on-Chip. The proposed methodology has been constructed around the Metadata-based Composition Framework paradigm, and based on common standards such as UML MARTE and the IEEE IP-XACT standard, an XML representation used for storing metadata about the IPs to be reused and of the platforms to be obtained at high-levels of abstraction. In fact, a componentizing process enables us to reuse the IP blocks, in UML MARTE, by wrapping them with PLB (static IPs) and proprietary (DPR blocks) interfaces. This is attained by reflecting the associated IP metadata to IP-XACT descriptions, and then to UML MARTE templates (IP reuse). Subsequently, these IP templates are used for composing a DPR model that can be exploited to create a Xilinx Platform Studio FPGA-design, through model transformations. The IP reflection and system generation chains were developed using Sodius MDWorkbench, an MDE tool conceived for the creation and manipulation of models and their meta-models, as well as the definition and execution of the associated transformation rules.



## Résumé :

La principale contribution de cette thèse porte sur la proposition et le développement d'une approche d'Ingénierie Dirigée par les Modèles (IDM), liée à une méthodologie basée sur des composants, pour faciliter la conception, design et implantation des Systèmes Dynamiquement Reconfigurables sur puce (FPGA). La méthodologie proposée repose sur l'utilisation du paradigme Metadata-based Composition Framework, et fortement basée sur des standards, tels qu'UML MARTE et, en particulier, l'IEEE IP-XACT, qui est exploité comme représentation intermédiaire pour les IPs utilisés et pour la plateforme matérielle composé aux hauts-niveaux d'abstraction. Un procès de emballage permet la réutilisation des blocs IP, qui ont été enveloppés par des interfaces PLB (IP statiques) et propriétaires (IP dynamiques). Subséquemment, la libraire est utilisée pour la composition d'un modèle de plateforme en UML, mais qui étant générative, permet la création d'une description cible de la composant matérielle de la plateforme, dans la forme d'un model spécifique à Xilinx Platform Studio, obtenu par des transformations des modèles. Les chaines de transformations pour la création de la libraire et de la plateforme, respectivement, ont été développées et implantées en utilisant Sodius MDWorkbench, un outil IDM conçu pour la création et manipulation des modèles et leur méta-modèles, ainsi que la définition et exécution des transformations des modèles associées.



SPIM

■ École doctorale SPIM - Université de Bourgogne/UFR ST BP 47870 F - 21078 Dijon cedex

■ tél. +33 (0)3 80 39 59 10 ■ ed-spim@univ-fcomte.fr ■ www.ed-spim.univ-fcomte.fr

UNIVERSITÉ DE BOURGOGNE