Model-based design for on-chip systems using and extending Marte and IP-XACT

Aamir Mehmood Khan

UNIVERSITÉ DE NICE-SOPHIA ANTIPOLIS

ÉCOLE DOCTORALE STIC
SCIENCES ET TECHNOLOGIES DE L'INFORMATION ET DE LA COMMUNICATION

T H È S E

pour obtenir le titre de

Docteur en Sciences

de l'Université de Nice-Sophia Antipolis

Mention : Informatique

présentée et soutenue par

Aamir MEHMOOD KHAN

MODEL-BASED DESIGN FOR ON-CHIP SYSTEMS
USING AND EXTENDING MARTE AND IP-XACT

Thèse dirigée par *Charles ANDRÉ* et *Frédéric MALLET*

soutenue le *11 mars 2010*

Jury :

M. Jean-Luc DEKEYSER	Professeur	Rapporteur, examinateur
M. François TERRIER	Directeur de Recherche	Rapporteur, examinateur
M. Charles ANDRÉ	Professeur	Examinateur
M. Frédéric MALLET	Maître de conférences	Examinateur
M. Pierre BRICAUD	Directeur R&D SYNOPSYS	Invité

To Mufti Saeed Khan Sahab and the Family Front.

Acknowledgments

Firstly, I would like to thank Mufti Saeed Khan for all his help and guidance and especially my cousins 'The Family Front' for keeping the fun part of my life alive. I say thanks to HEC (Higher Education Commission) Pakistan for providing me an opportunity to excel in life and serve better my homeland. I also thank SFERE (Société Française d'Exploration des Ressource Educatives) for guiding me and making my stay in France easier. I am thankful to the University of Nice, Sophia-Antipolis for the management of my studies and for the training by the CED (College des Études Doctoral) in which I participated. I pay my gratitude to the members of CIMPACA (Centre Intégré de Microélectronique Provence-Alpes-Côte d'Azur) design platform for providing an access to the DesignWare IP Room and all other relevant tools. Especially I would like to thank Mr. Pierre Bricaud for kindly arranging special lectures on Synopsys Tools.

I say thanks to INRIA Sophia-Antipolis for welcoming me and for having provided me with all its resources. Thanks to our project assistant, Patricia Lachaume for her help and advice in administrative and daily life. I express gratitude to each member of the AOSTE project for the time spent with them and for their support. I thank Benoît Ferrero for helping me on the technical aspects of my work. I thank Jean-Francois Le Tallec for the discussions and suggestions regarding SystemC and TLM. I thank Julien Boucaron, Anthony Coadou and Luc Hogie for having lively discussions. I thank Julien DeAntoni and Marie-Agnès Peraldi Frati for giving me their valuable suggestions at the meetings. I say thanks to my friends Uzair Khan, Sheheryar Malik and Shafqat-ur-Rehman for having valuable discussions during the coffee breaks. I thank Najam-ul-Islam for helping me in every aspect of my life and keeping a warm company with his jolly nature.

And especially, I would like to thank Charles André, Frédéric Mallet and Robert de Simone, my thesis supervisors, for helping, guiding and supervising my work. They are my best mentors in all aspects of my studies and research work. I am grateful to Jean-Luc Dekeyser and François Terrier for being my thesis reviewers. Thank you both for your feedback and the corrections you suggested. Your constructive criticism was really helpful.

Abstract

On-chip systems (also known as System-on-chip or SoC) are more and more complex. SoC design heavily relies on reuse of building blocks, called IPs (Intellectual Property). These IPs are built by different designers working with different tools. So, there is an urgent demand for interoperability of IPs, that is, ensuring format compatibility and unique interpretation of the descriptions. IP-Xact is a *de facto* standard defined in the context of electronic system design to provide portable representations of (electronic) components and IPs. It succeeds in syntactic compatibility but neglects the behavioral aspects. UML is a classical modeling language for software engineering. It provides several model elements to cover all aspects of a design (structural and behavioral). We advocate a conjoint use of UML and IP-Xact to achieve the required interoperability. More specifically, we reuse the UML Profile for MARTE to extend UML elements with specific features for embedded and real-time systems. MARTE Generic Resource Modeling (GRM) package is extended to add IP-Xact structural features. MARTE Time Model extends the untimed UML with an abstract concept of time, adequate to model at the Electronic System Level.

The first contribution of this thesis is the definition of an IP-Xact domain model. This domain model is used to build a UML Profile for IP-Xact that reuses, as much as possible, MARTE stereotypes and defines new ones only when required. A model transformation has been implemented in ATL to use UML graphical editors as front-ends for the specification of IPs and to generate IP-Xact code.

The second contribution addresses the modeling of the IP time properties and constraints. UML behavioral diagrams are enriched with logical clocks and clock constraints using the MARTE Clock Constraint Specification Language (CCSL). The CCSL specification can serve as a "golden model" for the expected time behavior and the verification of candidate implementations at different abstraction levels (RTL or TLM). Time properties are verified through the use of a dedicated library of observers.

Résumé

Les Systèmes sur puce (SoC) sont de plus en plus complexes. Leur conception repose largement sur la réutilisation des blocs, appelés IP (Intellectual Property). Ces IP sont construites par des concepteurs différents travaillant avec des outils différents. Aussi existe-t-il une demande pressante concernant l'interopérabilité des IP, c'est-à-dire d'assurer la compatibilité des formats et l'unicité d'interprétation de leurs descriptions. IP-Xact constitue un standard *de facto* défini dans le cadre de la conception de systèmes électroniques pour fournir des représentations portables de composants (électroniques) et d'IP. IP-Xact a réussi à assurer la compatibilité syntaxique, mais il a négligé les aspects comportementaux. UML est un langage de modélisation classique pour le génie logiciel. Il fournit des éléments de modèle propres à couvrir tous les aspects structurels et comportementaux d'une conception. Nous prônons une utilisation conjointe d'UML et d'IP-Xact pour réaliser la nécessaire interopérabilité. Plus précisément, nous réutilisons le profil UML pour MARTE pour étendre UML avec des caractéristiques temps réel embarquées. Le paquetage Modélisation Générique de Ressources de MARTE est étendu pour prendre en compte des spécificités structurelles d'IP-Xact. Le Modèle de temps de MARTE étend le modèle atemporel d'UML avec le concept de temps logique bien adapté à la modélisation au niveau système électronique.

La première contribution de cette thèse est la définition d'un modèle de domaine pour IP-Xact. Ce modèle de domaine est utilisé pour construire un profil UML pour IP-Xact qui réutilise autant que possible les stéréotypes de MARTE et en définit de nouveaux uniquement en cas de besoin. Une transformation de modèle a été mise en œuvre dans ATL permettant d'utiliser des éditeurs graphiques UML comme front-end pour la spécification d'IP et la génération des spécifications IP-Xact correspondantes. Inversement, des fichiers IP-Xact peuvent être importés dans un outil UML par une autre transformation de modèles.

La deuxième contribution porte sur la modélisation de propriétés et de contraintes temporelles portant sur des IP. Les diagrammes comportementaux d'UML sont enrichis avec des horloges logiques et des contraintes d'horloge exprimées dans le langage de specification de contraintes d'horloge (CCSL) de MARTE. La spécification CCSL peut alors servir de « modèle de référence » pour le comportement temporel attendu et la vérification des implémentations à différents niveaux d'abstraction (RTL ou TLM). Les propriétés temporelles sont vérifiées en utilisant une bibliothèque spécialisée d'observateurs.

Contents

Chapter 1

Introduction

In our daily lives, miniaturization and the growing complexity of electronic systems has resulted in a generalization of the use of on-chip systems. This growing complexity is driven by the consumer demands and the ability of the modern techniques to address complex physical hardware. *On-chip system* (also called system-on-a-chip or SoC) refers to integrating all components of an electronic system into a single integrated circuit (an IC chip). It is a blend of software and silicon hardware components intended to perform predefined functions in order to serve a given market. The distinctive feature of these systems is to be concentrated on a single block, the chip, to provide the maximum functionality. These SoC designs are very diverse, consisting of multiple design domains (hardware, software, analog), multiple source components (Core IPs, DSPs, ASICs, *etc.*) and have diverse constraint limitations (real-time, low power, cost efficiency, *etc.*). All these things make the SoCs very complex. This growth of complexity has been made possible by the unrelenting progress of technology integration of transistors on a chip. In short, a SoC is a complete system which would have been assembled on a circuit board just a few years back, but now can fit entirely in a single chip.

With the exponential growth of system complexity, designing systems at lower levels has become more difficult. So the focus of SoC designers has shifted to more abstract representations. The traditional approach of designing chips, their functionalities, implementation, and verification techniques does not follow the same innovative techniques to match the growth of the SoC complexity. This complexity can only be addressed by modeling at higher level and relying on greater component reuse. Many research initiatives are destined to address these issues to increase productivity techniques and innovations to meet the economic and technical constraints in demand.

1.1 Issues Addressed

SoC designs heavily rely on the reuse of building blocks, called IPs (Intellectual Property). These IPs are built by different designers working with different tools. Reuse and integration of these heterogeneous IPs from multiple vendors is a major issue of SoC design. The attempt to validate assembled designs by global co-simulation at the implementation level is doomed to failure because of the increasing complexity and size of actual SoCs. Thus, there is a clear demand for a multi-level description of SoC with verification, analysis, and optimization possibly conducted at the various modeling levels. In particular, analysis of general platform partitioning, based on a coarse abstraction of IP components, is highly looked after. This requires interoperability of IP components described at the corresponding stages, and the use of trace-

ability to switch between different abstraction layers. Although this is partially promoted by emerging specifications (or standards), it is still insufficiently supported by current methodologies. Such specifications include SystemC [IEE05], IP-Xact [SPI08], OpenAccess API [GL06], and also Unified Modeling Language (UML [OMG07]) based specifications like the UML Profile for Modeling and Analysis of Real-Time and Embedded systems (MARTE [OMG08c]) that specifically targets real-time and embedded systems.

System Modeling requires representation of both structural aspects at different levels of abstraction as well as functional aspects possibly considering time-related viewpoints such as untimed, logical synchronous, or timed models. So the focus areas of our work are both architectural and behavioral aspects of IPs.

For *system architecture* representation, UML uses class, component, and composite structure diagrams, while SysML [Wei08] (a UML profile for systems engineering) uses block diagrams. UML is a classical modeling language for software engineering. It provides several model elements to cover all structural as well as behavioral aspects of a design but it contains no specific constructs for modeling IPs. Tools like Esterel Studio, and virtual platforms like CoWare, Synopsys CoreAssembler and ARM RealView, introduce their own architecture diagrams, resulting in non-interoperable models. On the other side, IP-Xact relies on XML schemas for specification of IP meta-data and tool interfaces. It has become a *de facto* standard in the context of ESL design. It presents some ADL (Architecture Description Language) features for the declaration of externally visible design interfaces and the component interconnections. It provides portable representations of electronic components and IPs, thus ensuring a kind of syntactic compatibility. It also describes book-keeping information (ownership, versioning, tool chain used, *etc.*) and memory mappings, useful in deployment and simulation. However, IP-Xact scatters information, with data duplication sometimes over various locations. This makes IP-Xact difficult to handle manually. Moreover, it also lacks abstract representation of IPs, like the Communication Processes (CP) level. To sum up, we are still looking for a standard, covering all the aspects of the structural description of IPs.

For *component behavior* representation, we consider various abstraction levels discussed by Gajski [CG03] like CP (Communicating Processes), TLM (Transaction Level Modeling) with its sub-levels PV (Programmers View) and PVT (Programmer View with Time), CC (Cycle Callable), CA (Cycle Accurate), and RTL (Register Transfer Level). IP-Xact provides support for the behavior representation at low abstraction level (RTL), and medium level (TLM/PV, TLM/PVT, CC) through SystemC and VHDL files. These languages provide support for the timed modeling but are not used for abstract levels (CP). On the other hand, UML state-machines, sequence and activity diagrams can be used for modeling at abstract levels like CP (Communicating Processes). But UML lacks the notion of time required to model timed abstract systems. MARTE could be seen as introducing the relevant *timed* version at this level through logical time and abstract user-defined logical clocks.

1.2 Proposed Approach

In this thesis, we are addressing the issues of integration and interoperability between the various building blocks (IPs) of a SoC both at structural and functional levels.

The first contribution of this thesis is the definition of an IP-Xact domain model (meta-model). We propose a joint use of UML and IP-Xact, targeting a better integration of both specifications. By this effort, we are able to utilize the graphical and tool capabilities of UML (graphical modular designing, component reuse, different representation views, etc.), jointly with IP-Xact. For this purpose, we partially extend the UML profile for MARTE with IP-Xact-specific stereotypes. Relying on a profiling approach allows easy creation and extension of

model editors for IP-Xact based on existing UML graphical editors (*e.g.*,Eclipse UML, MagicDraw by NoMagic, Rational Software Architect by IBM, Artisan, Papyrus, ...). Selected UML structural models are extended with IP-Xact capabilities and UML behavior models complement the current IP-Xact/SystemC specifications. MARTE time model adds the necessary abilities to specify time requirements. This combined approach allows us the use of IP-Xact in a more abstract-level modeling environment. A transformation engine is built to export models to IP-Xact-dedicated tools. We chose to do this by specializing the MARTE profile, which already provides a number of modeling features for extra-functional aspects and for introducing logical time in the UML. We only define new stereotypes when nothing equivalent exist either in standard UML or in MARTE.

The second contribution of our research work addresses the modeling of time properties and constraints for the IPs. For *component behavior* representation, we enrich the UML behavioral diagrams with the MARTE time model and its associated clock constraint specification language (CCSL) to attach time/behavioral information to an IP structural representation. Our effort related to time information does not attempt to represent the behavior in its entirety but focuses on the IP timing properties extracted directly from the IP specification (datasheets). Expected time properties are expressed in CCSL. Such CCSL specifications then serve as the reference model for the expected time behavior and drive the verification of IPs. These properties are verified with CCSL observers for different implementations of the IP at various abstraction levels, to give some functional equivalence. As an example, we include the time information extracted from the IP datasheets and show how this information can be used to generate test-benches tailored for the different abstraction levels to test the desired IP properties.

1.3 Document Organization

This document is organized into three main parts. The first part consists of chapters 2, 3, and 4. It introduces the basic concepts involved in SoC design flow and model driven engineering (MDE). This introduction prepares a ground for discussing the more advanced issues. Thus, in chapter 2 we compare the traditional and a current approach for the design of SoCs. Later on, we review some of the languages used at the various abstraction levels for SoC designs. Readers from the EDA community can comfortably skip this chapter. In chapter 3, we focus on the classical model-driven approach. This includes a view on the use of models and metamodels in general, and the UML in particular. We introduce the profiling mechanism of UML and a discussion follows on the popular UML profiles related to this work. We then propose a transformation of UML models into IP-Xact models empowering us with the automation of design process. Readers familiar with model-driven engineering can skip these details. Finally in chapter 4, we introduce a simplified computer system example starting from high-level models and refining it to more concrete implementations. This example covers almost all the aspects of our research work. It functions as a running example for this contribution and is often referred to in the following chapters.

Second part of the thesis relates to our contribution regarding the description, integration and interoperability of structural building blocks of SoCs. It consists of chapters 5 and 6. Chapter 5 introduces our representation of the domain view of IP-Xact specification. This representation is then utilized in chapter 6 for creating the UML profile for IP-Xact based on the existing UML profile for MARTE. Later we model the Leon II system architecture in UML using this newly introduced profile.

In the last part of the thesis, we present our contribution to time behavioral aspects of IPs. This part consists of chapters 7 and 8. In chapter 7, we discuss the behavioral representation of our running example at various abstraction levels. This is followed by an introduction

to the clock constraint specification language (CCSL) and on suggestions to integrate this information into the IP-Xact specification. In chapter 8, we introduce the implementation of CCSL constraints in VHDL language which are then used to test and verify the selective properties of Leon II-based embedded system architecture.

The last chapter of this work concludes by recalling the main points of our contribution, the results of using our models with the tools from Synopsys coreAssembler and Innovator, and by providing some perspectives for future works.

Chapter 2

Electronic Systems Design

Contents

*Electronic System-Level (*ESL*) design is the use of appropriate abstraction layers to maximize the understanding of the system while reducing the development time and cost to the system. On the one hand this approach focuses the concurrent development of software/hardware design while on the other hand it introduces abstract modeling layers like transaction level modeling (*TLM*).*

In this chapter we explore the different concepts related to the ESL *design including SoC design flow and different abstraction layers. These concepts are then utilized in the structural and behavioral modeling of embedded systems. In the last part, we explore three families of languages (synchronous languages, system-level languages, and hardware description languages) that are used in the other chapters of this thesis for architectural and behavioral descriptions.*

2.1 New trends in Electronic Systems Design

Electronic systems are increasingly present in our daily lives. Advances in miniaturization allow chip design carrying a continually increasing number of features. Simple mobile phones of the past are now increasing having multiple features like advanced communications like GSM and 3G, camera, video recording, touch sensitive menus, Internet and wireless access, or GPS navigation system, all integrated into a single chip. These all features are associated with a processing chip lying at the center of any electronic device. This chip is an equivalent of the old designs of dispersed devices connected through the buses on the motherboard. Such a chip known as *System-on-a-Chip* or SoC is the brain of most the embedded devices designed these days.

The doubling of the complexing of systems on chips, approximately every two years as predicted by Moore's Law, has given us systems that are increasingly sophisticated, and are at the same time more and more difficult to conceive. On the other hand, the productivity of a system designer/developer was comparatively slow. Thus, there is a growing gap between the evolution of the physical chip designs and the software support for them, termed as the *design gap*. This problem gave way into finding the new trends focused on system design methodology.

Daniel D. Gajski has discussed, in his paper [Gaj07] and a panel discussion [SSS+03], about the strategies for dealing with the rising Systems-on-Chip (SoCs) complexity by using the higher abstraction levels. An abstraction level is a means of addressing the inability of the human mind to totally comprehend a complex system at a very detailed level [BMP07]. Thus we increase the size of the basic building blocks that are used in our designs. Presently used complex SoCs cannot be properly represented in the traditional register transfer level based methodologies.

In this chapter, we firstly present the design flow for the realization of Systems on Chips. Later, we explore in detail the major stages of this design flow including Register Transfer Level (RTL) and Transaction Level Modeling (TLM). Finally we try identify their shortcomings and the areas of possible improvement.

2.2 Systems on Chip (SoCs) Design Flow

2.2.1 Traditional/Classical Design Flow Approaches

Frank Ghenassia in his book [Ghe05] states that the design flow is a rigorous engineering methodology or process for conceiving, verifying, validating, and delivering a final integrated circuit design to production, at a precisely controlled level of quality. Moreover, he gives also a look into the traditional design flow of an embedded system as shown in figure 2.1. This design flow starts with the design specification given by the client. Based on this specification two separate developments of the hardware and software begins independently. Note that the two developments take place totally independently without any regard of others' existence. The two designs are kept separate until the prototype of the system-under-design is ready to be tested.

In a traditional design flow, the hardware design development usually begins with the creation of hardware models using hardware description languages (HDLs) like VHDL or Verilog. These models are then simulated for functional verification or correctness of the behavior of the module. Finally, synthesis is performed to obtain the code netlist. Later this design is laid on the chip to build our basic prototype. On the other hand, the software development goes independently without any knowledge about the hardware design. Although the software

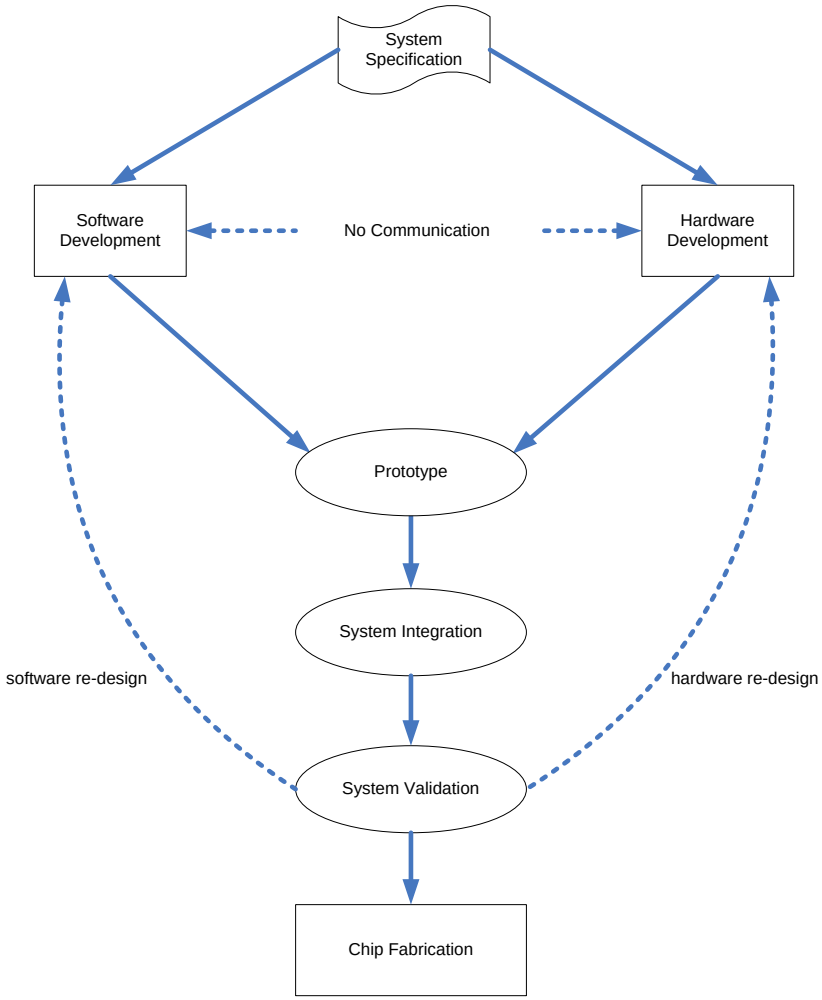


Figure 2.1: Traditional SoC Design Flow.

coding starts quite early in this way, but the testing of this code can only happen when the hardware design prototype is available.

One of the biggest disadvantages of the traditional design flow approach is the system validation at quite late stages of the design. Until the system prototype is ready and simulated along with the software design, we cannot validate the system. This problem is augmented with the fact that the software and hardware designs are modeled independently and without co-ordination. Hence, when at the later design stages a problem evolves, it costs more to correct it and re-design the system prototype. Moreover, the simulation speed for the designs at this level is very slow making the task of validating huge designs to be very long.

A new and classical system level design flow approach is shown in the figure 2.2. This classical design flow differs from the traditional system design flow in many ways. The traditional design flow uses a single system specification to describe both hardware and software. On the other hand, classical design flow models use the Y-chart methodology [Erb06] for the

hardware/software partitioning. This methodology uses a clear separation between an algorithmic/application model, an architecture model and an explicit mapping step to relate the algorithmic model to the architecture model. The application model describes the functional behavior of an application independent of architectural specification whereas the architecture model defines the system architecture characteristics. Thus, unlike the traditional approach in which hardware and software simulation are regarded as the co-operating parts, the Y-chart approach distinguishes algorithm/application and architecture simulation where the latter involves simulation of programmable as well as reconfigurable/dedicated parts [Erb06]. Moreover, in the traditional design flow, the system validation is not possible until the design prototype is ready whereas in the present time classical modeling approaches, the system can be hardware/software co-simulated just after the initial design specification phase. This co-simulation [Sch03] is absolutely necessary for the software development such that the software design teams can have an idea of the underlying hardware resources to be utilized efficiently. In the traditional approach, the software development can only proceed when the hardware development phase has completed whereas in the current practice, hardware and software are partitioned to co-simulate at the same time. Partitioning is the process of choosing what algorithms (or parts thereof) defined in the specification to implement in software components running on processors, what to implement in hardware components, and the division of algorithms within the software and hardware components [BMP07]. There are limitations for this as the developed code depends on the underlying hardware used. For this purpose, techniques have been developed to enable the software to run on *virtual hardware models* or *virtual platforms* like the ones using transaction models (discussed in detail later). This approach dramatically reduces the Time-to-Market (TTM) duration due to reduced number of errors and early system validation. *Time-to-Market* is the time duration that it takes for a system chip design to be materialized and introduced to the market and is an important index to judge the performance of design techniques used.

2.2.2 Stages of SoC Design Flow

A classical system level design flow (shown in the figure 2.2) can be divided into three main phases: system specification phase, architecture exploration phase and the system implementation phase, all discussed next.

System specification

In the *system specification* phase, we identify the requirements that the chip has to fulfill. SoC provides a variety of features on a single chip including interaction with the other component modules and with the user, which makes it quite complex. The SoC specification includes the desired requirements from the end-user and the specific constraints being enforced by the surrounding environment. These constraints can be of a vast variety. For instance, for a multimedia application, the loss of data is not as important as the jitters and the delay in the communication whereas for critical embedded applications like in avionics, the systems have to be real-time as well as accurate in data. All such constraints are taken into account while designing a chip.

Architecture exploration

System specification phase leads to the *architecture exploration phase* during which the system architect determines the hardware configuration that will be necessary to meet the needs

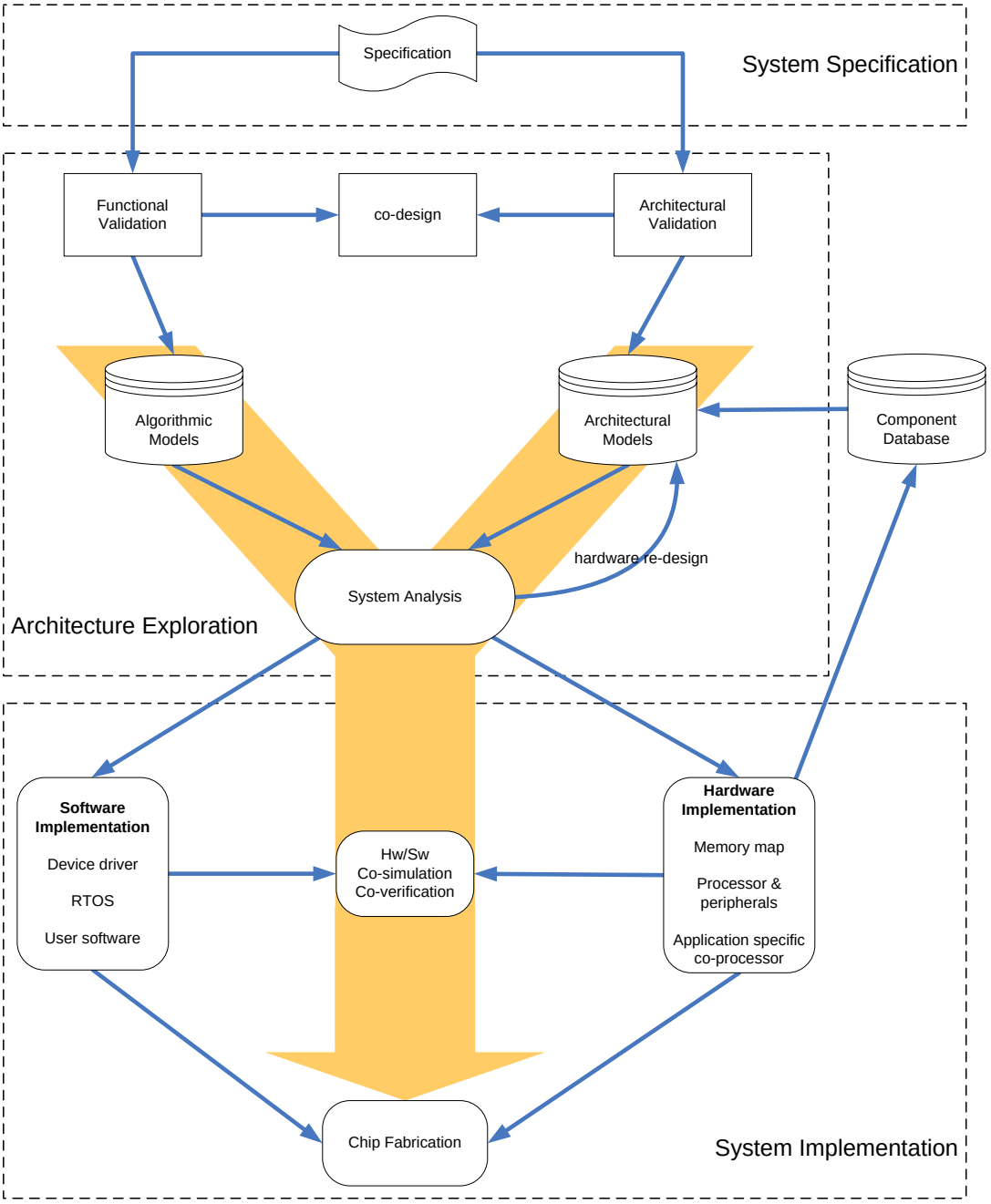


Figure 2.2: Classical System Level Design Flow.

expressed before. This step is broken down into two parts: functional modeling phase and system analysis phase.

During the *functional modeling phase*, different executable models for the hardware and

the software of the system are designed. Usually hardware modeling deals with the resource allocation while software modeling includes the definition of key functions, programs and routines. These models identify the key building blocks of the system and their interconnections. They provide the possibility of an early functional and architectural validation of the system, co-designing hardware and software models. These models also give us a big picture about the requirements and interdependence of system modules leading to early detection of logical system errors. The algorithmic and architectural models used for the creation of functional model are rarely rebuilt from scratch in each new chip but are usually reused from an IP library. Here, a standard like IP-Xact [SPI08], which is at the heart of this thesis, plays a key role by providing component IP databases. In fact the blocks which constitute a system on chip are generally created to be reused and usually a new system design architecture reuses and integrates these components, leading us to the concept of *platform based systems* [SVCDBS04].

System analysis phase decides the low level details about the system design. This step crucially determines the remaining design flow as by this time the system architect must determine how the chip will be incorporated to meet the requirements specified in the previous phase. A key parameter for this determination is the hardware/software partitioning. At the end of this phase we obtain a hardware/software partitioned system that has an adequately allocated proportion of design functionality between the hardware and the software. After this phase our system architecture is finalized and we can proceed with the system implementation phase. As an example we consider a functional model of a system running some specific programs. There are several ways to execute this specific program on the given model. In one approach, we can assign the task to a processor and hence algorithm is implemented in the software. This approach is quite easy and offers great flexibility as we can alter the program (to a limit) even after the hardware fabrication. However, if the executed program is too complex, the processors are not an efficient choice in terms of time and power consumption. This leads us to the second approach of creating ASICs (Application Specific Integrated Circuits) which costs us the flexibility to modify but are very efficient in terms of energy and time consumption leading to shorter time-to-market durations. A third approach placed in between these two is of the use of FPGAs. FPGAs are fully customizable hardware units and give more flexibility for hardware design as compared to ASICs. The system analysis process finally leads us to the determination of a system whose software has been specified as well as components that will execute it. This gives way to the development of respective software or hardware implementations in the next phase.

System implementation

Following the system analysis, we have the *system implementation phase* after which we have the software and hardware implementation models of the system.

In the present times, the *software implementation* for a SoC design has become a pivotal part for the overall system development. Software development includes variety of programming including the low level device drivers coding, the operating system development and management, and finally the problem specific end-user application. With the current trends of evolution of system-on-chip architectures, having generalized multiprocessor system designs, the importance of implementation software augments. The device drivers and operating systems are the softwares that are directly dependent on the underlying hardware and make the application programming simple and hardware independent. Software programming heavily depends on the system requirements specification. On one side a simple SoC design can have a naive software implementation whereas on the other side if the software is implemented for the critical embedded applications like in avionics or life saving devices, its correctness and proper functioning matters a lot. Various design techniques have emerged in the EDA industry

to deal with the verification and validation of such software implementations.

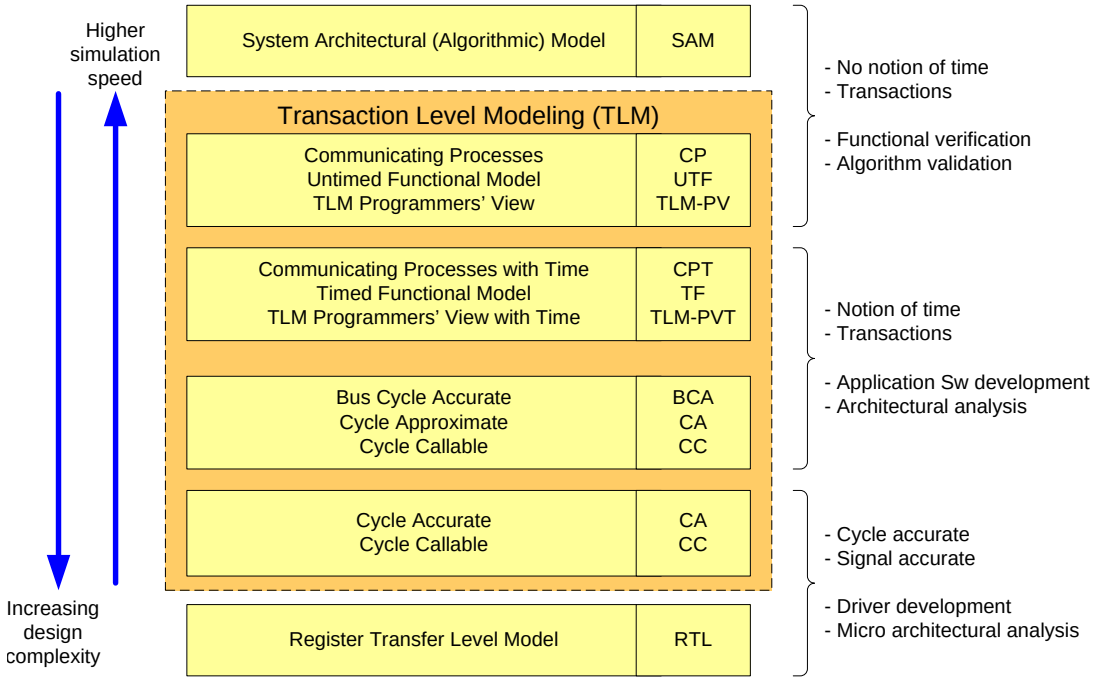


Figure 2.3: ESL System Design Layers.

The *hardware implementation* is the process of creating models that may be synthesized into gate-level models. These models of a system design can be represented at different modeling levels. Every design engineer can have his own view of the abstraction level of system design. There can be potentially hundreds of abstraction levels for an electronic system design, but mainly we focus on three commonly used abstraction levels [Rev08] namely *System Architectural Model* (SAM), *Transactional Level Modeling* (TLM), and *Register Transfer Level* (RTL) (as shown in figure 2.3). From a system designer's view point, the design starts from a model specification. *System Architectural Model* comprises of design specification and its algorithmic model. The *design specification* is the system requirement usually documented in a textual language. Then this requirement specification is converted into a simple model comprising of functions. Both these representations give the broad picture of the system representation and are usually not meant for simulation purposes. These representations are characterized by untimed transactions and can be used for high level system validation purposes. At the higher levels of abstraction, the range of modeling levels is coarsely defined and overlaps with the adjacent levels. Transaction Level Modeling (TLM) is an abstract modeling level providing the initial executable model of the system, hence leading to early system simulation and validation. While most of these levels are optional and are only there to facilitate the design flow, the last RTL model is mandatory. RTL models give the most precise way to represent the components present on the chip, and which can be automatically synthesized into a logical combination of elementary operations (logic gates) whose implementation is materialized through the transistors and integrated circuits (ICs) engraved on the circuit chips. These different modeling levels demand different models and languages. We present three families of languages addressing the

various levels.

2.3 Synchronous Languages

Most electronic systems are *reactive* and present to some extent *real-time* features. Synchronous languages have been developed to deal with reactive systems. We present them in this chapter on electronic systems for three main reasons:

- the importance of electronic systems in real-time and embedded applications;
- the intrinsic interest of the synchronous paradigm and its relationship with synchronous circuits;
- the recent evolution of some synchronous languages, like Esterel, to electronic system design.

2.3.1 Reactive and real-time systems

Reactive systems interact with the real-world environment by reading the sensors or receiving interrupts and produce the output commands/information. For correct functionality they have to produce valid data under strict time constraints (deadlines) hence leading to both logical and temporal correctness of real-time systems [BB91]. Safety is a critical issue for many real-time systems with numerous human lives at stake. The behavior programming in general and especially of such systems can be done using combination of finite state machines (FSMs). StateCharts [Har87], which are hierarchical and concurrent state-transition models, can be used instead of a collection of FSMs. Both FSMs and StateCharts are normally implemented in HDLs or C language. Such low-level languages do not help much to deal with the complexity of the systems and have limited component reuse, difficult to debug, and error-prone (with error detection at late design stages). Moreover, these languages are not adequate for high level design for which system-level design languages like SystemC [IEE05] and SpecC [DGG02] were introduced lately. These system-level languages facilitated the hardware/software co-design and co-simulation leading to early system validation. However they lack a formal semantics and hence are not a good choice for the formal verification and the correct-by-construction [Ber07] implementation of systems.

Computer systems broadly fall into three categories based on their behavior; transformational systems, interactive systems, and reactive/reflex systems [Hal92, chapter 1]. Example of *transformational systems* is the compiler which takes input code and generates an output targeted for a specific architecture. The *interactive systems* are the ones which intake several input stimuli and produce various output signals according to the input signals, without any regard for the time taken to process this information. This means that the environment/client does not know when the output will be available. On the other side, the *reactive systems* are the ones in which correct functioning of the system not only depends on processing the input but also on the timely response of the system (just like our reflexes). Hence the interactive systems are non-deterministic while the reactive systems are predictable. In another way, we can say that the interactive systems have got their own pace of execution while interacting with the environment whereas the reactive systems theoretically react instantly to the inputs and their pace of execution is dictated by the environment. All these types of computer systems are shown graphically in the figure 2.4.

The behavior of a reactive system is usually described as a sequence of reactions, resulting from *execution cycles* in which (sequentially) it reads inputs from the sensors, processes that

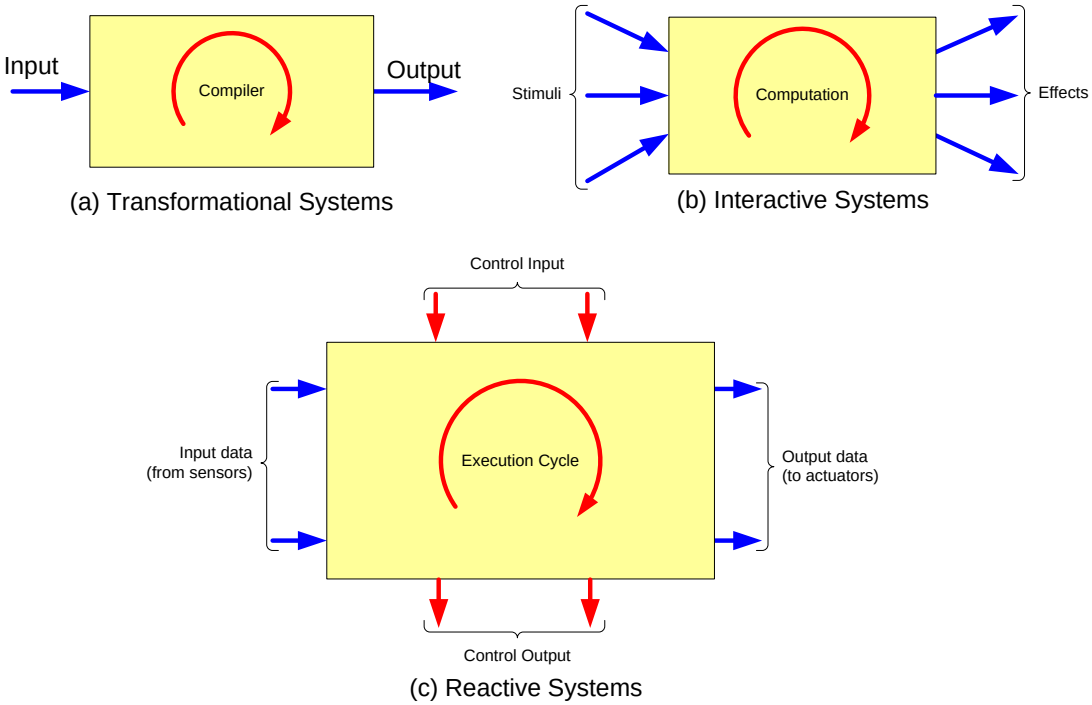


Figure 2.4: Types of Computer Systems.

input data based on the control commands, and finally generates output to manipulate the actuators. The reactive systems are generally implemented by finite state machines (FSMs). Traditionally, the tools to design reactive systems forced the users to choose between the determinism and concurrency for a system. Contrary to the traditional programming languages, *synchronous languages* prove to be the good candidate for the programming of reactive/real-time systems. These languages provided the system designers with the ideal primitives to deal with the issues related to reactive systems design. Synchronous Languages were initially introduced in the beginning of 1980s [1]. They are based on the *perfect synchrony hypothesis* [Ber00b], in which concurrent processes are able to perform computation and exchange information in zero time. Such a synchronous model has a great appeal in the real-time modeling domain including embedded systems (avionics, SoC design), communication systems and hardware design. These languages can be classified as declarative languages and imperative languages, based on their programming style. *Declarative synchronous languages* adopt a data-flow style. They express the logic of a computation without explicitly describing its control flow. They focus on 'what' the program should accomplish. Synchronous programming languages like *Lustre* [HCRP91] and *Signal* [LGLBLM91] represent this category. The former is a synchronous flow language that relies on equations; the latter is more general and considers relations. Both can be efficiently used for steady-process control and signal processing applications. In the contrast, *imperative synchronous languages* describe (reactive) behavior in terms of statements that change a program state; they focus on the 'how'. *Esterel* [BdS91, Ber00b] and *SyncCharts* [And96] represent such languages. They are better used in control-intensive

[1] http://www.esterel-technologies.com/technology/history/

applications. Recently, developments in the synchronous language tools have made this classification a bit vague. For instance SCADE, a Lustre-based industrial synchronous methodology and tool-set [2], has integrated SSMs (Safe State Machines, a variant of SyncCharts) which are control-oriented, and Esterel (version 7) now supports Lustre-like equations.

The synchronous languages provide the formal semantics as well as the abstract representation of designs at system level which makes them a good choice for behavior representation. The imperative programming languages like Esterel also facilitate the designing of FSMs, discussed in the next subsection.

2.3.2 Esterel Language

In our work, to program the reactive applications and FSMs, we have used Esterel Language along with SyncCharts. These applications can be represented using textual (Esterel) as well as graphical (SyncCharts) forms. *SyncCharts* is a graphical language expressing hierarchical and concurrent state machines. SyncCharts is a model akin to StateCharts. However, they differ on their semantics: SyncCharts adheres to the perfect synchrony hypothesis while StateCharts has many synchronous but not perfect synchronous semantics [vdB94, HN96]. Because SyncCharts and Esterel share a common semantics, they can be freely mixed in a specification. In the Esterel Studio Suite, a syncChart is first translated into a semantically equivalent Esterel program, and then treated as an ordinary Esterel program. In the following, an Esterel program stands for its textual as well its graphical format. The compilation chain of Esterel can generate program codes in languages like C, VHDL, or SystemC. These programs have a behavior equivalent to the one specified by the Esterel program, but concurrency has been compiled out.

The early versions of the Esterel language [BdS91] (in the 80's and 90's) targeted reactive systems and safety-critical systems without special consideration for electronic systems. Since 1999, within an industrial context, the Esterel compiler has undergone broad syntactic extensions, leading to Esterel v7 [Est05], and its main application domain has become electronic system design. To understand this evolution we have to briefly describe the basics of Esterel and bring out the main changes in the language.

Esterel v5

The syntax and an informal semantics of Esterel v5 are presented in a language primer [Ber00a]. A book [PBEB07] describes formal semantics of the language and different compilers. An Esterel program can be seen as specifying a collection of communicating threads running concurrently. Here, concurrency should be understood as cooperation, not competition as it is often the case in interactive systems. Note that most of the Esterel compilers generate a sequential executable code; thus the threads we consider are logical or conceptual, not physical (run-time threads).

The simplest way to understand the behavior of a synchronous program is to consider that each thread performs a cycle-based computation in which environment actions strictly alternates with thread actions. The many threads that constitute the program communicate and interact during their active phase. These concurrent/cooperative evolutions define an *instant*. All the actions and the information exchanges are *simultaneous* (*i.e.,*at the same instant).

In Esterel, *signals* are the only means of communication and provide the unique support for communication between threads. When a thread *emits* a signal, this signal is instantly broadcast and thus can be seen by any thread that may be concerned. A signal is either

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

present or absent at a given instant. *Pure* signals carry only this presence information. *Valued* signals also convey a typed value.

To emit or test a signal is a reactive statement in Esterel. Preemptions are other typical reactive statements. *Preemption* is the capability for a thread to block the execution of another thread. The preemption can be temporary (*suspension*) or definitive (*abortion*). The simplest form of abortion is the `await` statement which makes a thread wait for the next presence of a signal.

Syntactically, an Esterel program consists of one or several modules. A *module* is a program unit that has a declarative part (*header*) and an imperative part (*body*). The header specifies the interface of the module: it declares signals with their direction (input, output, inputoutput) and their type. The body contains executable statements and structured blocks of statements that can be nested and composed in parallel or in sequence. Modules can also be instantiated in a module, allowing hierarchical specifications. *Local signals* declared within the body of a module serve as communication medium for its concurrent threads.

The semantics of Esterel guarantees that preemptions at any depth and any degree of concurrency are always deterministic. The unicity of each signal status (presence and value, if any) is also assured at each instant. This makes Esterel an outstanding language for programming complex control-dominated systems that require fully predictable behavior.

Note that Esterel, like the other synchronous languages [Hal92], relies on a *logical time* rather than *physical time*. Only the ordering of event occurrences is meaningful, not the "physical" duration between them. An event can occur before or after another one, but it is also possible for them to occur simultaneously. In contrast to non-synchronous models, the notion of simultaneity is precisely defined in synchronous languages and characterizes things occurring at the same instant.

The compilation of an Esterel program is a hard task [Ber00b, PBEB07]. Each signal must respect two *coherence rules*:

1. a signal is only present or absent in an instant, never both;
2. during a reaction, for any output or local signal, all the emit actions must precede any test action.

The latter rule is a form of causality relationship not easy to check on complex programs. The *constructive semantics* has defined simple correctness criteria for causality. A second difficulty came from the size of the generated code when FSMs were taken as target representation. In the mid 90's appeared the *circuit semantics* of Esterel. This semantics translates Esterel programs into sequential circuits, or equivalently into logical equations, similar to the ones found in Lustre programs. This logical representation is amenable to powerful optimizations that have been extensively studied in the hardware community. Direct implementations in hardware and software can be generated from the optimized circuits. The close relationship between the Esterel semantics and the circuits is probably one of the main reasons why Esterel has successfully evolved to Esterel v7 which is equipped to address electronic design.

Esterel v7

Esterel v5 failed at modeling/programming large-scale system-level or hardware designs. It lacked prominent features [BKS03] in its ability to describe data paths, to deal with bit-vectors, to directly support Moore machines, etc. Esterel v7 has corrected these deficiencies by syntactic improvements that have left the semantics of the language mostly unchanged.

For our usage of Esterel in this thesis, the most interesting changes are in the declarative part. Three types of program units are now available:

- *Data unit* which declares data types, constants, functions, or procedures. It can *extend* one or several other data units by importing all the objects they declare.
- *Interface unit* which primarily declares input/output signals and can also *extend* other data and interface units by importing all their data and signal declarations. Interface units can also be declared as *mirror* of some other interface which means that the interface type is the same, but the interface signal directions are reversed.
- *Module unit* that defines the reactive behavior of the system. The header of the module can extend data and interface units by importing the objects they declare.

Units can be generic, which allows better reuse. Also of interest is the introduction of *port*. A port is a group of signals typed by an interface. A port allows renaming of signals in interface instantiations. Esterel v7 supports *arrays*. Ports and signals can be declared as arrays. The array mechanism facilitates the specification and the handling of large structures like buses in computer architecture.

Modern hardware designs usually have more than one clock. Moreover, for power saving, these clocks can be turned on and off. To address these new design challenges, Esterel v7 supports *multi-clocking* and adopts the GALS (Globally Asynchronous Locally Synchronous) design paradigm. A new kind of signal (`clock`) and a new kind of unit (`multi-clock`) have been added to the language. A multi-clock unit is similar to a module unit but it declares one or many clocks in its header. It can also define local signals and clocks. Several module and multi-clock units can be instantiated in a multi-clock unit. The classic module instances perform computations, usually driven by an explicit clock, while the enclosing multi-clock unit only deals with clocks and signals.

All the correct Esterel programs are synthesizable in hardware and software, including multi-clock designs. Moreover, the programming structure of Esterel v7 resembles more to the HDLs. So, the advent of Esterel v7 has greatly improved the capabilities of the language for use in EDA (Electronic Design Automation) applications. Besides all these language improvements, Esterel Studio now supports *architecture diagrams*, a graphical notation similar to hardware block-diagrams and representing the Esterel design structure.

2.4 Transactional Level Modeling

Transaction Level Modeling (TLM) is a high level approach to modeling digital systems where details of communication among modules are separated from the details of the implementation of the functional units or of the communication architecture [Gro02]. Low simulation speeds and complex SoC designs turn the system designers towards the more abstract level system representations. Transaction Level Modeling (TLM) is one such abstract system representation which can be regarded as the first representation of the implemented components after hardware/software partitioning. Because of its high level of abstraction, TLM gives us high simulation speeds for a complete system on chip design representation as compared to respective RTL models. Figure 2.3 shows the TLM abstraction layer. The transactional level itself is not a particularly defined modeling level but is a collective logical name for four model layers communicating processes (CP), communicating processes with time (CPT), bus cycle accurate (BCA), and cycle accurate (CA).

The *communicating processes* (CP) models are also called as untimed functional models (UTF). They are the architectural models targeted at early functional modeling and verification of systems where timing annotations are not required. They consist of asynchronous concurrent message-passing elements as there is no notion of time at this level just as for

system architectural models. As they do not contain unnecessary implementation details, they can be developed, optimized and rapidly modified to the designers' needs. These models are characterized by high simulation speeds and less complex design models. On the other hand, the *communicating processes with time* (CPT) model is a micro-architectural model containing essential time annotations for behavioral and communication specifications [Ghe05]. Compared to CP models, these CPT models are closer to RTL models. These models focus on the simulation accuracy, architecture analysis and real-time software development. The Open SystemC Initiative (OSCI) [3] has defined the above given modeling levels from the software engineering point of view. CP level is also called as the programmers' view (PV) and CPT level is called as programmers' view with time (PVT).

The *bus cycle accurate* (BCA) models consider the temporal granularity at the bus transaction levels. The message passing is usually atomic. At bus cycle accurate levels also known as cycle-approximate levels, the actual operations of the bus or IP accesses are present, but the timing between them is not known precisely. In *cycle accurate* models, the communication information is passed at clock boundaries and exact cycle counts are known. *Gate propagation* level is the most detailed hardware implementation model at which the timing within the clock period is also known precisely.

Electronic System Level being an established approach for System-on-a-chip (SoC) design can be accomplished through the use of SystemC as an abstract modeling language. *SystemC language* was developed by OSCI, a not-for-profit organization [SystemC], in 1999. It is a actually a collection library of C++ routines, macros and classes that can be compiled with any conventional C++ compiler. Its working and capabilities are similar to HDLs (VHDL and Verilog) like the simulation of concurrent processes, events, and signals but provides more flexibility, greater expressiveness, different templates, and data types, as well as the full power of the C++ language. It can be considered as a language for high level structure modeling and behavior representation using constructs like communication channels, buses, interfaces, modules and threads. Due to these features, SystemC is often duly associated with ESL design and TLM. It has got a wide variety of application like software development, system-level modeling, architectural exploration, functional verification, high-level synthesis, and performance modeling.

Modules are the basic building blocks of a SystemC design structure. A SystemC model usually consists of several modules that communicate through ports. Modules are represented using two types of C++ functions, methods and threads. Threads are the function calls that can be paused using the wait statement whereas methods run till the end of their execution once triggered or called. SystemC models execute in discrete events of time. Channels are the main communication of SystemC connecting module interfaces. They can be either simple wires or complex communication mechanisms like FIFOs. Modules, ports, interfaces, and Channels together form interface-based design style, where computation and communication can be modeled separately. SystemC also introduces a number of different data types.

2.5 Register Transfer Level

Register transfer level (RTL) refers to that level of abstraction at which a circuit is described as synchronous transfers between functional units, such as multipliers and arithmetic-logic units, and register files [Gro02]. The *basic building blocks* of an RTL design are modules constructed from simple gates. These building blocks are comprised of functional units, such as adders and comparators, storage components, such as registers, and data routing components, such as multiplexers. This abstraction level is also referred to as module-level abstraction [Chu06].

[3] http://www.systemc.org/home/

Contrary to the gate level representation, RT level *data representation* is more abstract. Ports and data types are used to represent groups of hardware connections and signals respectively. Similarly, the *behavioral description* of a system, to specify the functional operations at this level, generally uses finite state machines (FSMs) to express the data flow. The system state of the FSM describes the internal behavior state of the system at any specific instance of time. The *timing representation* at the RT level is shown by a clock signal input to the storage components like register blocks. These clock signals act as a trigger for the actions in the synchronous modules. Data is manipulated (latched, sampled, read, written *etc.*) on the edge (rising or falling) of these clock signals. In this way, these clocks act as the synchronizing pulse for the system. The clock pulses have frequency limitations by the hardware implementation. Clock signals have to be of long enough durations that they account for the system propagation delays. Thus due to the use of clocks, we do not usually consider the system changes occurring within the clock cycle and the system timing is considered in terms of number of clock cycles.

A digital system design can be described at different levels of abstraction and with variety of view points, depending on the background and knowledge of the designer and requirements of the system. Here comes the role of the hardware description languages (HDLs) which let the designers to accurately model the designs from the structural and behavioral view points, at a desired level of abstraction. This thing gives the system designers an opportunity to focus their efforts on the design and do not mix up the different view points of a design also. This effort also gives a standardized approach to the system design and make various designs compatible with each other. Traditional programming languages are not adequate for modeling digital hardware which also increases the necessity of the role of hardware description languages. The traditional general purpose languages like C follow the sequential process paradigm which helps the programmers to easily formulate a step by step design approach. On the other hand the digital hardware consists of concurrently operating small modules of sequential or combinational circuits. Such sort of design modeling is not possible with general purpose languages.

One of the popular hardware description language that we focus on in our work is VHDL. VHDL stands for VHSIC (Very High Speed Integrated Circuit) HDL. It is used to describe the structure and behavior of a system. It can describe a system at various levels like behavior, data flow, and structure. VHDL system design and programming is usually a top-down design methodology where a system design is modeled at a higher level. Later this model is tested using simulation techniques and then refined to low level synthesizable hardware implementations. A typical VHDL module consists of two major parts: an entity declaration and an architecture description. The *entity declaration* defines the interface of the module. This interface is visible and available to the other components. The *architecture body* specifies the internal operation or organization of the circuit. VHDL supports "hardware" types (`bit`, `bit_vector`,...), and multi-valued logic types (`sd_logic`) through its IEEE library. The statements inside the architecture body are executed as concurrent building blocks (in parallel). An *after* keyword is used to trigger the statement blocks at a particular instance of time relative to the input clock. These VHDL modules (also known as components) once declared and defined, can be instantiated in the architecture body of other modules. Hence, we can have a library of defined components which can be used in our designs as needed. Common VHDL editors/simulators come with a set of such libraries having commonly used digital design components. When instantiating a component in our design, the *port map* keyword is used to bind the interfaces of the instantiated component to the ports of the component specification. Such a structural description of our design facilitates the hierarchical design of complex systems. VHDL also facilitates us to program sequential statements. Such a code is sometimes also referred to as *behavioral description*. These features are considered as an exception to the VHDL semantics [Chu06]. These statements are encapsulated in a special construct known as

a *process*. A process can have a sensitivity list, a set of signals, which trigger its execution.

One of the major advantages of programming at the RTL level in languages like VHDL is of program simulation. This helps us to study the functioning of our program (and eventually the hardware) or its functional verification. Testing and verification are studied in chapter 8.

2.6 Conclusion

In this chapter we have described the general concepts of ESL design especially in context of different levels of abstraction. We have initially presented the traditional view of design-flow approach used, followed by the focus on new trends like co-simulation and hardware/software partitioning. In the end, we have introduced programming languages for modeling at different abstraction levels. These languages are later utilized while dealing with the behavioral representation of electronic systems (chapter 7).

Chapter 3

Model Driven Engineering

Contents

*Model-driven engineering (*MDE*) is a software development methodology which focuses on models, meta-models, and model transformations. The Unified Modeling Language (*UML*) has greatly contributed to the popularity of the* MDE *in software engineering. Specialized forms of* UML *extend modeling capability of* UML *far beyond the scope of software engineering, for instance to Systems Engineering with* SysML*. This thesis is in line with this approach and proposes a* UML*-based modeling of electronic systems.*

This chapter is a brief introduction to UML*, focusing on a subset of this modeling language on which relies the* UML *profile for* IP-Xact*, our contribution developed in chapter 6. Parts of the* UML *profile for real-time and embedded systems (*MARTE*), reused in our profile, are also presented. Model to model transformation, seen as an* MDE *actvity, is briefly explained in a last section.*

3.1 Introduction

Model-driven engineering (MDE) is a software development methodology focusing on creating specialized engineering models or abstractions. A model is the simplified representation of a system that highlights the properties of interest for a given purpose or point of view. Thus, a model effectively is an *abstraction* of the real world problem/system removing or hiding all the irrelevant details giving a more understandable view. A good model further enforces that it correctly represents the properties of interest of the system. It should also be capable of accurately predicting the behavior of that modeled system. Model-driven engineering increases productivity by simplifying the process of design, maximizing compatibility between systems, and enhancing communication between individuals and teams working on the system. A modeling paradigm for MDE is considered *effective* if its models are useful from the point of view of the user and can serve as a basis for implementing systems.

Model-driven development (MDD), another term frequently used in this domain, is an approach to software development in which models become essential artifacts of the system development process rather than merely serving as inessential supporting blocks [Sel06]. As the models approach completion, they enable the development of software and systems through the use of model transformation techniques. An increase of automation in program development is reached by using computer aided automated transformations in which higher-level models are transformed into lower-level models until the model can be made executable using either code generation or model interpretation.

In this chapter, the basic building blocks of model-driven techniques, *i.e.,*the models and metamodels are described in the next section. In Section 3.3, we dig into the basics of the metamodel of UML, a general purpose modeling language. In section 3.4, we describe parts of the MARTE profile for UML we reuse in our approach. Section 3.5 addresses *model to model transformation*, another pillar of MDE, which will allow transformation of UML models into IP-Xact specifications.

3.2 Models and metamodels

MDE aims to raise the level of abstraction in program specification and increase automation in program development. The idea promoted by MDE is to use models at different levels of abstraction for developing systems. Similarly, in Electronic System Design, different languages are used for various representations of the same components at different view points like SystemC for TLM modeling and Verilog/VHDL for RTL modeling. These models differ in the amount of information they contain. For instance, TLM-PV lacks timing information whereas the TLM-PVT also includes the timing and behavior information of the same design under consideration.

A model is specified in some notation or language. Since model languages are mostly tailored to a certain domain, such a language is often called a *Domain-Specific Language* (DSL). A DSL can be visual or textual. A sound language description contains its abstract syntax, one or more concrete syntax descriptions, mappings between abstract and concrete syntax, and a description of the semantics. The abstract syntax of a language is often defined using a *metamodel*. A metamodel is a precise definition of the constructs and rules needed for creating well-formed, *i.e.,*legal models.

Model hierarchy

The MDE defines a relationship, called "isRepresentedBy", which relates a *system*, or simply anything of interest and its *model*. The "conformsTo" relationship is another one which binds a model to its *metamodel*. A metamodel is actually a model representing a collection of models. A model from this collection is said to *conform to* the metamodel. Since in the MDE, a metamodel itself is also a model and it has to conform to its metamodel (or meta metamodel of the system). The relationship "conformsTo" could therefore apply an indefinite number of times as each metamodel being itself a model has to comply to a superior meta-model. Fortunately, in MDE, the meta metamodel is self-describing, so that 3 levels of abstraction suffice. In the OMG approach to MDE—called *Model Driven Architecture* or MDA®—the meta metamodel is the MOF (Meta-Object Facility), an OMG standard that enables metadata management and modeling language definition. UML is one of the modeling language whose metamodel is specified in MOF. The matter is a bit more complex, because with UML 2, the MOF and the UML kernel are now closely dependent.

3.3 UML

Unified Modeling Language (UML) is an OMG standardized general-purpose modeling language in the field of software engineering. UML includes a set of graphical notations to create models of specific systems. A UML model consists of number of different model elements put in different diagrams targeted to a specific use. These diagrams include class diagrams, component diagrams, composite structure diagrams for structural aspects; activity diagrams, state machine diagrams, and sequence diagrams for the behavioral aspects. A UML model must conform to the UML metamodel specified in a huge OMG document entitled "UML Superstructure" (742 pages for the last version [OMG09b]). In this section, we describe some selected UML metaclasses and propose simplified excerpts of the UML metamodel. As said before, the abstract syntax of UML should be specified with a MOF-based metamodel. However, the *Infrastructure-Library* defined in the "Unified Modeling Language: Infrastructure" [OMG09a] can be strictly reused by MOF 2.0 specifications, so that the UML metamodel looks like a UML model. This maybe a source of confusion for the reader. The level of modeling (UML users's model or UML metamodel) will be explicitly given in case of ambiguity.

3.3.1 UML Classifiers

UML model elements are grouped in hierarchical *packages*.

The UML::Classes::Kernel package contains the core modeling concepts of the UML. UML has the general concept of *Classifier*. A classifier describes a set of objects; an object is an individual thing with a state and relationships to other objets. Thus, a classifier describes a set of instances that have features in common. A classifier is an *abstract* metaclass[1]: it is not directly instantiable, only its *concrete* specialized subclasses are. The class diagram in figure 3.1 represents some metaclasses of the Kernel package and their relationships. This diagram shows that a Classifier *is a* Namespace, *i.e.*,it contains a set of named elements that can be identified by name. It also appears that Class is a subclass of Classifier. A class owns 0 or many *properties*, and 0 or many *operations*. A Property is a *structural feature* (related to the structure of instances of the classifier). An Operation is a *behavioral feature* (related to an aspect of the behavior of the instances of the classifier). Class is the most used UML metaclass.

[1] On graphical representations, the name of an abstract metaclass is written in italics.

Its instances are called *objects*. Note that a class may own nested classifiers, a property we will exploit in our profile for IP-Xact (chapter 6).

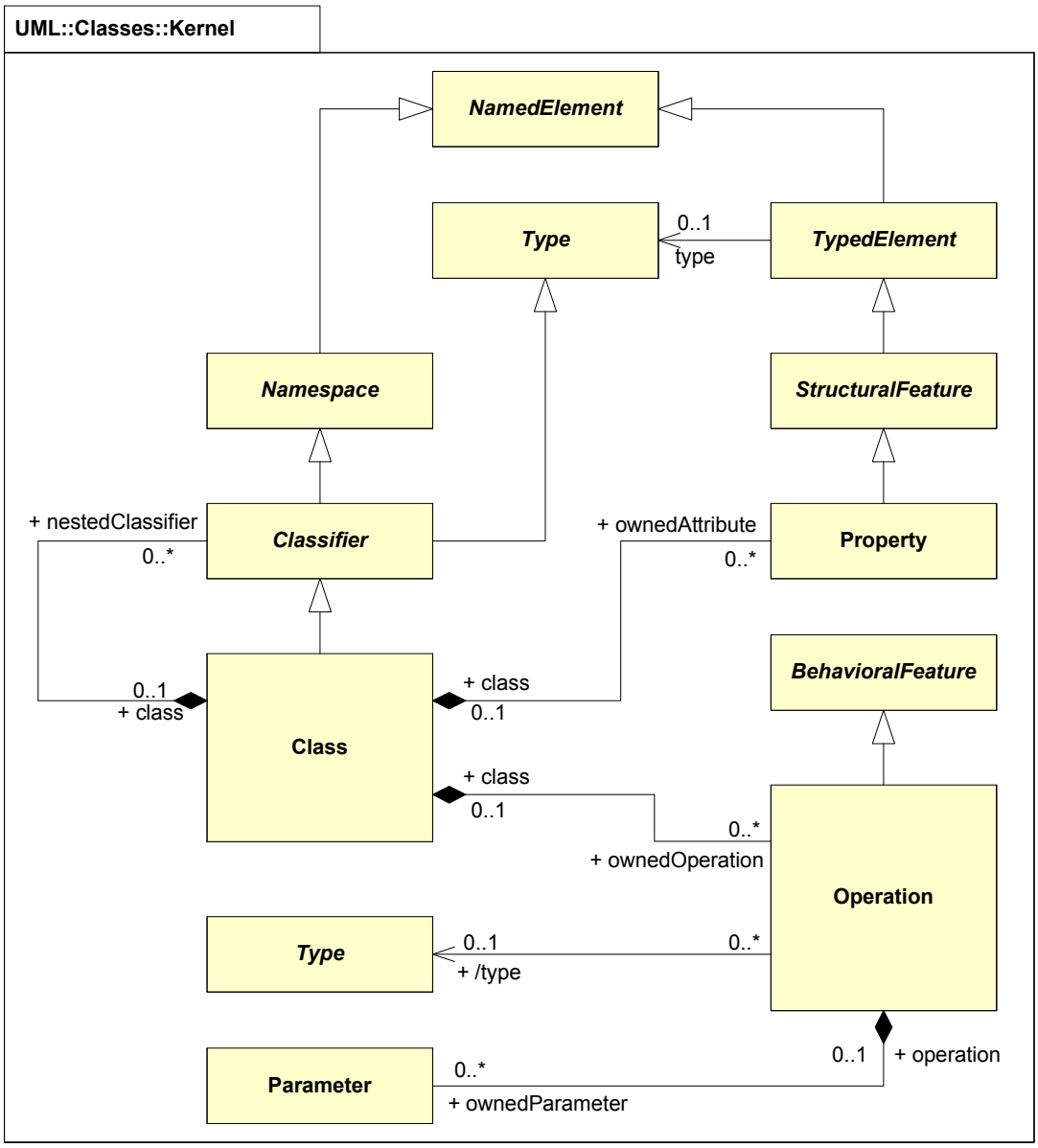


Figure 3.1: Simplified class diagram of the Kernel metaclasses.

The Kernel package also contains the Constraint metaclass, not shown in figure 3.1. A constraint is a condition or a restriction that precises the semantics of an element. Constraints can be expressed in OCL [OMG03], which is a formal language used to describe expression on UML models and metamodels. These expressions specify invariant conditions that must hold for a valid model. At the metamodel level, OCL constraints are useful to fix the semantics of

some elements, especially when defining profiles.

Classes, such as defined in the Kernel package, along with *associations*, are the main model elements encountered in a user's class diagram. These diagrams have the same form as the one in figure 3.1, but they are not at the same modeling level. In a user's model, an association specifies a semantic relationships between typed instances. This association is general, not specific to particular usages or contexts. Class diagrams were the main *static structural* representations in UML 1.x. Early attempts at modeling complex Real-Time systems [Sel98] and more generally using UML as a standard architecture description language (ADL) [RSRS99] showed that UML 1.x class diagrams even combined with collaboration diagrams were not sufficient to deal with architectural aspects. A major improvement of UML 2 has been the introduction of *composite structure diagrams* and *structured classifiers* described in the next subsection.

3.3.2 UML Structured Classes

An *architecture* defines the high-level structure and behavior of a *system* during execution [WGK+00]. Quoting this paper, "Architecture is concerned with the *decomposition* of a system into parts, the *connections* between these parts, and the *interaction* of the parts of the system required for the functioning of the system". This definition applies as well to electronic systems. When modeling an architecture, we want to know "who speaks to whom" in the particular context of the system at hand. This information is not provided by a class diagram that does not reflect the effective decomposition of the system. In fact, the architecture of a system is composed of instances—not classes—of various kinds and their interconnections. An object diagram, familiar to Object-Oriented programmers, is not a solution either because objects models show completely reified objects, *i.e.*,examples, and thus are not reusable. To address this modeling issue, UML 2 has introduced a new diagram: the *Composite Structure Diagram* and related new concepts (Structured Classifier, Ports,...). These model elements are shown in figure 3.2. Note that some concepts, like Property or Class, already defined in the kernel package, reappear in the newly introduced packages. This is usual in the UML specification and is known as a *merge increment*: the increment is additional characteristics given to an element (*e.g.*,the capability to have an internal structure and ports for Class), the merge applies to packages and combines their contents. This is a complex operation we will not detail. For the user it is sufficient to be aware that a UML model element can be enriched several times in the specification, and this results in a model element with all the additional features.

Connectable Elements and Structured Classifiers

UML 2 has introduced the concept of *connectable element*. A ConnectableElement is an abstract metaclass representing a set of instances that play roles of a classifier. Property is a concrete subtype of ConnectableElement (another example of merge increment). A StructuredClassifier is an abstract metaclass that represents any classifier whose behavior can be described by the collaboration of owned or referenced instances. Properties—as connectable elements—owned by a structured classifier are called *parts*. The choice of this word is judicious. It can be understood in two ways: as part of a whole, *i.e.*,a piece, an element of a composite structure, and as a role performed by an actor. Thus, structured classifiers fit to architecture modeling, and connectable elements can conveniently represent the components.

A *connector* is a link that enables communication between two or more instances. A Connector is not an Association. It makes it clear "who speaks to whom".

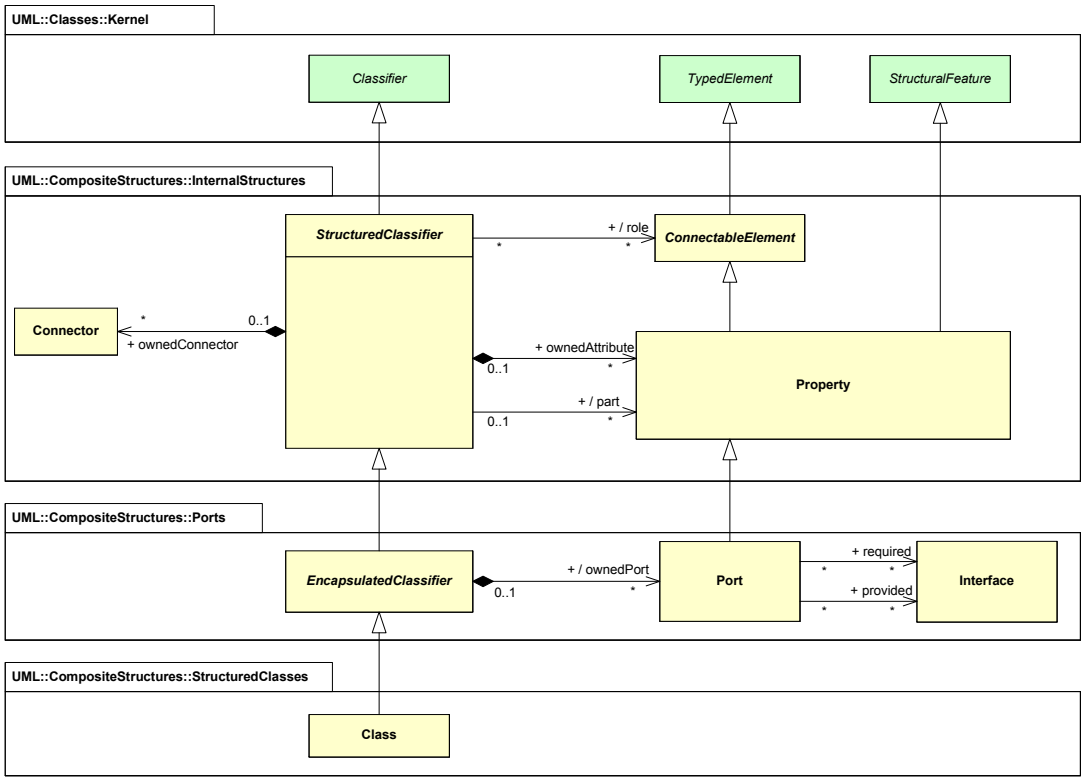


Figure 3.2: Simplified class diagram of the CompositeStructures metaclasses.

Ports and Encapsulated Classifiers

An EncapsulatedClassifier is a classifier that owns ports, through which it may interact with its environment. A Port is a property of its owning classifier that specifies a distinct interaction point between that classifier and its environment or between the (behavior of the) classifier and its internal parts. Ports are connected to properties of the classifier by connectors through which requests can be made to invoke the behavioral features of a classifier. A Port may specify the services a classifier provides (offers) to its environment as well as the services that a classifier expects (requires) of its environment. Ports can either delegate received requests to internal parts, or they can deliver these directly to the behavior of the structured classifier which owns this port.

Class with internal structure

The subpackage UML::CompositeStructures::StructuredClasses contains only one metaclass (Class) which extends the kernel metaclass Class with the capability to have internal structure and ports.

Usually, classes with internal structure are represented in *Composite Structure Diagrams*, this is a new kind of static structure diagram not present in UML 1. This diagram represents the internal structure of a class and the collaborations that this structure makes possible. It is the composition of interconnected elements, representing run-time instances collaborating

over communications links to achieve some common objectives. Figure 3.6 on page 29 is an instance of such a diagram.

3.3.3 UML Profiles

UML is a general purpose modeling language. It can represent concepts from many specific domains and adapt its semantics accordingly. *Lightweight extensions* of UML modify semantics of UML concepts by specialization. This specialization must not contradict the semantics of UML, taken as the reference model. On the contrary, *heavyweight extensions* introduce non-conformant concepts or incompatible change to existing UML semantics/concepts. Heavyweight extensions are defined at the MOF level, whereas lightweight extensions modify (specialize) the UML metamodel through the UML profile mechanism, a specific metamodeling technique. In this thesis we consider leightweight extensions only.

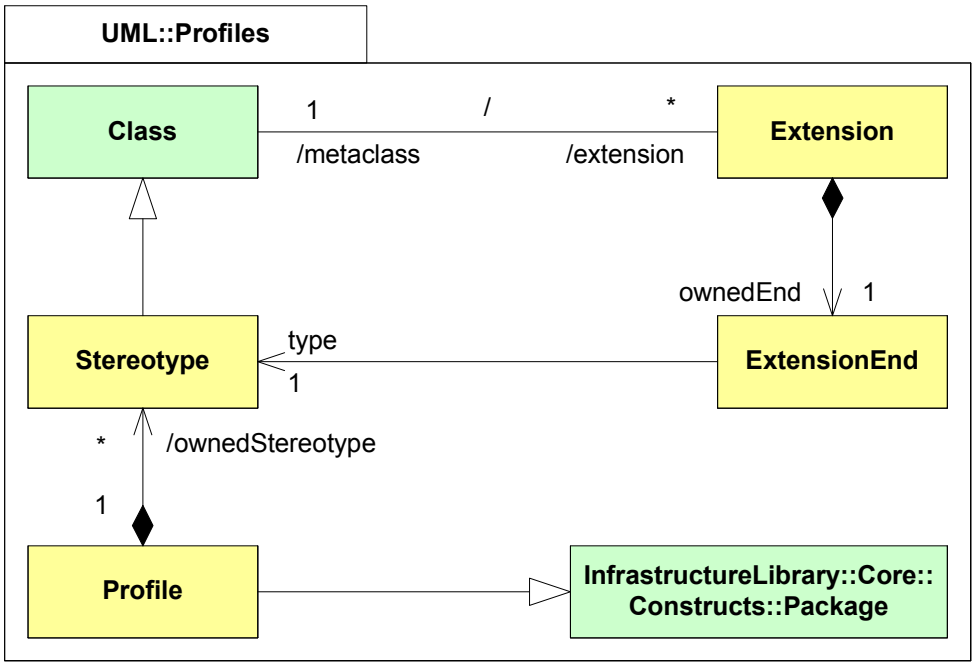


Figure 3.3: Simplified class diagram of the Profiles metaclasses.

The UML::Profiles package contains the UML metaclasses that allow lightweight extensions. Figure 3.3 shows some of these model elements. A Profile is a specialization of Package. This package contains imported elements, imported packages and *stereotypes*. A Stereotype gives specific roles and semantics to existing metaclasses and records additional context-specific information in meta-attributes also known as *tagged values*. Having a closer look at figure 3.3, we see that an Extension, which is a kind of Association, relates a Stereotype to the metaclass it extends. Here, there is an issue with this metamodel, since a metaclass could only be a Class or one of its numerous specializations. Thus, metaclasses like Property, Operation, etc. could not be stereotyped. This is in total contradiction with what is supported by UML (meta)modeling tools. This metamodel inconsistency has been reported in a paper entitled

"Uses and Abuses of the Stereotype Mechanism in UML *1.x and* UML *2"* [HSGP06]. This problem has also been identified within the OMG and is being worked on. It seems that there is a confusion between two concepts of Class. The Extension::metaclass association end from Extension to Class should terminate on MOF::Class, whereas the Stereotype metaclass should extend the (UML) Class. There is no such example of mixing MOF and UML levels in the UML superstructure specification, that may be the reason for the delayed fixing of this metamodel error. In what follows, we assume that any UML metaclass, but the Stereotype metaclass, can be stereotyped. In fact, a stereotype may only generalize or specialize another stereotype.

3.3.4 Example of UML modeling of a Timer

A simple timer counts occurrences of an event and signals when a given number of occurrences has been received. Often the event is bound to time, for instance clock pulses, hence the given name of timer. A Timer is a device made of one or several simple timers.

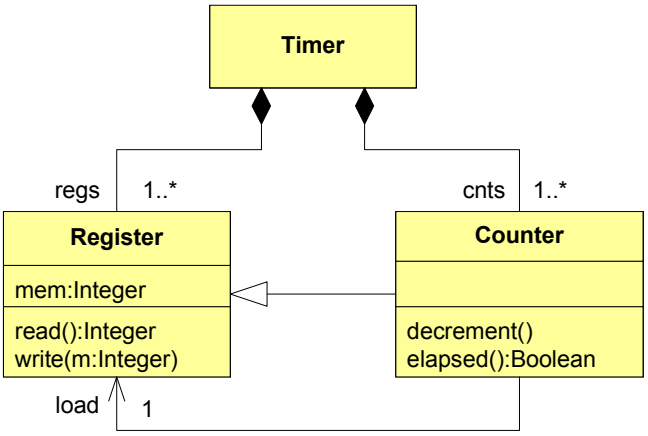


Figure 3.4: A class diagram for a Timer.

A simple timer consists of a *register* and a *counter*. The register contains the number of occurrences to be counted, the counter effectively (down) counts the occurrences and checks if the count is 0. A class diagram is proposed in figure 3.4. The class Register owns an integer attribute mem, and has two access operations: read and write. The class Counter is a specialization of Register with an additional property (load), and two additional operations: decrement and elapsed. Once Counter.mem is 0, its value is updated to the value contained in the associated register (Counter.load.mem).

Figure 3.5 introduces classes with ports to represent registers and counters. The port named access of the Register class offers read and write functionalities. The ports of the Counter class offers additional functionalities (decrementation and end-of-count signaling). Note that, due to the generalization relationship between Register and Counter, the latter has also an access port, as shown in the right-hand side of the figure.

The class diagram in figure 3.4 does not show "who speak to whom". This information is clearly visible on a composite structure diagram (figure 3.6). The class Timer2 consists of two independent timers and a prescaler that divides the frequency of the incoming clock (clk port). Indeed, the prescaler is a simple timer whose elapsed port feeds the two other simple timers. The command port is a *behavioral port* through which behavioral features of the classifier can

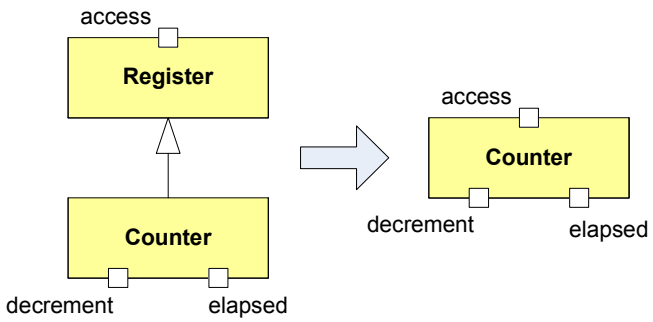


Figure 3.5: Class with ports.

be invoked, for instance to load a new count value in a timer or the prescaler. Ports to1 and to2 stand for timeout ports. They are connected to the corresponding elapsed ports.

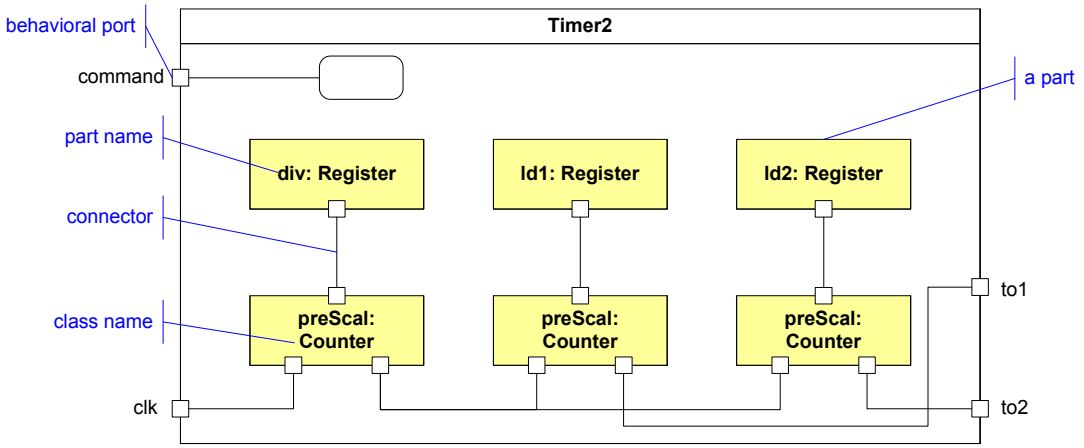


Figure 3.6: Timer as a Structured Class.

3.4 The UML Profile for MARTE

3.4.1 Overview

The new OMG UML profile for MARTE supersedes and extends the former UML profile for Schedulability, Performance and Time (SPT [OMG05]). MARTE also addresses new requirements: specification of both software and hardware model aspects; separated abstract models of applications and execution platforms; modeling of allocation of the former onto the latter; modeling of various notions of Time and Non-Functional properties.

Figure 3.7 represents an overview of the MARTE domain model. MARTE consists of three main packages. The first package defines the *foundational concepts* used in the real-time and

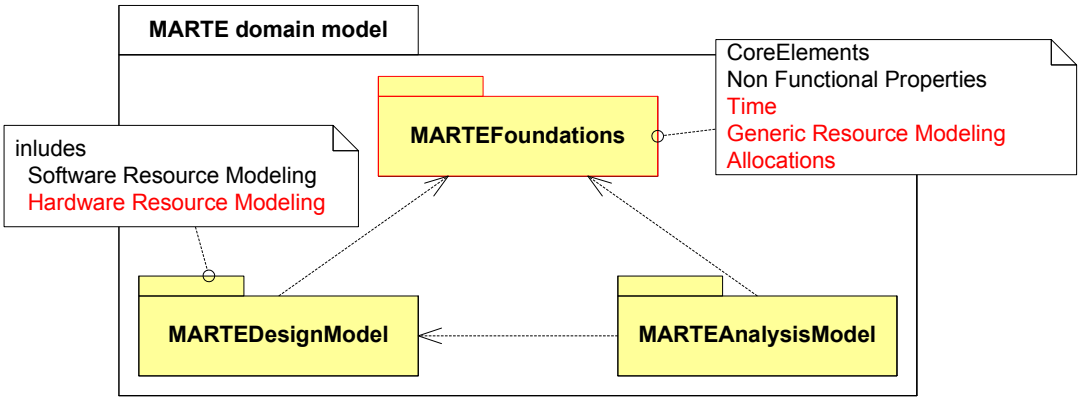


Figure 3.7: MARTE overview.

embedded domain. These foundational concepts are refined in the two other packages to respectively support modeling and analysis concerns of real-time embedded systems. The second package addresses *model-based design*. It provides high-level model constructs to depict real-time and embedded features of applications, but also detailed software and hardware execution platforms. The third package addresses *model-based analysis*. It provides a generic basis for quantitative analysis sub-domains. Our profile for IP-Xact reuses several model elements from the first and second packages. The following subsections briefly describe these borrowings, highlighted in figure 3.7 as red texts in note symbols. The MARTE Time profile receives a special attention for its advanced applications in chapter 7.

3.4.2 Resources and Allocation

The central concept of *resource* is introduced in Generic Resource Modeling (GRM) package of MARTE. A *resource* represents a physically or logically persistent entity that offers one or more *services*. A Resource is a classifier endowed with behavior (a BehavioredClassifier in UML terminology), while a ResourceService is a behavior. Resource and ResourceService are types of their respective *instance* models. See figure 3.8 for the domain view of the Generic resource Modeling. The GRM profile (one of the sub-profiles of MARTE) defined two stereotypes Resource and GrService to represent the concepts of resource and resource service respectively. These sterotypes extend several UML metaclasses. We just mention some: Classifier, Property, and InstanceSpecification for Resource, and BehavioralFeature for GrService.

Several kinds of resources are proposed in MARTE like ComputingResource, StorageResource, CommunicationResource, TimingResource. Two special kinds of communication resource are defined: CommunicationMedia and CommunicationEndPoint. The communication endpoint acts as a terminal for connecting to a communication medium; typical associated services are data sending and receiving.

For structural modeling, MARTE enriches the concepts defined in the UML composite structures. For instance, the UML metaclass Port has been extended into two stereotypes: FlowPort and ClientServerPort. Flow Ports model interaction points through which information flows. The direction of the flow is specified by the direction metaattribute. Possible direction values are in, out, or inout. Client-server ports have been introduced to make easier the use of UML ports. The metaattribute clientSeverKind makes it explicit whether a port provides,

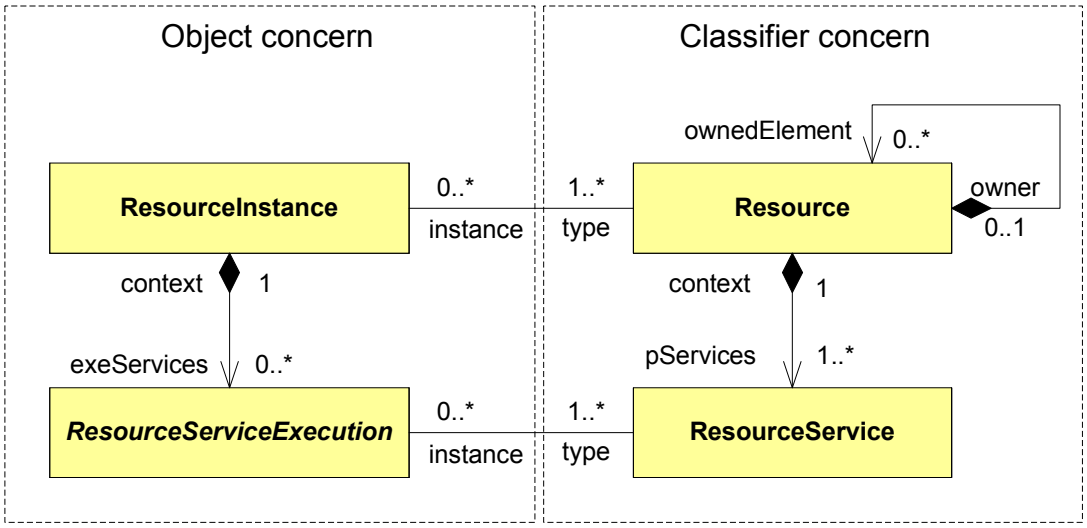


Figure 3.8: MARTE resource domain view.

requires, or both provides and requires behavioral features. In electronic systems these new ports can be useful to represent data exchanges (FlowPorts at RTL level) and transactions (ClientServerPorts at TM level).

The MARTE *Allocation* associates functional application elements with the available resources (the execution platform). This comprises both spatial distribution and temporal scheduling aspects, in order to map various algorithmic operations onto available computing and communication resources and services. It also differentiates *Allocation* from *Refinement*. The former deals with models of a different nature: application/algorithm on the one side, to be allocated to an execution platform on the other side. The latter allows navigation through different abstraction levels of a single model: System-level, RTL and TLM views.

The Detailed Resource Modeling (DRM) package of MARTE specializes these concepts. It consists of two sub-packages: Software Resource Modeling (SRM) and Hardware Resource Modeling (HRM). Only the latter is considered in this thesis.

As shown in figure 3.9, HwResource (HwResourceService resp.) specializes Resource (ResourceService resp.) defined in the GRM package. A hardware resource *provides* (hence the prefix 'p_' in the role name) at least one resource service and may *require* ('r_' prefix) some services from other resources. Note that a HwResource can be hierarchical. The HRM package is further decomposed into two sub-packages: HW_Logical and HW_Physical. The former provides a functional classification of hardware entities; the latter defines a set of active processing resources used in execution platform modeling and close to several Spirit IP-Xact concepts. HwResource is specialized in the same way as the generic resource of the GRM package (lower part of figure 3.9).

3.4.3 Time in MARTE

Both Resource and Allocation refer to the time model defined in the Time package of MARTE. While SPT considered only time models based on *physical time*, MARTE introduces two distinct models called *chronometric* and *logical* time. The former supersedes the SPT model and its

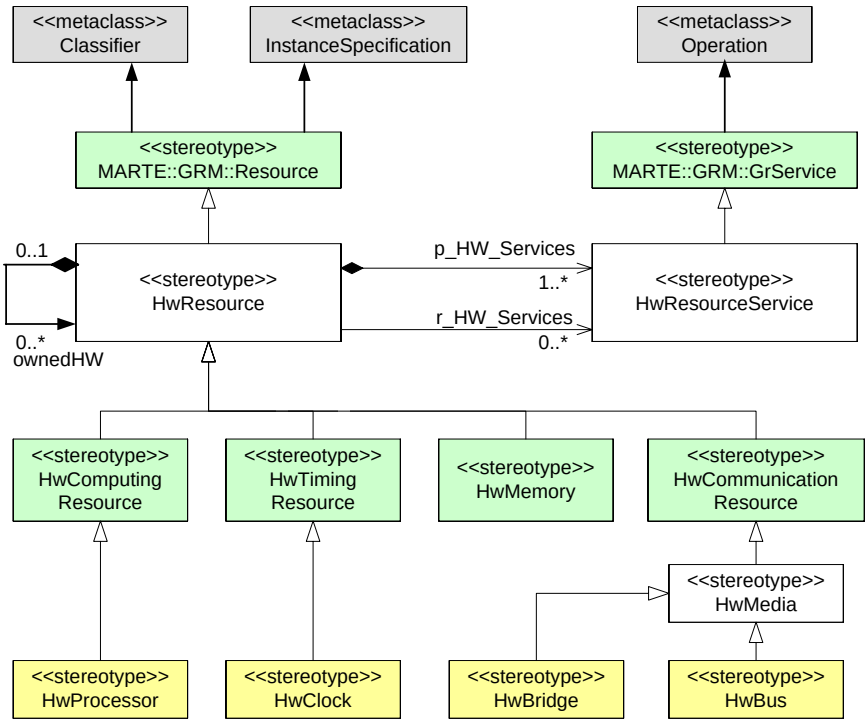


Figure 3.9: Excerpt from MARTE hardware resource profile.

time values are expressed in classical time units (second or one of its sub-multiples). The latter may "count" time in ticks, cycles, busCycles, or any other units. In fact, any event can define a logical clock that ticks at every occurrence of the event. Thus, logical time focuses on the ordering of instants, not on the physical duration between instants. Another noteworthy feature of the MARTE time model is the support of multiple time bases, required to address distributed embedded systems and modern electronic sytem designs.

In MARTE, the underlying model of time is a set of time bases. A time base is an ordered set of instants. Instants from different time bases can be bound by relationships (coincidence or precedence), so that time bases are not independent and instants are partially ordered. This partial ordering of instants characterizes the *time structure* of the application. This model of time is sufficient to check the logical correctness of the application. Quantitative information can be added to this structure when quantitative analyses become necessary.

A Clock is the model element that gives access to the instants of a time base; a ClockConstraint—a stereotype of UML Constraints—imposes dependency between instants of different time bases. Complex time structures and temporal properties can be specified by a combined usage of clocks and clock constraints. Examples are given in chapter 7.

MARTE also introduces the concept of *timed model element*. A TimedElement associates at least one clock with a model element. This association enriches the semantics of the element with temporal aspects. Thus, a TimedValueSpecification necessarily refers to clocks. A TimedEvent is an event whose occurrences are explicitly bound to a clock. A TimedProcessing represents an activity that has known start and finish times, or a known duration, and whose instants and durations are explicitly bound to clocks. The stereotype TimedProcessing may be applied to UML Action, Behavior, and even Message.

The MARTE Time and Allocation models are introduced in the MARTE specification, but for a better understanding of the underlying concepts and their practical use, the reader is urged to refer to academic papers. The paper [AMdS07] entitled *"Modeling Time(s)"* is a general introduction to the MARTE time model and the associated semantics. The clock constraints, which are not detailed in the UML specification, can be specified with CCSL, the *Clock Constraint Specification Language*, a formal language whose syntax and semantics are described in papers [MA09, AM09a] and a research report [And09b]. Publications about applications of MARTE are many, ranging from first assessments of the profile for control system specification [DTA+08] to more specific uses of the Time profile in different domains: automotive [AMPF07, MPA09], and avionics standards [AMdS08, MdS09, MAD09]. No surprisingly, most of the references of Time in MARTE are from the project-team AOSTE, because the Time and Allocation profiles have been proposed, written and maintained by members of this project. For more general references on MARTE, the reader may consult the dedicated website (http://www.omgmarte.org/).

3.5 Model to model transformation

Automation of the design process is one of the objectives of the MDE aimed at applying the MDE techniques to the practical world. One of the key steps of this design automation is the model to model transformation, which converts one design representation/abstraction into another form. A set of nice definitions related to model transformation are given in the book 'MDA Explained' by Kleppe et al. [KWB03]. It defines a *model to model transformation* as the automatic generation of a target model from a source model, according to a transformation definition. A *transformation definition* is a set of transformation rules that together describe how a model in the source language can be transformed into a model in the target language. A *transformation rule* is a description of how one or more constructs in the source language can be transformed into one or more constructs in the target language. The underlying theme of model transformation is to form a correspondence relationship between the input and output models of the system, as shown in the figure 3.10. The transformation rules are themselves a model and hence conform to a metamodel, which is usually provided by the transformation engine. These models can be expressed in a variety of ways like we can use UML to design graphical models or some programming languages to express the program source code. All these models must conform to their metamodels (shown in figure 3.10) which defines their syntax and semantics.

The model to model transformations can be categorized as endogenous or exogenous transformations based on the kind of conversion they perform [MG06]. The *endogenous transformations* (also referred to as rephrasing) are the one in which the input and the output models use the same metamodel for their syntax and semantics definitions. This means that the two models are expressed in the same language. Examples of such transformations are *optimization* (like code optimization to enhance the performance), *refactoring* (changing the internal structure of a design), and *normalization* (where complex syntactic constructs of a model are broken down into smaller simple modules). On the other side, the *exogenous transformations* are between models expressed using different languages, thus having different metamodels at the input and output. Typical examples of such transformation are *synthesis* (conversion of high level abstract model into low level concrete model, like in code generation or model compilation), *reverse engineering* (extracting abstract model specification from an implementation), and *migration* (transforming the code in one language like Java into another one like C++ without changing the level of detail of the model).

In the figure 3.10, the transformation engine takes only one input model and produces a

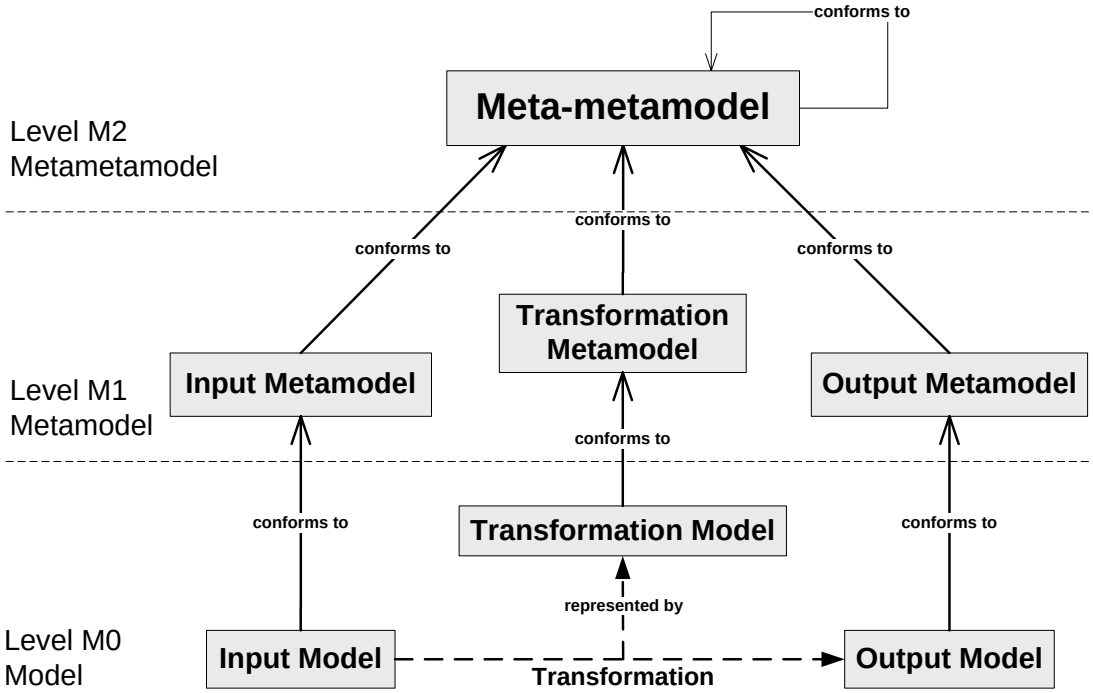


Figure 3.10: Model to model transformation.

single output model, but in other cases, it is possible to have several input and/or output models conforming to different meta-models. Moreover, we can have a single meta-model used for the input and output models, in which case we call such a transformation as *model refinement*. There are some indigenously coined terms in the model transformation domain referring to different concepts. As an example, Tom Mens et al. use the above given term 'refinement' for the vertical transformation mentioning that a model is refined to a low level implementation, whereas research leading to tool like ATL (Atlas Transformation Language) [Fou06] uses the terms *refinement* or *refining* for horizontal or endogenous transformations, giving an output model resulting from the 'fine tuning' of input model. In our work, we shall stick to the terminology used by the popular model transformation tools (like ATL) as they concern us the most. In such tools, model refinement is usually the single metamodel transformation and model elements are relocated in it.

There are several model to model transformation tools in the MDE world notably the ATL (Atlas Transformation Language)[JABK08, JAB+06] developed at INRIA. It is one of the most powerful tool implemented on the QVT (Query View Transform) standard defined by the OMG [OMG08a]. It is also used in our model transformation experiments (described in subsequent chapters). In our model transformation project (converting UML models into IP-Xact models), the ATL language allows us to program model transformation algorithms (*UML2IPXACT.atl*) implementing the transformation rules which must conform to the transformation metamodel provided by ATL. Both the input model provided to the transformation and the output model produced by the transformation conform to their respective metamodels. Figure 3.11 shows the transformation mechanism along with the example transformation. Here the role of transformation engine is to load the input UML model *Leon2TLM.ecore* in compliance to the input UML meta-model *UML2.ecore*, later it interprets the rules to create

the instances of elements of the output model mapped to the input model elements. The output model *IPXACTOut.xml* is generated in a way that it conforms to the output meta-model *IPXACT.ecore*. The input and output meta-models must conform to the OMG's MOF (MetaObject Facility) specification.

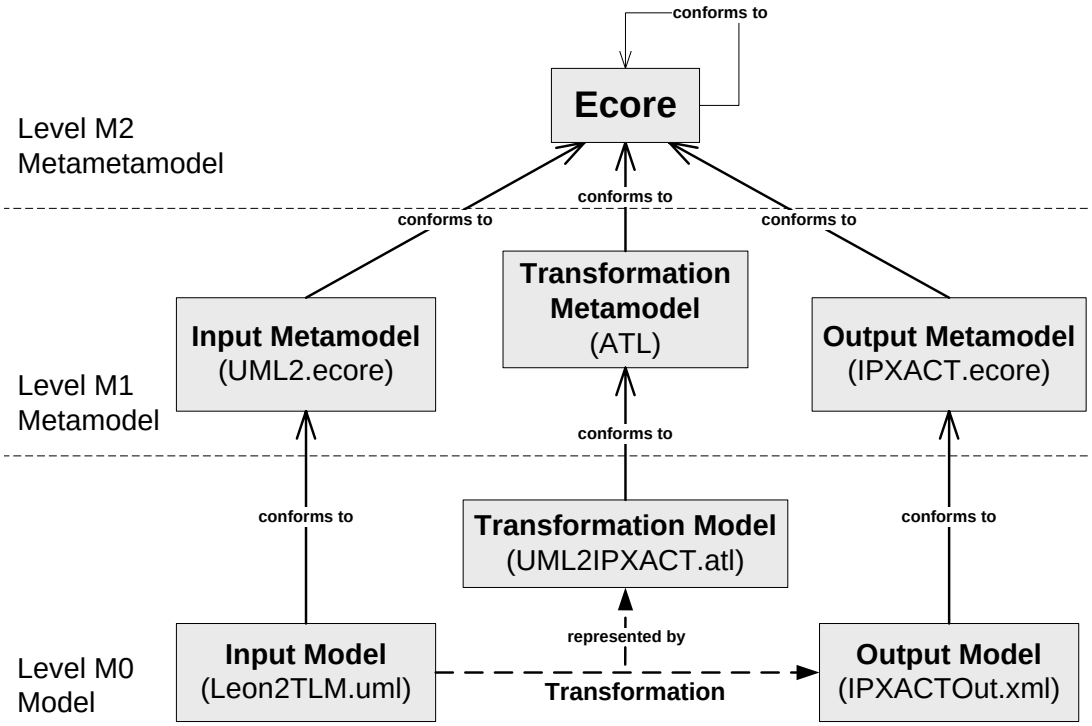


Figure 3.11: Model to model transformation using ATL.

Another important feature to consider in model transformation is the declarative or imperative approach used [MG06]. This concept is similar to the one discussed before about the synchronous languages. The *declarative approach* focuses on the model elements themselves (focus on data flow) or on the *what* aspect, *i.e.,*what needs to be transformed into what? Such an approach is best suited from the theoretical point of view, giving us simpler formally founded models. On the other hand, the *imperative/operational approach* focuses on process of transformation (focus on control flow) or on the *how* aspect, *i.e.,*how input model elements will be transformed into the output model elements. These transformations are better suited to the transformation leading to an incremental updating of source model. The ATL transformation tool provides both types of transformation approaches but mostly the designers recommend declarative approach in the model transformations [2].

In our present work, model transformation is used to transform the UML based models into the IP-Xact XML models (discussed later in chapter 5). Hence, we get the power of representing the textual IP-Xact model information into UML graphical models. Other than this we can have numerous other applications for the model transformations. We can define models at one abstraction level like TLM and later write transformation rules to transform those models into another representation like RTL, hence providing the interoperability of models. Moreover, the model refinement techniques (using similar input/output meta-models) allow us to analyze

[2] http://wiki.eclipse.org/ATL_FAQ

the input models and produce processed output models for a particular point of view. Model transformations allow us to create output model skeletons deduced from the input models. This can be helpful to reduce the development time of a design structure. As an example, we consider the work done by Revol [Rev08] in which he designs a model in UML representing the behavior of a system. Then the model transformation is used to generate the skeleton of code modules in SystemC language, to better simulate the behavior at later stages. Such a code skeleton can then be tailored to give the optimum desired results.

3.6 Conclusion

In this chapter we introduced the basic concepts of MDE followed by one of its greatest representatives, the UML. We have also highlighted the concept of profiles as an extension mechanism to this language, allowing us to cover the key aspects of our work. In the next chapters, we will apply these MDE concepts, including UML and its MARTE profile, to IP-Xact. Model transformation then will help to transform UML-based design models into IP-Xact specification, which is one of the objectives of this thesis.

Chapter 4

Illustrative Example

Contents

In this chapter, firstly we propose a definition of protocol associated with a communication. Then we establish an example of a simplified acquisition system which can cover all the aspects of our research work discussed later. Finally, we model the structural aspects of this application with different languages: Esterel, VHDL*, and SystemC.*

4.1 Communications and interactions

4.1.1 Protocol

Modeling a system as a MoCC brings a clear separation of computations and communications. The system can be represented by its *architecture* that consists of computation units and interconnections between these units. Architecture description languages (ADLs) have been proposed as modeling notations to support architecture-based development [MT00].

Communications or interactions between computation units often follow some well-established set of rules, known as a *protocol*. This term was first used, with this meaning, in telecommunication systems. The protocol is there a set of rules governing the format and chronology of message exchange in a communications system. We aim at a more general concept. B.P. Douglass [Dou98] proposed a wider definition he applied to multiprocessor systems. *"The overall structure of the protocol represents a set of important architectural decisions. A protocol is defined to be the rules, formats, and procedures agreed upon by objects wanting to communicate. By this definition, a protocol includes not only the physical characteristics of the medium but also the additional rules of behavior governing the use of the medium"*. Thus, a protocol imposes the format of the exchanges, the *role* of each communicating partner and the sequencing (behavioral aspect). In this thesis, we focus on the last two points (roles and behavior) in our models. We use protocols as a high-level description of communications, covering both structural and behavioral aspects. Similar approaches have been proposed in the past for real-time embedded systems. Besides the already mentioned Douglass's book [Dou98] on Real-Time UML, ROOM [Sel96] can be considered as a pioneer in this domain. It was a real-time object-oriented modeling proposed by Bran Selic in the 90's. In ROOM, the application is described as a collection of concurrent objects called *actors*. An actor communicates with other objects through one or more interface objects called *ports* bound to a *protocol*. Rumpe et al. [RSRS99] analyzed the limitations of such an approach relying on objects and UML1. They suggested extensions to make UML an effective ADL (Architecture Description Language). Selic has revised [Sel05] his approach by integrating new modeling features of UML2. Our proposition clearly comes within this model-driven approach, relying on UML2 structured classes instead of limited UML1 classes.

4.1.2 Protocol metamodel

In our structural modeling, we attach a protocol to a set of ports. Figure 4.1 is the metamodel we propose for protocols and related concepts. This metamodel does not pretend to cover all the features of protocols. The influence of our application domain (electronic systems) is manifest.

A Protocol coordinates two or more protocol ports. It adopts a style, for instance the masterSlave control pattern style. The *master* is the participant that initiates the communication/interaction. The property isDirect set to true says that a master can be directly connected to a slave. Otherwise, there must be at least one intermediate component between the master and the slave. A procotol refers to a set of *messages*. This is a derived property deduced from the protocol port types. A message represents an exchange of information. It can either be a (UML) Signal or an Operation. Both are BehavioralFeatures in the UML.

A classic UML port can be specialized as a ProtocolPort. This port has a portType and two special attributes (role and isMirrored). The latter deserves an explanation and is related to the portType of the port. The ProtocolPortType is a set of declarations consisting of pairs 'direction, message'. The direction has different interpretations according to the nature of the behavioral feature. The interpretations are given in table 4.1. The last column shows the

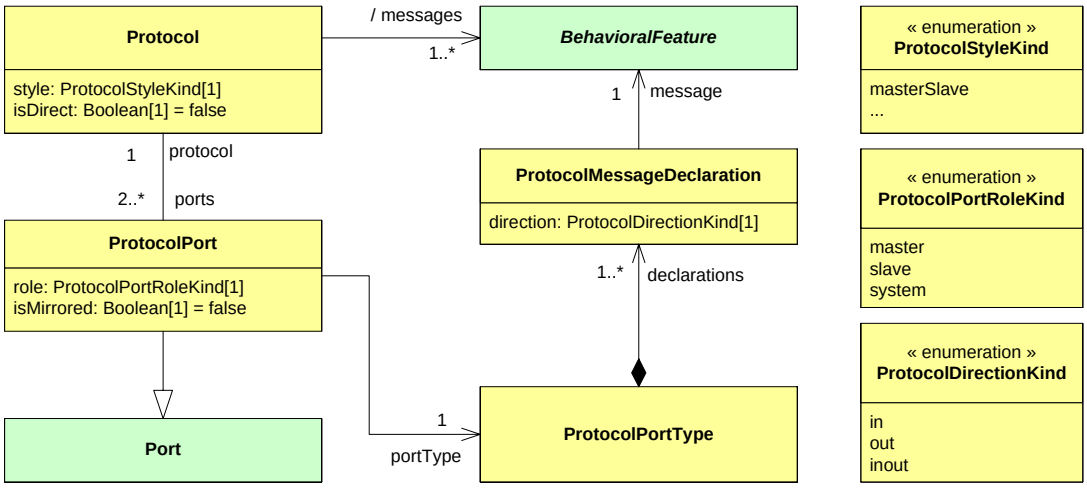


Figure 4.1: Protocol metamodel.

"mirrored" direction.

direction	operation	signal	mirrored
in	provided	reception	out
out	required	sending	in
inout	both dir.	both dir.	inout

Table 4.1: Interpretations of the direction attribute.

A protocol specifies legal (partial) orderings on the occurrences of protocol messages. It can be expressed in natural language, interaction diagram, activity diagram, or (protocol) state-machine.

4.1.3 Protocol profile

Figure 4.2 contains a UML profile for protocols. It is applied in the next section.

4.2 Acquisition system

4.2.1 System overview

We have chosen a very simplified and partial data acquisition system as a running example for the thesis. This is a component-based description which highlights protocol-based communications. The system consists of two processors getting information from two sensors and storing data in one of its three memories. The choice of the target memory is left unspecified in this presentation. Note that this system is a "black hole": information is sinking into memory but never read. Describing stored data accesses and their exploitations would have brought no new interesting modeling issues.

Figure 4.3 is an informal representation of the system. It shows a network of communicating processing units. The clouds in the figure indicate communications ruled by protocols. The

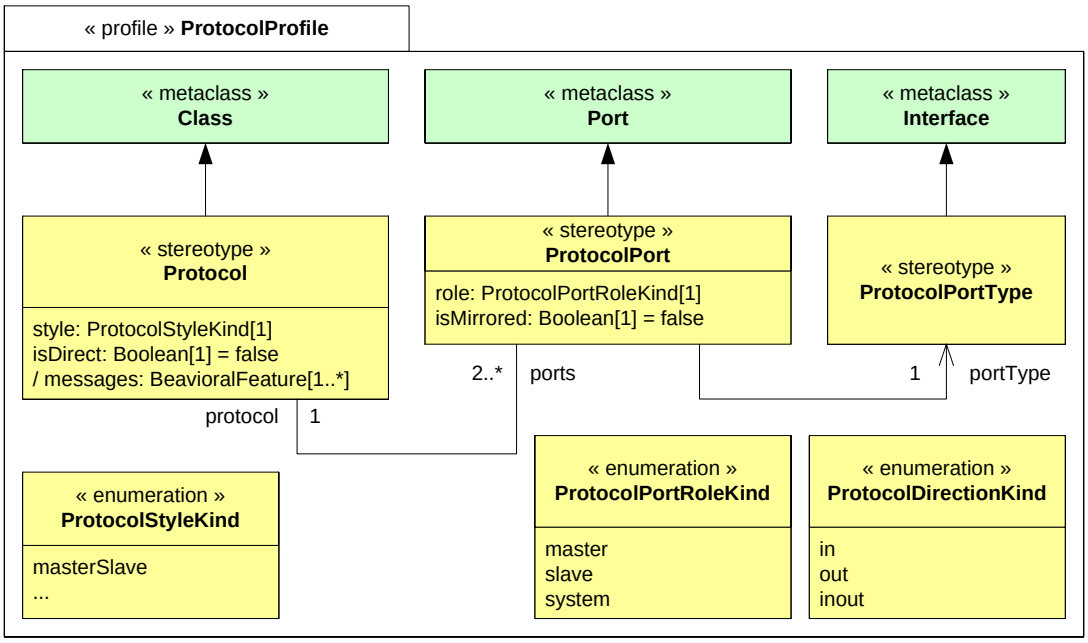


Figure 4.2: Protocol profile.

acquisition protocol drives a simple point-to-point communication. The data saving protocol is more complex: it addresses a two-to-three communication.

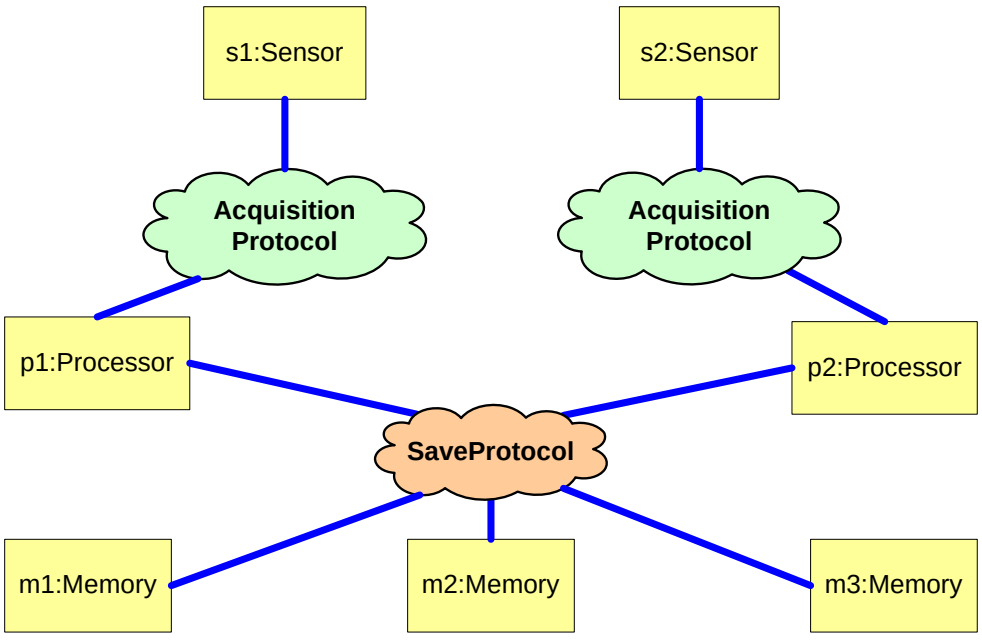


Figure 4.3: A simple acquisition system.

System model revised

When interconnecting IPs, connections are generally 1-to-1, possibly using busses that are themselves modeled as components. So, we propose a revised version in figure 4.4. The Bus has been added. Its color and the color of its adjacent edges (light orange) suggest that they all participate in the data saving protocol. Similarly, the green edge between a sensor and a processor denotes a communication respecting the acquisition protocol.

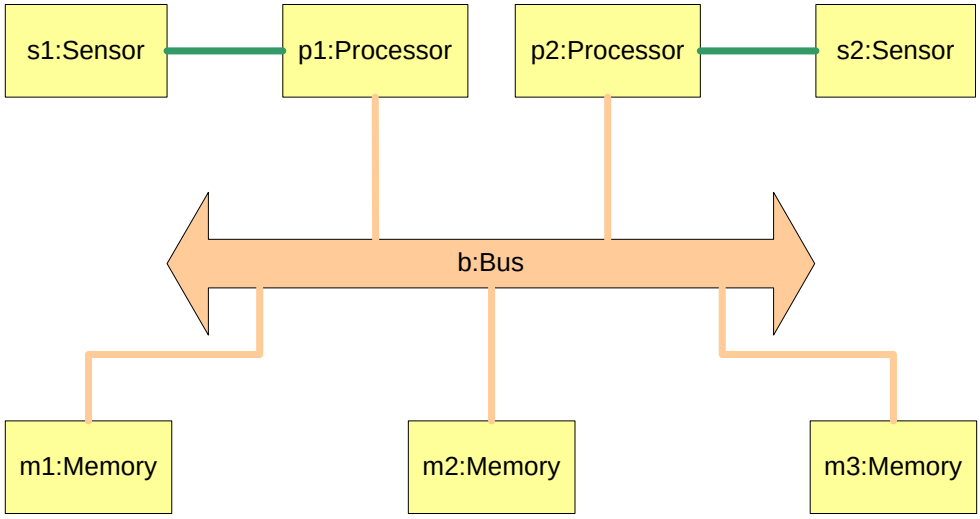


Figure 4.4: A simple acquisition system (revised version).

4.2.2 Specification of the protocols of the application

We focus on special cases, we have no intention to address generality. The two protocols are specified in conformance with the metamodel given in subsection 4.1.2. We have taken some liberties with the concrete syntax, especially in the notation of protocol port types. Giving behavioral features prefixed by a direction is more convenient than the UML representation of interfaces, where provided and required interfaces are separate. We also make use of stereotype property compartments as proposed in SysML.

Protocol AcqP

- *Purpose*: On-demand acquisition of data in the sensor and transfer to the processor.
- *Behavior*: The protocol imposes the alternation of messages SAMPLE and TRANSFER. The first message causes the sampling of a new value by the sensor, the second causes the effective transfer from the sensor to the processor.

Note that, at this point, nothing has been said about who is doing what. Other prescriptions are given in figure 4.5. The Protocol AcqP specifies that we adopt a master-slave control, the connection master-slave is direct, and the messages are SAMPLE and TRANSFER. The ProtocolPorts explain the role of each partner: the master (*i.e.*,the initiator of the communication) is the processor, the slave is the sensor. The type of the ProtocolPort M_A is AcqT, so

that message SAMPLE is an outgoing message from M_A, while message TRANSFER is incoming. The messages have the opposite directions at S_A because the type of S_A is `mirrored` AcqT.

At this abstract level, we do not mention whether messages are Operations or Signals.

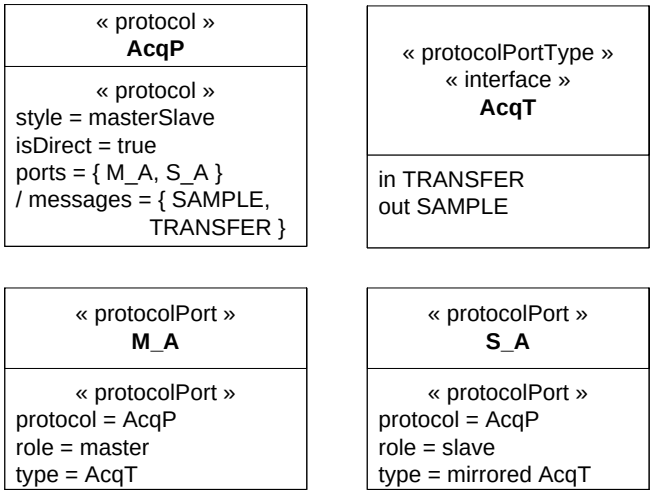


Figure 4.5: Acquisition protocol.

Protocol SaveP

- *Purpose*: Storing data contained in the processor into a memory. The choice of the memory is left unspecified at this modeling level.
- *Behavior*: The protocol is more complex because it involves two masters and three slaves (this information is extracted from figure 4.6). Informally, a master will first request for the bus (BREQx), wait for the bus grant (GRANTx), set the address (ADDR) and the data (DATA). One SELx then occurs. The value of x results from the chosen address in an unspecified way. DONEx indicates the end of the saving.

Note that we have two different protocol port types (MSaveT and SSaveT). The former types the master ports whereas the latter types slave ports. In figure 4.6, property ports of protocol SaveP has been omitted. It can be deduced from the structure diagram in figure 4.7.

4.2.3 Architecture model

Figure 4.7 is a model of the acquisition system including protocols. Stereotyped ports are represented by special notations explained in the legend. The two protocols are not explicit in the figure. Protocol aspects appear only through the protocol port types and the iconic representations of the protocol ports.

4.2.4 Components

Connectors and ports in figure 4.7 are high-level representations of the actual connections. So, we have to specify how to refine these model elements. This can be done by providing a

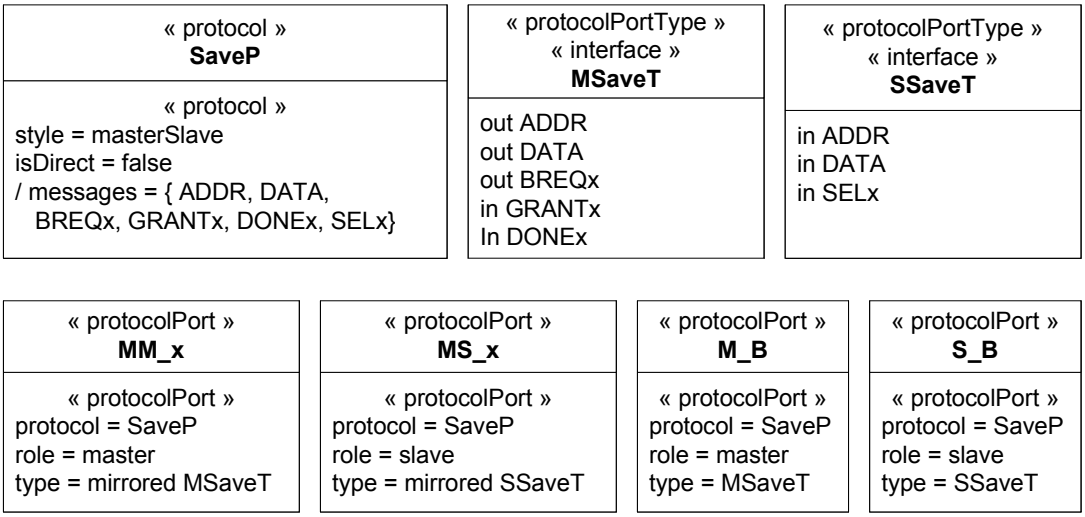


Figure 4.6: Save protocol.

black-box view of each component along with a port mapping. The component exposes only its (actual) *component ports*. The mapping links component ports to protocol messages.

Processor

Figure 4.8 specifies the component Processor with its ports (left part of the figure). The mapping is given in a tabular form (right-hand side).

Figure 4.9 shows how the high-level view (upper part of the figure) can be interpreted (lower part).

Memory

The component ports and the port map of the component Memory are given in figure 4.10.

Sensor

The connections of the component Sensor are described in figure 4.11.

Bus

Component Bus involves more protocol ports. Figures 4.12 represents its ports and the port map.

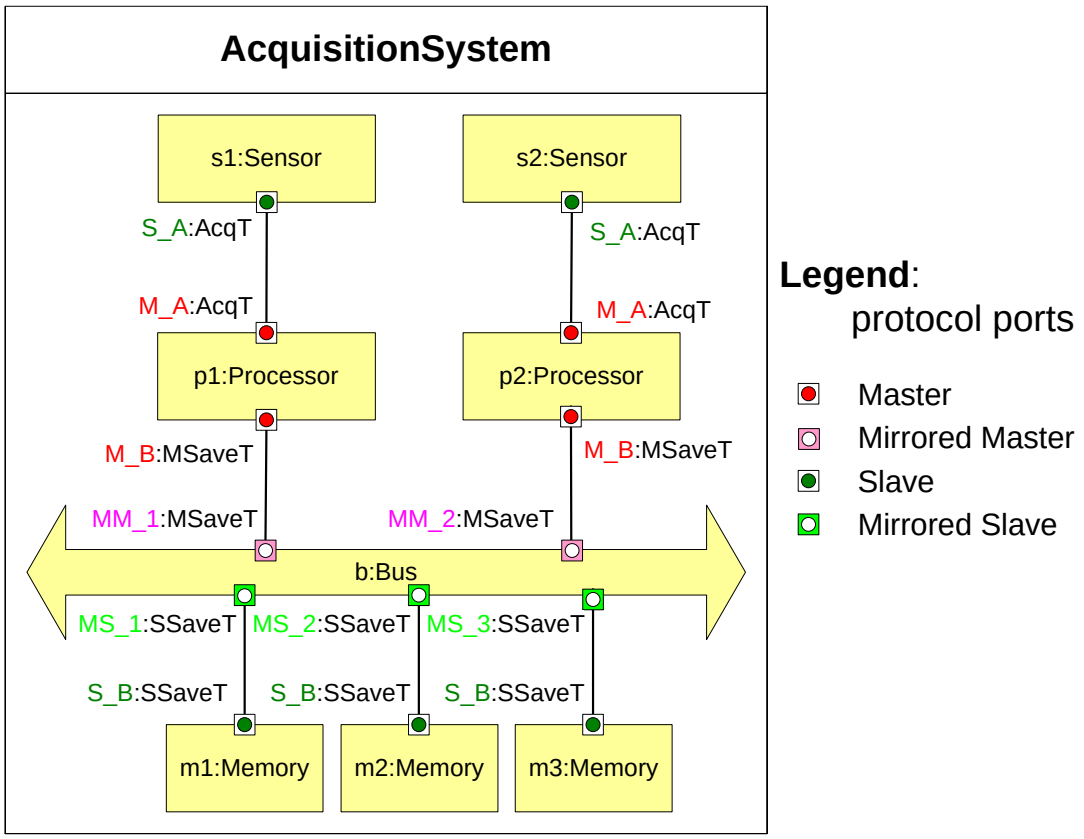
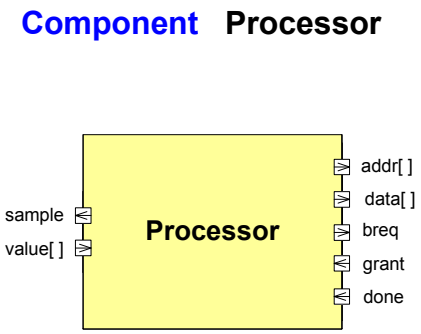


Figure 4.7: Architecture of the acquisition system.



PortMap		
Protocol Port	Message	Component Port
M_A (master)	SAMPLE	sample
	TRANSFER	value
M_B (master)	ADDR	addr
	DATA	data
	BREQx	breq
	GRANTx	grant
	DONEx	done

Figure 4.8: Component 'Processor'.

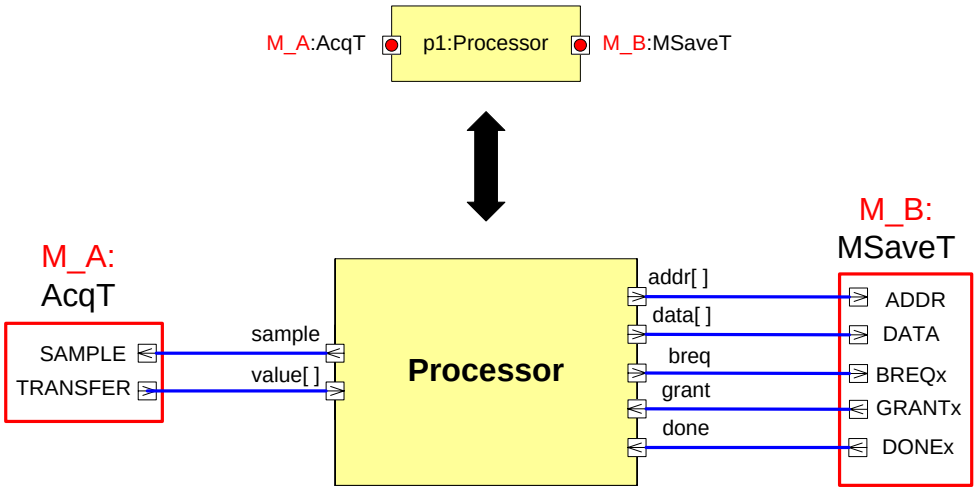


Figure 4.9: Component 'Processor': port mapping view.

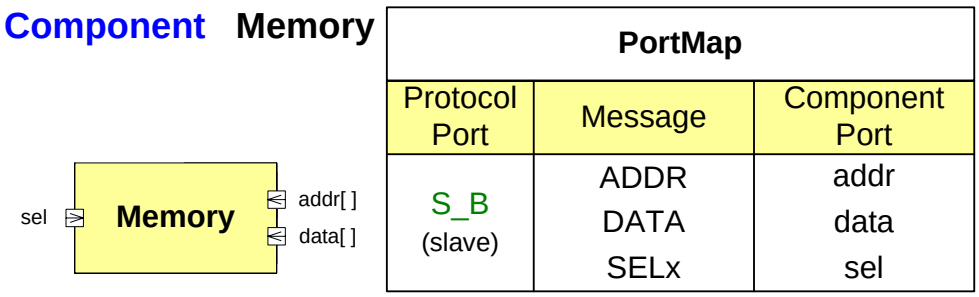


Figure 4.10: Component 'Memory'.

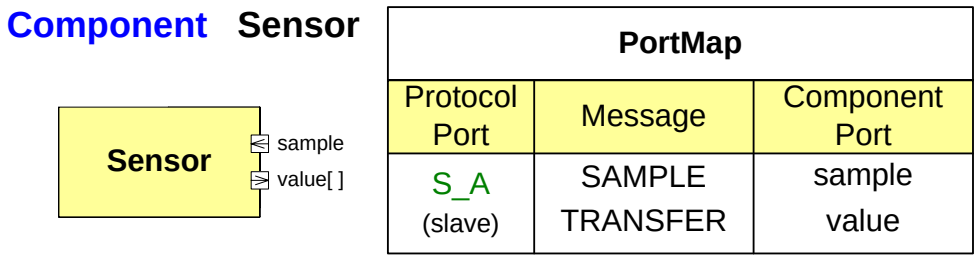
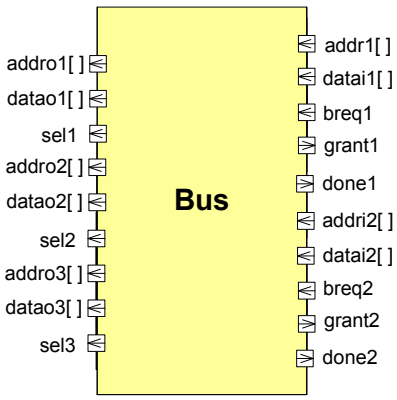


Figure 4.11: Component 'Sensor'.



PortMap		
Protocol Port	Message	Component Port
MM_1 (mirrored master)	ADDR	addri1
	DATA	datai1
	BREQx	breq1
	GRANTx	grant1
	DONEx	done1
MM_2 (mirrored master)	ADDR	addri2
	DATA	datai2
	BREQx	breq2
	GRANTx	grant2
	DONEx	done2
MS_1 (mirrored slave)	ADDR	addro1
	DATA	datao1
	SELx	sel1
MS_2 (mirrored slave)	ADDR	addro2
	DATA	datao2
	SELx	sel2
MS_3 (mirrored slave)	ADDR	addro3
	DATA	datao3
	SELx	sel3

Figure 4.12: Component 'Bus'.

4.3 Esterel modeling

In this section we propose to represent the structural aspects of the acquisition system in Esterel. This exercise will highlight the facilities offered by this language in architecture modeling.

4.3.1 Data units

In the data unit ApplTypes we gather the all the application-specific types. According to the modeling level, we provide two different implementations of this unit.

For a TLM description it is sufficient to say that the address and the data types are natural numbers, hence the use of the **unsigned** predefined type.

```
data ApplTypes:
  type Data_t = unsigned;
  type Addr_t = unsigned;
end data
```

For a RTL description, we fix the size of each type. Here we choose bitvectors (**bool**[ ]).

```
data ApplTypes:
  type Data_t = bool[8];
  type Addr_t = bool[16];
end data
```

In both cases, the rest of the program refers to these types only by their name.

4.3.2 Interfaces specifications

In the interface units we introduce signals and ports. In Esterel, a signal is the basic communication support, and a port is a collection of signals with their direction. A ProtocolPortType can be represented by an (Esterel) interface.

For the acquisition protocol

```
interface AcqT:
  extends ApplTypes;
  input Value: Data_t;
  output Sample;
end interface
```

The statement '**extends** ApplTypes' imports the application types defined in the data unit `ApplTypes`. `Value` is an input valued signal whose type is `Data_t`, whereas `Sample` is an output pure signal.

For the data saving protocol

```
interface CommonT:
  extends ApplTypes;
  output Data: Data_t;
  output Addr: Addr_t;
end interface
```

The interface CommonT gathers two signals (Data and Addr) that are common to the two protocol port types MSaveT and SSaveT.

```
interface MSaveT:
  extends CommonT;
  input Grant, Done;
  output Breq;
end interface
```

The statement '**extends** CommonT' imports an interface and thus all the signals contained in this interface.

```
interface SSaveT:
  extends mirror CommonT;
  input Sel;
end interface
```

Esterel offers the facility (use of the keyword '**mirror**') to import an interface while reversing the direction of the signals.

4.3.3 Modules

Component representation

Esterel modules can represent the application components. In this section we specify the interface only. The behavioral part is hidden. The Esterel ports in the modules below correspond to the protocol ports.

```
module Processor:
  port M_A: AcqT;
  port M_B: MSaveT;
  // behavioral part
end module

module Sensor:
  port S_A: mirror AcqT;
  // behavioral part
end module

module Bus:
  port MM_1: mirror MSaveT;
  port MM_2: mirror MSaveT;
  port MS_1: mirror SSaveT;
  port MS_2: mirror SSaveT;
  port MS_3: mirror SSaveT;
  // behavioral part
end module

module Memory:
  port S_B: SSaveT;
  // behavioral part
end module
```

Architecture

The components have now to be connected. Figure 4.13 is a screen copy of the architecture diagram built with Esterel Studio.

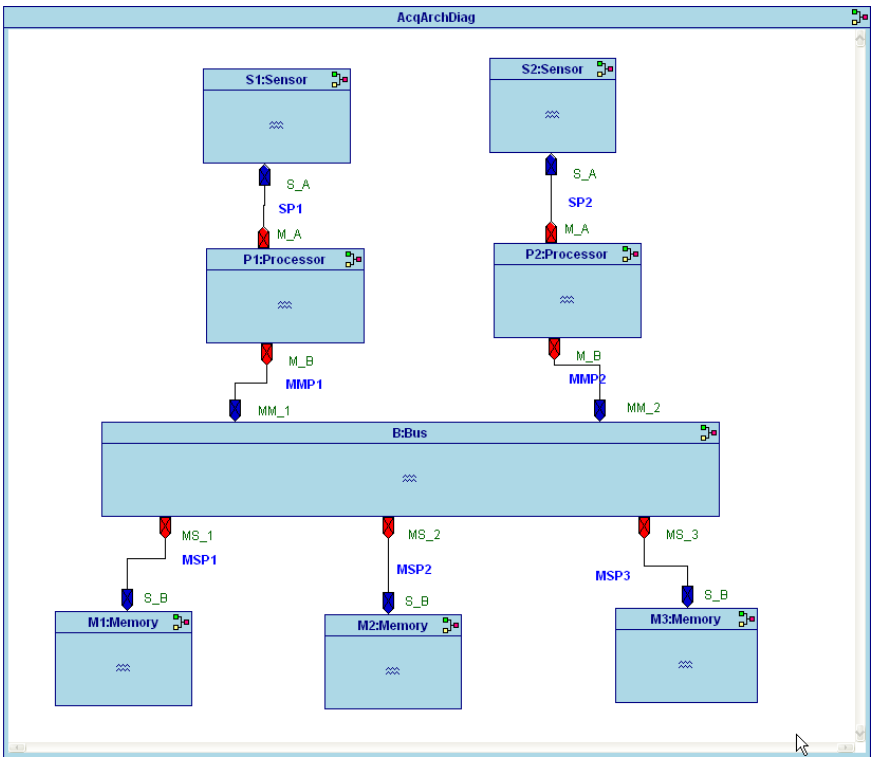


Figure 4.13: Esterel Architecture Diagram of the application.

The following module is the textual translation of the architecture diagram. It better shows how Esterel works. First, this module is the top-level module (keyword **main**). In this simplified version of the acquisition system there is no input/output declaration (the connections with the environment have not been specified). All the modules representing the application components are instantiated (keyword **run**) and run in parallel (|| operator). Local ports are defined (SPx, MMPx, and MSPx), they represent the connectors. The port mapping is done by signal and port renamings. The connections are specified in the braketed part of the **run**. The syntax is *actual element / formal element*.

```
main module Application:
    // header: connections with the environment
    //         omitted
    signal
        port {SP1,SP2}: AcqT,
        port {MMP1,MMP2}: MSaveT,
        port {MSP1,MSP2,MSP3}: SSaveT
    in
        run S1/Sensor [SP1/S_A]
    ||
        run S2/Sensor [SP2/S_A]
    ||
        run P1/Processor [SP1/M_A,MMP1/M_B]
    ||
        run P2/Processor [SP2/M_A,MMP2/M_B]
```

```
    ||
        run M1/Memory [MSP1/S_B]
    ||
        run M2/Memory [MSP2/S_B]
    ||
        run M3/Memory [MSP3/S_B]
    ||
        run Bus [MMP1/MM_1,MMP2/MM_2,
                 MSP1/MS_1,MSP2/MS_2,MSP3/MS_3]
     end signal
end module
```

It appears that the Esterel program is an almost direct transposition of the architecture shown in figure 4.7.

4.4 VHDL modeling

Like Esterel, VHDL allows type declarations, component interface definitions (*entities*), behavior specifications, and component assembly (*architecture*). Only the structural aspects are illutrated in this section.

4.4.1 Application types

VHDL is a language with strong typing. It supports hardware-oriented types, like bit vectors of any size, and thanks to its IEEE–1164 library, can represent multi-valued logics (`std_logic` type). The definition of user's types, borrowed from ADA, avoids implicit conversions and bad mixture of data. For the Acquisition system, we define two new sub-types of `std_logic_vector`:

```
library ieee;
use ieee.std_logic_1164.all;
package ApplTypes is
  subtype Data_t is std_logic_vector(7 downto 0);
  subtype Addr_t is std_logic_vector(15 downto 0);
end ApplTypes;
```

4.4.2 Component specifications

All the entity specifications include a '`use`' statement to access the types definitions:

```
use work.ApplTypes.all;
```

A component interface specifies the ports through which the component communicates with its environment. A signal is declared with a direction, a type, and an optional initial value.

```
entity sensor is
  PORT (
    S_A_Value: out Data_t;
    S_A_Sample: in Bit);
end sensor;

entity memory is
  PORT (
    S_B_Addr: in Addr_t;
```

```
      S_B_Data: in Data_t;
      S_B_Sel: in Bit);
end memory;

entity processor is
  PORT (
      M_A_Value: in Data_t;
      M_A_Sample: out Bit;
      M_B_Addr: out Addr_t;
      M_B_Data: out Data_t;
      M_B_Breq: out Bit;
      M_B_Grant: in Bit;
      M_B_Done: in Bit);
end processor;

entity a_bus is
  PORT (
      MM_1_Addr: in Addr_t;
      MM_1_Data: in Data_t;
      MM_1_Breq: in Bit;
      MM_1_Grant: out Bit;
      MM_1_Done: out Bit;
      MM_2_Addr: in Addr_t;
      ...
      MM_1_Done: out Bit;
      MS_1_Addr: out Addr_t;
      MS_1_Data: out Data_t;
      MS_1_Sel: out Bit;
      MS_2_Addr: out Addr_t;
      ...
      MS_3_Sel: out Bit);
end processor;
```

The contents of the declarations is the same as the Esterel ones. It appears that the Esterel ports and interface units make the declarations more concise and readable.

4.4.3 Architecture

The description of an architecture is also similar to the Esterel description.

```
1  architecture AcquisitionSystem is
2  component sensor
3    PORT (
4       S_A_Value: out Data_t;
5       S_A_Sample: in Bit);
6  end component;
7  component processor
8    PORT (
9       ...);
10 end component;
11 component a_bus
12   PORT (
13      ...);
14 end component;
15 component memory
```

```
16    PORT (
17       ...);
18  end component;
19
20  signal SP1_Value, SP2_Value: Data_t;
21  signal SP1_Sample, SP2_Sample: Bit;
22  signal MMP1_Addr, MMP2_Addr: Addr_t;
23  ...
24  signal MMP1_Done, MMP2_Done: Bit;
25  signal MSP1_Addr, MSP2_Addr, MSP3_Addr: Addr_t;
26  ...
27  signal MSP1_Sel, MSP2_Sel, MSP3_Sel: Bit;
28
29  begin
30    S1:sensor port map (
31      S_A_Value => SP1_Value;
32      S_A_Sample => SP1_Sample );
33    S2:sensor port map (
34      S_A_Value => SP2_Value;
35      S_A_Sample => SP2_Sample );
36    P1: processor port map (
37       ...
38       );
39    P2: processor port map (
40       ...
41       );
42    M1: memory port map (
43       ...
44       );
45     ...
46    M13: memory port map (
47       ...
48       );
49    B : a_bus port map (
50       ...
51       );
52  end architecture AcquisitionSystem;
```

Firstly, the component signatures are declared (lines 2–18). Then signals used in communications are declared (lines 20–27). The architecture itself is specified in lines 29 to 51. The components are instantiated and their connections are given by a `port map`. Of course, many VHDL programing environments propose a graphical interface in which components can be instantiated by simple drag-and-drop and then interconnected.

4.5 SystemC modeling

The Esterel and VHDL descriptions in the previous sections rely on signals. This is usual in RTL descriptions. We consider now a TLM description programmed in SystemC. Note that this choice does not mean that Esterel cannot be used at the transactional modeling level. Here we describe the master, slave and bus modules for our running example of acquisition system with the focus on their transactional ports. This focus will help us to better understand the TLM style of modeling electronic systems.

4.5.1 Transaction level modeling

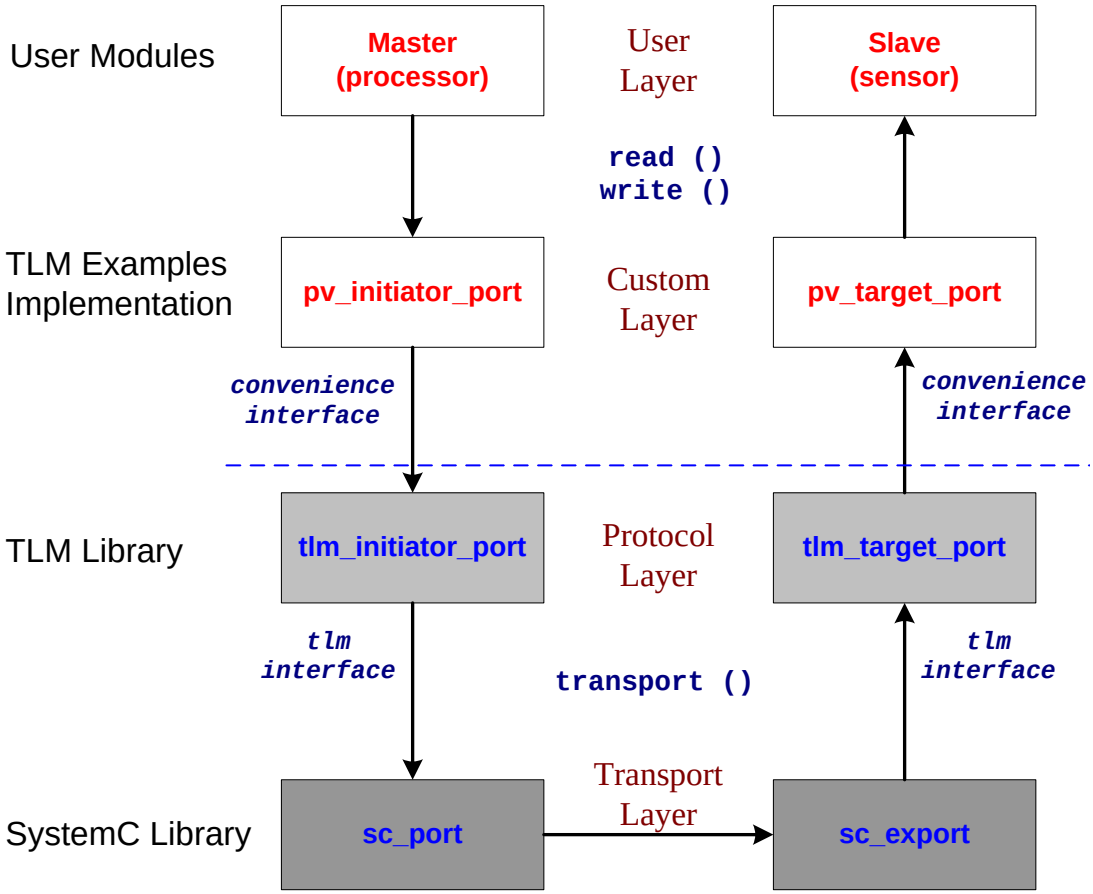


Figure 4.14: Abstract Modeling using TLM Library

The advantages of modeling at the transaction level are evident in figure 4.14. Use of SystemC and TLM libraries provides us with the predefined function modules which give an abstract view to the computation and communication in the system. At the lowest level, SystemC library provides simple structures like `sc_port` and `sc_export` to represent the interfaces of the structure of a component. This is the basic modeling requirement to implement the co-design paradigm (*i.e.*,to represent structures using software code). These port modules from the basic SystemC library correspond to detailed views of the design and as such can be used at register transfer level. Communication between the components at this modeling level is through transport function calls.

With the passage of time, SystemC proved its worth as a co-design and co-simulation language and further improvements were made by adding TLM libraries. This addition provided the system designers to implement models at abstract levels like TLM. Here the behavior representation gave the application layer programmers simple functions like `read()` and `write()` to implement complex communication transactions. In our present model, we have added a *custom layer* to further simplify these implementations. This layer consists of predefined

implementations of key communication modules and is present in the OSCI TLM examples. It consists of modules like fully functional initiator and target ports, routers and arbiters to represent the bus components. Such a modular and library-based approach of designing abstract systems provides the ease that we just have to pick the required module and integrate it into our system model. Quite evidently, this approach gives us easier ways to model systems than the alternative Esterel and VHDL approaches. Conclusively such an approach facilitates with the design of communication modules and allows the designers to concentrate their efforts on the computational modules.

4.5.2 Modules

Here, we present the structural aspects of the SystemC TLM model of the acquisition system. Full code of the acquisition system implemented in SystemC is given in the Appendix B.2.

processor

```
1  #include <systemc.h>
2  #include "tlm.h"
3  #include "pv_initiator_port.h"
4  #include "types.h"
5
6  class processor: public sc_module {
7  public:
8    SC_HAS_PROCESS(processor);
9    processor(sc_module_name module_name);
10   ~processor();
11   // Interfaces
12   pv_initiator_port< Addr_t, Data_t > M_A;
13   pv_initiator_port< Addr_t, Data_t > M_B;
14 private:
15   Data_t lData;
16   Addr_t lAddr;
17 };
```

We firstly describe the interface of the processor in SystemC modeled at transactional level. The processor class contains two transactional ports M_A (line 12) and M_B (line 13). The M_A port communicates with the sensor component and the M_B port communicates with the bus. Both the ports are *initiator ports* as they are the ones to initiate the transactions. The initiator and target port are the terms standardized by SystemC and are also adopted by the IP-Xact standard. These ports are defined in the `pv_initiator_port.h` header file (line 3). They are presented in the supplementary example code given in the TLM library provided by the OSCI organization. This port definition is the SystemC class implementing the interfaces and channels from the TLM standard library (like `tlm_transport_if` [Ayn09]). These ports use the custom data types defined in the `types.h` header file just as we defined earlier for Esterel program:

```
typedef unsigned int Addr_t;
typedef unsigned int Data_t;
```

These types are used as input arguments by these transactional initiator ports to perform the read and write operations. In the read operation, the data variable provided is used to get the return value for the particular address location.

Bus

On the bus side, we have chosen the pv_router class implemented in the examples of the TLM distribution library. Here pv_router class determines dynamically the number of target ports (slave module ports) attached to the bus and marks address range for those ports given in the map file. This map file has entries like `mytop.umem0.S_B 0000 1000`, where `mytop` is the instantiation of the main module and `umem0` is the instance 0 of memory module. This instruction assigns the memory module's slave (target) port with a starting address (0000), and the total address range size (1000). Later on, all the transaction requests for the slave modules are routed to the respective slave instances. This routing mechanism is implemented in the `pv_router` class provided in the TLM library.

```
1  #include <systemc.h>
2  #include "pv_router.h"
3  #include "types.h"
4
5  // this class has 2 target ports and 3 initiator ports
6  typedef pv_router< Addr_t, Data_t > basic_router;
7
8  class bus:  public basic_router {
9  public:
10   bus (sc_module_name module_name, const char* mapFile):
11     basic_router (module_name, mapFile) {}
12   void end_of_elaboration() {
13     basic_router::end_of_elaboration();
14     cout << name() << " constructed." << endl;
15   }
16 };
```

Memory

Lastly, the interface of slave memory module consists of a target port and its read/write functions. The target port S_B (line 16) is connected to the bus's initiator port during the instantiation phase. Two functions `write` (lines 19–23) and `read` (line 25–29), defined in the TLM library's header file `pv_target_port.h`, provide the required reception functionality. Whenever a read or write function call is initiated by a master module, the respective functions in slave module are invoked. These functions can then use the input parameters and in case of read function, the value stored in the 'data' variable is updated. All other parameters are optionally present based on the initiator port's implementation. Finally, the memory module contains a data array (line 31) to describe the internal storage cells of the memory. The behavior of the different modules and TLM transaction calls are explained in detail in chapter 7.

```
1  #include <systemc.h>
2  #include "tlm.h"
3  #include "pv_slave_base.h"
4  #include "pv_target_port.h"
5  #include "types.h"
6
7  class memory:
8    public sc_module,
9    public pv_slave_base< Addr_t, Data_t >
10 {
```

```
11  public:
12    SC_HAS_PROCESS(memory);
13    memory(sc_module_name module_name);
14    ~memory();
15    // bus side interface
16    pv_target_port< Addr_t,Data_t > S_B;
17
18    // bus functions
19    tlm::tlm_status write (
20      const Addr_t &addr, const Data_t &data,
21      const unsigned int byte_enable = tlm::NO_BE,
22      const tlm::tlm_mode mode = tlm::REGULAR,
23      const unsigned int export_id = 0);
24
25    tlm::tlm_status read (
26      const Addr_t &addr, Data_t &data,
27      const unsigned int byte_enable = tlm::NO_BE,
28      const tlm::tlm_mode mode = tlm::REGULAR,
29      const unsigned int export_id = 0);
30  private:
31    Data_t iMemory[0x1000];
32  };
```

The slave sensor module is similar to the memory module. Its class is omitted but available in Appendix B.2.

Architecture

Once these components are modeled, they are connected together in a main module that instantiates all other components. In our Acquisition example, the module `top` (declared in the `top.h` file) is used for this purpose. Here we provide only some part of the code with dotted lines marking the omitted code. In the class `top`, we define local SystemC signal (line 7) which connects with the simulation interface of the processor module. In the constructor of the main module, we define a one-time execution thread `Simulation` (line 11) that runs the simulation patterns (described later in chapter 7). This precedes the instantiation of all the components (lines 13–17) in the design. Finally we define the port connections (bindings) between the various component (lines 20–27). With this the constructor execution ends and control passes to the SystemC kernel for elaboration phase.

```
1  #include <systemc.h>
2  #include "sensor.h"
3  ...
4
5  SC_MODULE(top) {
6    public:
7      sc_signal<int>  myStart0;
8      ...
9
10   SC_CTOR(top) {
11     SC_THREAD(Simulation);                  // Thread Declaration
12
13     sensor i_sensor_0 ("usensor0");         // instantiation
14     processor i_proc_0 ("uproc0");
15     bus i_bus ("ubus");
```

```
16        memory i_mem_0 ("umem0");
17        ...
18        cout<< "Instantiated..." << endl;
19
20        i_proc_0->Start(myStart0);          // Interface Binding
21        ...
22        i_proc_0->M_A(i_sensor_0->S_A);
23        ...
24        i_proc_0->M_B(i_bus->target_port);
25        ...
26        i_bus->initiator_port(i_mem_0->S_B);
27        ...
28        cout<< "Ports Binding done" << endl;
29     }
30  };
```

4.6 Conclusion

In this chapter, we have explored the modeling and programming techniques used with different languages. Designing systems with high-level modeling languages through the structure/block diagrams is easy to understand and implement. These languages are design concepts intensive and need less programming efforts. UML is an obvious choice for modeling at such levels using various diagrams. Esterel language is the next ideal for such a programming with constructs (like mirroring, port grouping) to deal with abstraction. Low level modeling is more textual and programming intensive. These models contain more details about the structure and are usually programmed in languages like VHDL or SystemC.

IP-Xact is another standard, to be discussed in the next chapter, which comes as a blend of high and low level modeling features. Here we focused on the structural aspects while comparing these modeling paradigms and the functional aspects with focus on these languages will be discussed in chapter 7.

Chapter 5

SPIRIT IP-XACT Metamodel

Contents

In this chapter, we propose a domain view (or metamodel) for IP-X*act. This metamodel has been elaborated from the* XML *Schema Definition (*XSD*) files of the* IP-X*act specification. It introduces the main concepts of the domain in a technology-independent way and serves as a reference model for the profile described in the next chapter. Along this chapter, the acquisition example illustrates how information is represented and stored in the* IP-X*act specification.*

5.1 Introduction

IP-Xact is a standard proposed by the consortium of major players in the electronics industry—like electronic design automation (EDA), semiconductor, electronic intellectual property (IP) providers, and system design communities—grouped in the SPIRIT Consortium[1] to provide with a well-defined and unified specification for the meta-data which represents the components and designs within an electronic system [SPI08]. The main goal of this specification is to enable delivery of compatible IP descriptions from multiple IP vendors; better enable importing and exporting complex IP bundles to, from and between EDA tools for SoC design (system on a chip design environments); better express configurable IP by using IP meta-data; and better enable provision of EDA vendor-neutral IP creation.

The IP-Xact specification is a mechanism to express and exchange information about an IP design and its required configuration. Indeed, the final format of a component is generally in its description in a synthesizable implementation language such as VHDL or Verilog. The information on the interface of these implementations, defining their connectivity and thus their integrability in a system, are embedded in the details of the code. Traditionally, a design engineer had to open these files and check if a given component can be connected to the target component, and these connections were then done by hand considering what signal entering or leaving a component must be connected to another. With the IP-Xact standard, the primary purpose is to facilitate this IP management, encapsulating these implementations in a description based on XML. Thus by defining a machine readable and an automated way of storing information about the interfaces of the components, it becomes possible to use graphical tools to manipulate and interconnect these components. Such CAD (Computer Aided Design) tools can then automatically generate RTL implementation of the *netlist* physically instantiating and interconnecting the different components.

The IP-Xact specification is formally defined in the XML schema definition (XSD) files which specify the structure of metadata. These schema files contain seven top-level schema definitions (each related to an important IP-Xact concept) for Component, Bus definition, Abstraction definition, Design, Abstractor, Generator chain, and Configuration.

5.1.1 Design Environment

Before going to the IP-Xact metamodel, an important term to understand is of the IP-Xact Design Environment (DE). An IP-Xact design environment is an IP-Xact design tool that enables the designer to work with IP-Xact design IPs through a coordinated front-end and IP design database. In fact, the DE coordinates a set of tools and IPs (or representations of those IPs as models), through the creation and maintenance of meta-data descriptions of the whole SoC such that its system-design and implementation flows are efficient and re-use centric. These tools create and manage the top-level meta-description of system design as shown in the figure 5.1 which is inspired from an example design environment given in the IP-Xact specification [SPI08]. Such a design environment can be graphical like the tools from Magillem Design Services[2] and coreTools from Synopsys[3] or XML tree structure representation like in Eclipse IP-Xact plug-in[4]. These design environment tools import the IP component descriptions from the vendor provided database along with the buses information (abstraction and bus definitions). Then the design files and design configurations can be created in the environment

[1] http://www.spiritconsortium.org
[2] http://www.magillem.com
[3] https://www.synopsys.com/dw/doc.php/ds/o/coretools_ds.pdf
[4] http://www.eclipse.org/dsdp/dd/ipxact/gettingstarted/QuickStart.html

or can be imported also. Finally, these design files are used to automatically generate the interconnection netlist files for the given abstraction level.

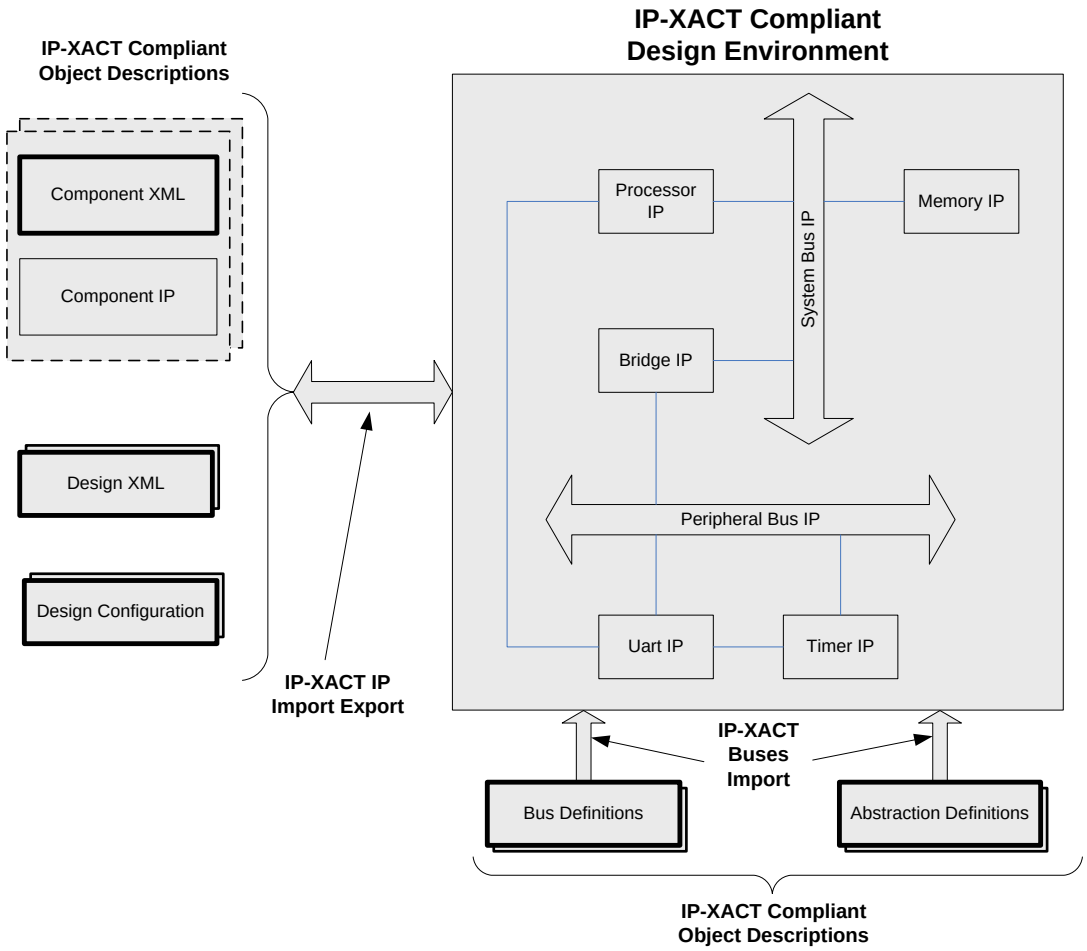


Figure 5.1: IP-XACT Design Environment.

5.1.2 IP-XACT Metamodel Overview

In this chapter, we focus on the most important concepts of our domain view considering the IP-Xact Component, Bus definition, Abstraction definition, Design, and Abstractor. The Generator chain and the Configuration parts are related to the IP code generation and netlist creation and thus, does not come in the direct focus of this work.

A *domain view* is a technology-independent representation of domain concepts and allows interactions with domain experts, which are not necessarily familiar with modeling languages like UML. The domain view also serves as a reference model to ensure that all concepts have been implemented in the chosen technology, *i.e.,*a UML profile in our case. Building a domain view before building a profile is considered as best practice in the profiling community [Sel07]. This domain view or metamodel is not provided in the IP-Xact specification by the SPIRIT Consortium and is one of the contributions of this thesis work.

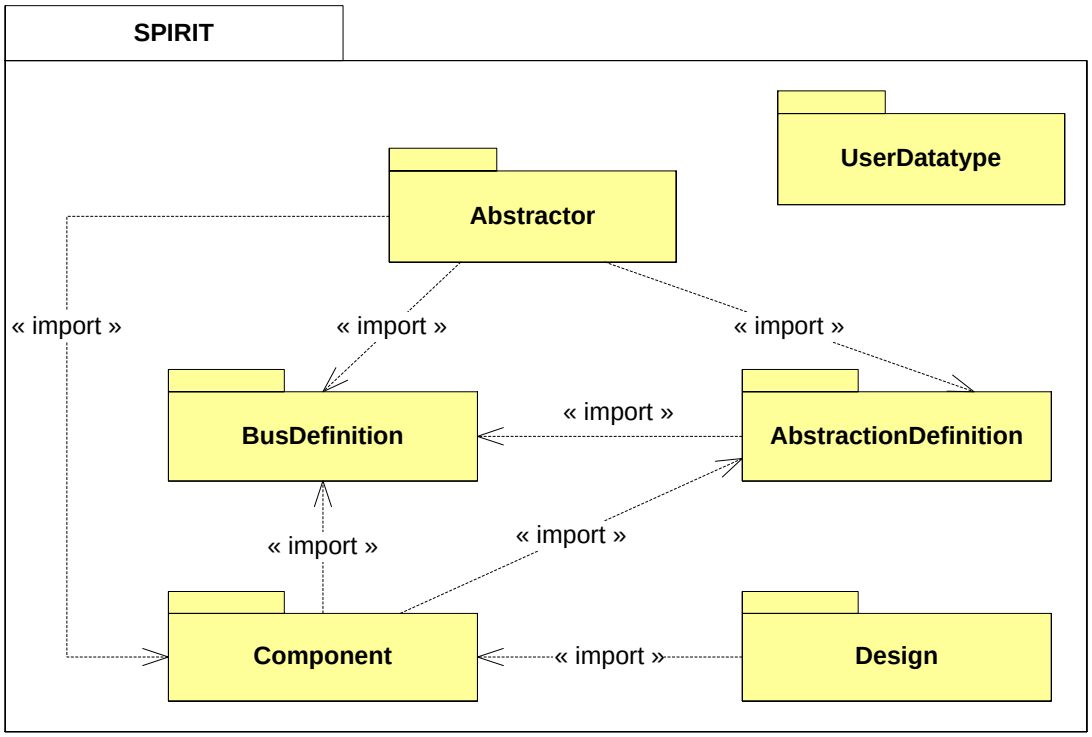


Figure 5.2: IP-XACT Metamodel Overview.

Figure 5.2 gives a broad overview of our domain view of IP-Xact specification. The principal IP-Xact concepts are shown as the packages grouping together the related elements. These packages are interdependent and interlinked, with the component and the bus information being in the center place. The component package contains the elements which describe the structure of an IP component and works in close collaboration with the bus elements represented by the bus definition and the abstraction definition. BusDefinition gives the basic information about the bus imported inside the system (like AMBA), whereas the AbstractionDefinition gives bus information related to a particular abstraction level (like TLM, RTL). The Abstractor package contains elements that are used to form a bridge between IP-Xact components implemented at different levels of abstraction. It requires the bus and component interfacing related informations to function. Finally, the Design package instantiates the components to depict the functioning electronic system.

An important point to note here is that all the key IP-Xact elements instantiated in the top level design environment (*i.e.*,the five packages given above) are uniquely identified by an identifier whose type is VersionedIdentifier. This unique identifier of the element consists of its name, the containing library (or the package it belongs to), its vendor and its version number and is also known as the VLNV of the component (where VLNV is the acronym for Vendor, Library, Name, and Version). VersionedIdentifier is defined as a data type in the `UserDataType` package of our domain view, as shown in figure 5.4. This package introduces user-defined data types which are frequently used in IP-Xact specification but are not available in UML. These data types include `PositiveNumber` (from 1 to infinity) and a `NameValuePair` type which is actually collection of a property name and its associated value. Similarly, there are several other optional attributes common among all these top level elements like the description

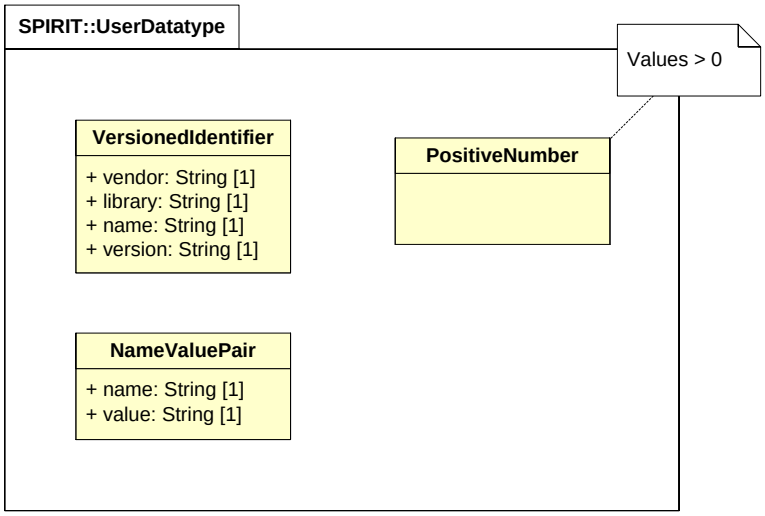


Figure 5.3: IP-XACT Custom Datatypes.

attribute which adds textual description to the element to precise its intended role and the vendorExtensions which gives the flexibility to tailor the elements beyond the scope of IP-Xact specification. For the sake of simplicity, we have omitted such attributes in our domain view.

In the further sections, the packages introduced before are explained one by one, in detail. First we sketch the specification of the Acquisition system in IP-Xact.

5.1.3 The Acquisition system in IP-Xact

An IP-Xact description consists of many model elements scattered over several structures or files. Table 5.1 indicates the names given to these model elements in IP-Xact and the place where they are specified. This may help the reader and will be illustrated with our Aquisition system example.

Model element	Name in IP-Xact	Described in
Architecture	Design	Design
Component	Component	Component
Protocol port	Bus interface	Bus definition
Component port	Component port	Component
Message	Logical port	Abstraction definition
PortMap	PortMap	Component
Port role	mode	Component

Table 5.1: Main IP-Xact model elements and their location.

5.2 Component

An IP-Xact component is the central placeholder for the objects meta-data. Components are used to describe cores (processors, co-processors, DSPs, etc.), peripherals (memories, DMA controllers, timers, UART, etc.), storage elements (memory, cache), and buses (simple buses,

multi-layer buses, cross-bars, network on chip, etc.). In IP-Xact, every IP is described as a component without distinction of type, whether it represents a computation core, a peripheral device, a storage element, or an interconnect. In general, components can be either static or configurable, where the static components can not be altered while the configurable components can have parameterized instances initialized in the design environment. Moreover, the IP-Xact components can also contain other IP-Xact components in a hierarchical fashion.

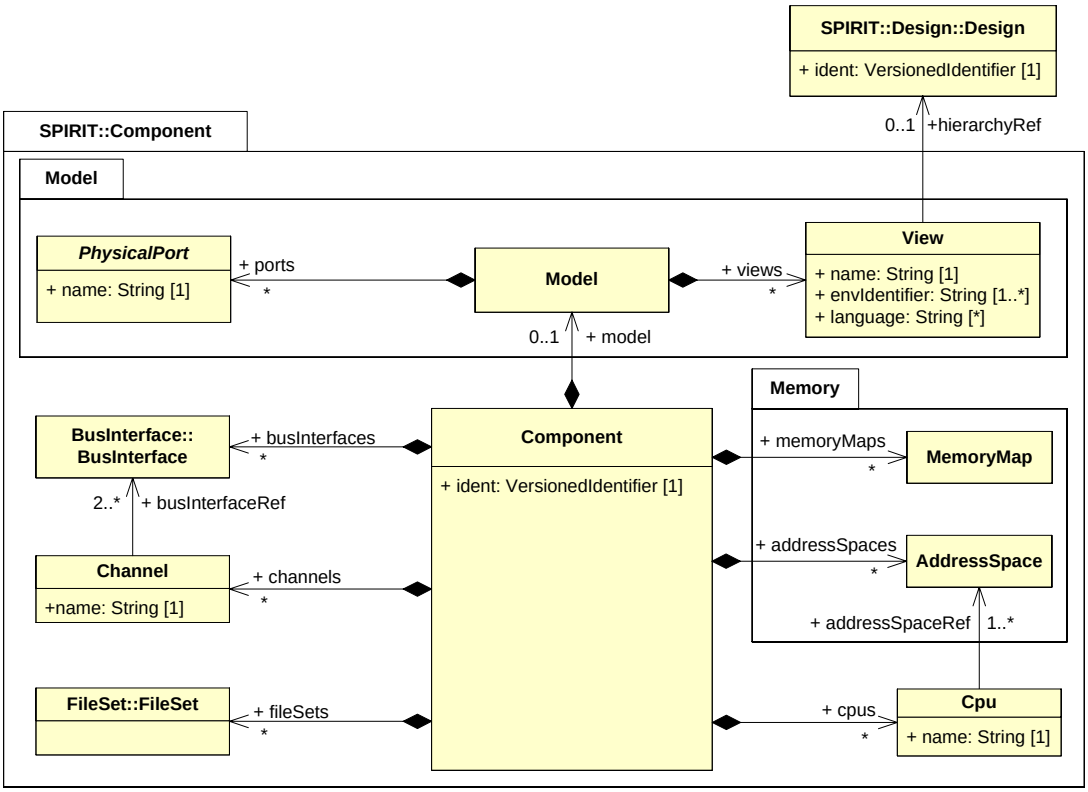


Figure 5.4: IP-XACT Component Metamodel.

Figure 5.4 shows the main features of our IP-Xact component metamodel: the interfaces and the memory hierarchy. The VLNV identifier, discussed earlier, is the only mandatory element of the component and all other elements are optional for the component specification. The IP-Xact component consists mainly of the model, bus interface and the description of the memory hierarchy (address spaces and memory mappings). We discuss them in detail in the following subsections.

Acquisition system: For this system four components are specified: Sensor, Processor, Memory, and Bus.

5.2.1 Model

Component Model is the basic building block of the component. It gives the concrete structure of a component describing the views and ports of the given component. The *view* mechanism is a provision for having several descriptions of the same component at different levels of

abstraction like RTL or TLM, shown in figure 5.4. A model may have many different views. As an example, an RTL view may describe the source hardware entity in VHDL along with its pin level interface information while the TLM view may define the source module behavior using C files along with its .h file interface information. The model view must contain at least one hardware environment in which this view applies, defined by the envIdentifier variable. The format of the variable string is `language:tool:vendor_extension`, with each piece being optional. The `language` part indicates this view may be compatible with a particular tool, but only if this language is supported in that tool and `tool` part indicates the tool name for which this view is targeted. An example value taken from the Leon 2 architecture of IP-Xact specification [SPI08] is `:osci.systemc.org:`. Having more than one envIdentifier indicates that the view applies to multiple environments. The hierarchyRef attribute references a Spirit design or configuration document that provides a design for the given component. It is required only if the view is used to reference a hierarchical design. View specifies the hardware description language used for it (like Verilog, VHDL or SystemC) by using the language attribute.

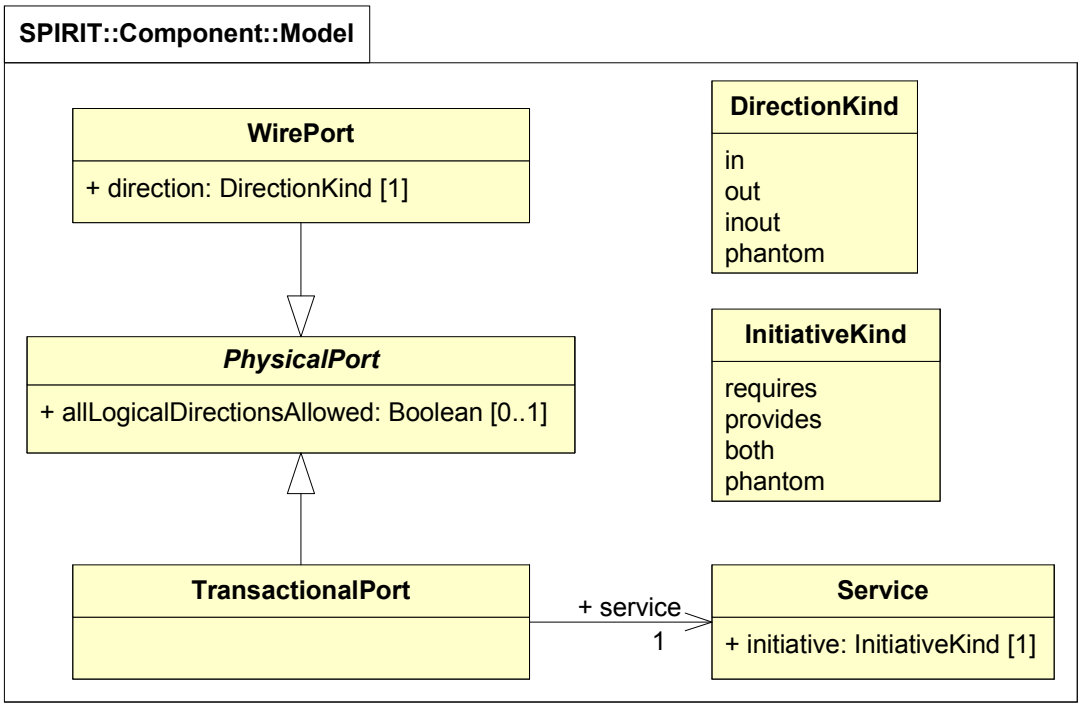


Figure 5.5: IP-XACT Physical Port Metamodel.

Other than the views, the IP-Xact component model also contains all the *ports* that are present in the component. The *Physical Ports* are represented by an abstract class. The concrete classes are given in the Component::Model package (figure 5.5). An abstract physical port can either be a RTL low level port called *WirePort* or a TLM transactional level port called *TransactionalPort*. The wire port can also be used in the transactional models to show non-bus type ports. These two ports together constitute the physical ports which are referred to by the port maps and the models. The structure of both the wire and the transactional ports is quite similar. We have *direction* of the port having values in, out, inout and phantom. The phantom port defined here is the port in the component port list, which does not correspond to ports of the implementation. Correspondingly, we have *service* in the transactional ports,

which is mandatory to define. Service defines the connection type initiative of the port like it requires or provides the connection. Service optionally gives the type of the ports represented by any predefined type, such as Boolean or some user-defined type, such as `addr_type`. The type can also be definition by providing a link to the file location where the type is defined *e.g.*,some SystemC include file containing the type definition. The optional attribute allLogicalInitiativesAllowed defines whether the port may be mapped to a logical port in an abstraction definition with a different initiative (see subsection 5.3).

Acquisition system

For each component, the associated XML file defines its (physical) ports. In this application, all ports are wire port, therefore implicitly Boolean. Some are arrays of Booleans, called *vectors* in IP-Xact. In this case the size and the bit ordering are given.

The information associated with two of the system components is given below in a tabular form.

Sensor :

port name	direction	vector	
		left	right
sample	in		
value	out	7	0

Processor :

port name	direction	vector	
		left	right
sample	out		
value	in	7	0
addr	out	15	0
data	out	7	0
breq	out		
grant	in		
done	in		

5.2.2 Bus Interface

BusInterface is the most important interface element of an IP-Xact component. It is a grouping of ports related to a function (*i.e.*,collaborate to a single protocol), typically a bus, defined by a *bus definition* and an *abstraction definition* (see next subsection). Components communicate with each other through their bus interfaces tailored for a specific bus. Bus interfaces provide the link between the component and the bus; they are shown in both figures 5.4 and 5.10 along with the component and the bus definitions respectively. Broadly, we can divide the bus interfaces into three main parts; *reference* to the buses, *mode* of the bus interface and the *port maps*. As bus interfaces are tailored for a specific bus, they have to mention the bus definition in the busType. The abstraction definition, being an abstract view of the bus, is not mandatory to mention and is referred to by the abstractionType attribute. The two bus definitions are discussed in detail in the subsequent sections. The bus interface and its relation with the ports and the definitions files is well depicted in the figure 5.6.

The second mandatory property of the bus interface is its interfaceMode. The busInterface's mode designates the purpose of the busInterface on this component. There are seven possible

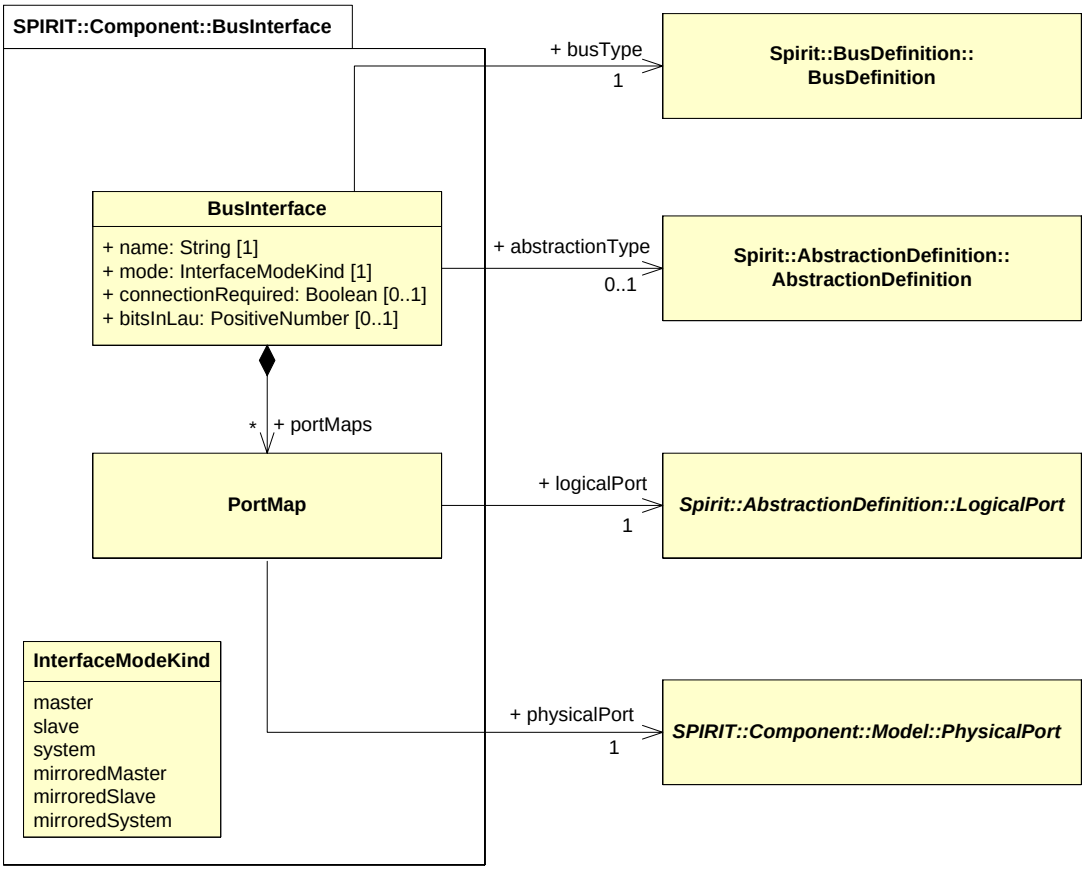


Figure 5.6: IP-XACT Component Bus Interface.

modes: three pairs of standard functional interfaces and their mirrored counterparts, and a monitor interface for component IP verification. These modes can be master, mirrored master, slave, mirrored slave, system, mirrored system and monitor. The *master interface* (or an initiator) is one that initiates transactions whereas the *slave interface* (or the target) is the one that responds to the transactions. The *system interface* is an interface that is neither a master nor slave interface, and allows specialized (non-standard) connections to a bus (*e.g.,*clock). The mirroring mechanism guarantees that an output port of a given type is connected to an input port of a compatible type, and vice versa. The *monitor interface* is a special interface that can be used for verification by gathering data from other interfaces. A monitor connection is a connection between a monitor interface and any other interface mode. Monitor connections are purely for non-intrusive observation of an interface. More than one monitor may be attached to the same interface.

The bus interfaces map the physical ports of the component to the logical ports of the abstraction definition (see figure 5.6). This mapping is done inside the optional portMap attributes. A bus interface can have indefinite number of port mappings. Once a port map is defined, both the logical and physical ports are mandatory to be provided. It is important to note that the same physical port may be mapped to a number of different logical ports on the same or different bus interfaces, and similarly the same logical port may be mapped to a number of different physical ports.

The optional connectionRequired Boolean attribute, if true, specifies that this interface shall be connected to another interface for the component integration to be valid. The bitsInLau attribute describes the least unit of memory addressable by the bus interface, with the default value being 8.

Acquisition system

Each component description specifies its bus interfaces, defines its ports (physical ports), refers to logical ports, and provides port maps. Bus interfaces are declared but their definition is delegated to other descriptions (bus definition and abstraction definition). Similarly logical ports are used in a port map but their specifications are given elsewhere in the abstraction definition. Only the physical ports and the port maps are effectively contained in the component description. The port map is a list of pairs < *logical port*,*physical port*>. This is illustrated on two components.

Sensor:

- Bus Interface: `S_A`, bus type = `Acq`, mode = slave
  - Logical ports: `TRANSFER`, `SAMPLE`
  - Port map: <`SAMPLE`,`sample`>, <`TRANSFER`,`value`>
- Physical ports: `sample`, `value`.

Processor:

- Bus Interface: `M_A`, bus type = `Acq`, mode = master
  - Logical ports: `TRANSFER`, `SAMPLE`
  - Port map: <`SAMPLE`,`sample`>, <`TRANSFER`,`value`>
- Bus Interface: `M_B`, bus type = `Save`, mode = master
  - Logical ports: `ADDR`, `DATA`, `BREQx`, `GRANTx`, `DONEx`
  - Port map: <`ADDR`,`addr`>, <`DATA`,`data`>, <`BREQx`,`breq`>, <`GRANTx`,`grant`>, <`DONEx`,`done`>
- Physical ports: `sample`, `value`, `addr`, `data`, `breq`, `grant`, `done`.

5.2.3 Memory

The memory of a component is of two types: *address spaces* and *memory maps*, shown also in figure 5.8. An AddressSpace specifies the addressable area as seen from master bus interfaces or from cpus whereas a MemoryMap specifies the addressable area as seen from slave bus interfaces. An AddressSpace is the logical addressable space of memory and each master interface can be assigned a logical address space. Address spaces are effectively the programmer's view looking out from a port of a master bus interface. Some components may have address spaces associated with more than one master interface while other components may have multiple address spaces. For instance, Harvard architecture processors have one address space for instruction and another one for code.

An address space as well as a memory map must provide the *range* and *width* of the memory. Range gives the address range of an address space and is expressed as the number

of addressable units of the address space. The size of an addressable unit is defined by the addressUnitBits element. Width is the bit width of a row in the address space expressed in whole numbers. Figure 5.7 illustrates an example of memory block. Some processor components require specifying a memory map that is local to the component. Local memory maps, expressed by the localMemoryMap element in the AddressSpace, are blocks of memory within a component that can be accessed and seen exclusively by the master bus interface viewing this address space. On the other side, a MemoryMap is the map of address space blocks on the slave bus interface and can be defined for each slave interface of a component.

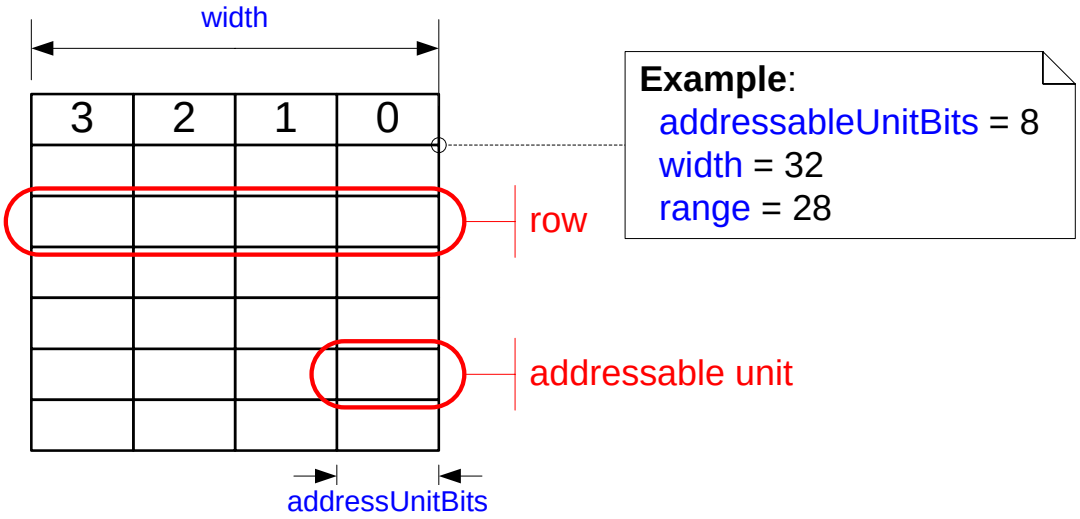


Figure 5.7: Example of memory block.

Other than the logical view of memories, both the address spaces and the memory maps are similar and are made up of *address banks* and *address blocks*. An addressable area can either be a single block or an address bank itself further decomposed into other banks or address blocks. The addressBlock element describes a single, contiguous block of memory that is part of a memory map. Its baseAddress element specifies the starting address of the block. The bank element allows multiple address blocks, or banks to be concatenated together horizontally or vertically as a single entity. The usage and the access attributes describe together the type of the memory. For an example, the ROM memory has the usage set as memory and the access as readOnly. With an access type of readWrite, the memory behaves as a RAM. The memory bank also contains the mandatory bankAlignment argument. A *parallel* bank alignment specifies each item (constituent bank or address block) is located at the same base address with different bit offsets whereas the *serial* alignment specifies the first item is located at the bank's base address and each subsequent item is located at the previous item's address plus the range of that item. This allows the user to specify only a single base address for the bank and have each item line up correctly.

Some address blocks can be reserved to be locations for registers. A Register is a storage location internal to the the processing elements like processors. The mandatory size attribute gives the width of the register, counting in bits. The addressOffset describes the offset, in addressing units from the containing memoryMap element. Lastly, the reset element describes the value of a register at reset. The registers are further decomposed into fields, with which a value is associated. A field element of a register describes a smaller bit-field of a register.

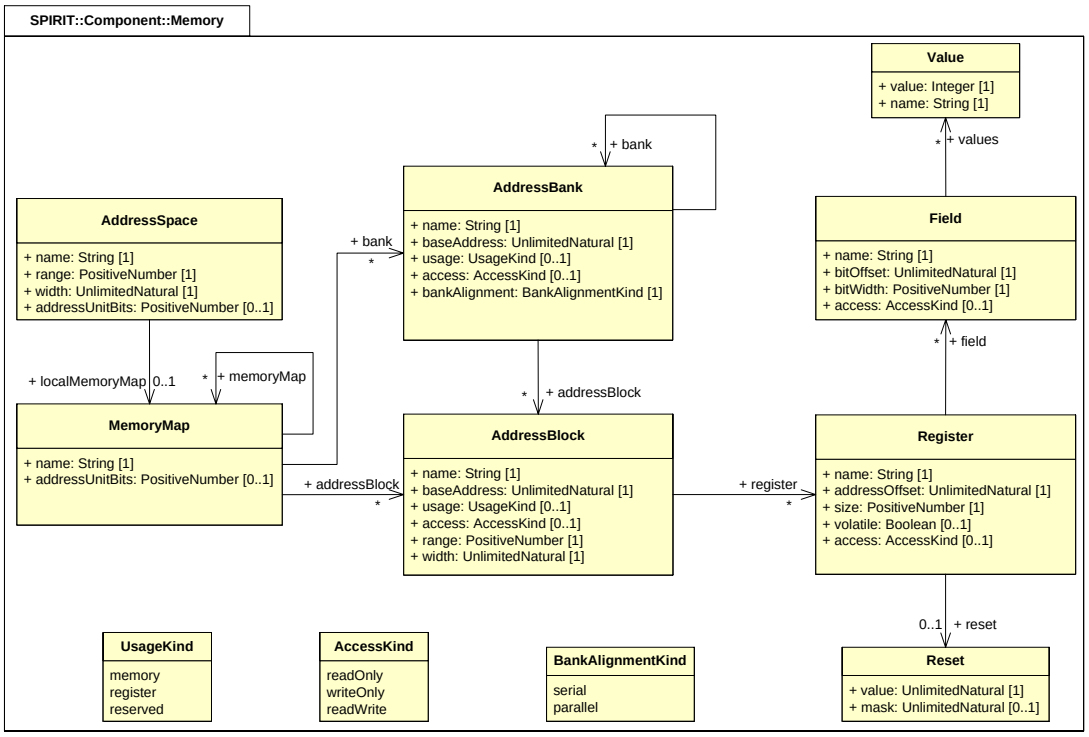


Figure 5.8: IP-XACT Component Memory Metamodel.

Contrary to registers, it contains the bitOffset element which describes the offset (from bit 0 of the register) where this bit-field starts.

Our memory metamodel reflects the actual IP-Xact specification. However, S. Revol has proposed an alternative metamodel [RTT+08], which is much more flexible and relies on the pattern Item/Descriptor. He distinguishes register definitions from register instances. The former consists in a generic definition of a register whereas the latter gives the specifics. For instance, all registers of the same size and type have the same definition. This is a valuable improvement to the IP-Xact specification, however, the purpose of our domain view is to provide a synthetic and faithful overview of IP-Xact concepts and to check the conformity of the proposed profile with this view. We do not intend here to improve this particular aspect of IP-Xact and therefore we adopt the metamodel shown in figure 5.8.

5.2.4 Other Elements

At the start of this subsection we described that the IP-Xact components are used to describe cores, peripherals, storage elements, and buses without any distinction of their type. But in actual, to a little extent, IP-Xact specification keeps provision for the processing core elements (figure 5.4). The *Cpu* model element describes a containing component with a programmable core that has some sized address space. That same address space may also be referenced by a master interface and used to create a link for the programmable core to know from which interface transaction the software departs. Its mandatory addressSpaceRef element indicates which address space maps into this cpu and gives the reference to a unique address space.

An IP-Xact component may also contain channels. A Channel model element is a special

object that can be used to describe multi-point connections between regular components, which may require some interface adaptation. Each channel element contains a list of all the mirrored bus interfaces in the containing component that belong to the same channel. Channel interfaces are always mirrored interfaces. A channel connects component master, slave, and system interfaces on the same bus. A channel can represent a simple wiring interconnection or a more complex structure, such as a bus. Channels support memory mappings but can only have one address space.

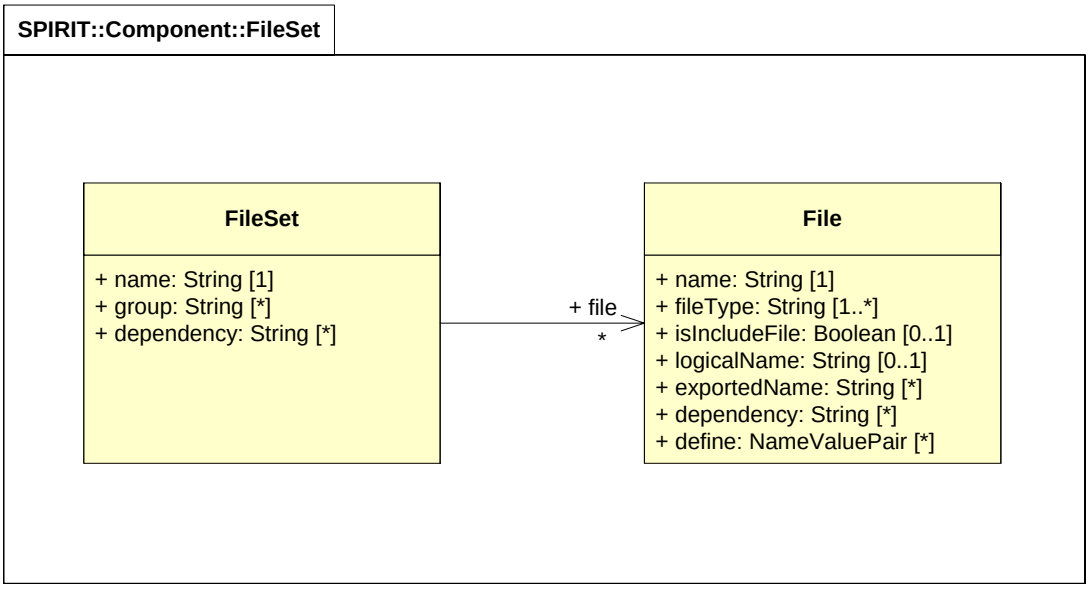


Figure 5.9: IP-XACT Component FileSet Metamodel.

5.3 Interface 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*. In IP-Xact, these two definitions are collective called as *interface definitions*. These two descriptions are referenced by components or abstractors in their bus or abstractor interfaces, also shown in figure 5.2, page 62. We go through the two definitions in the subsequent subsections.

5.3.1 Bus Definition

The busDefinition element describes the high-level attributes of the interfaces connected to a bus or interconnect (see figure 5.10). It defines the structural information associated with a bus type, independent of the abstraction level, like the maximum number of masters and slaves allowed on the bus defined by the maxMasters and the maxSlaves attributes respectively. As we expressed before, just like components the two bus definition elements being the top level elements are uniquely identified by the identifier of the type VersionedIdentifier. A mandatory directConnection Boolean variable specifies what connections are allowed. A value of true specifies these interfaces may be connected in a direct 'master to slave' fashion whereas the false indicates only non-mirror to mirror type connections are allowed (like master to mirrored

master, *etc.*). The last mandatory Boolean variable isAddressable specifies if true, that the bus has addressing information and a memory map can be traced through this interface or conversely do not contain any traceable addressing information.

IP-Xact also provides a mechanism to *extend* bus definitions. Extending an existing bus definition allows the definition of compatibility rules with legacy buses. For instance, an AHB (Advanced High-performance Bus) definition may extend the AHBLite bus definition (one of the basic version of the bus from AMBA). The extensions are constrained. An example of compatibility rule is that an extending bus definition must not declare more masters and slaves than the extended one. The extends element contains the VLNV reference of the other bus definition.

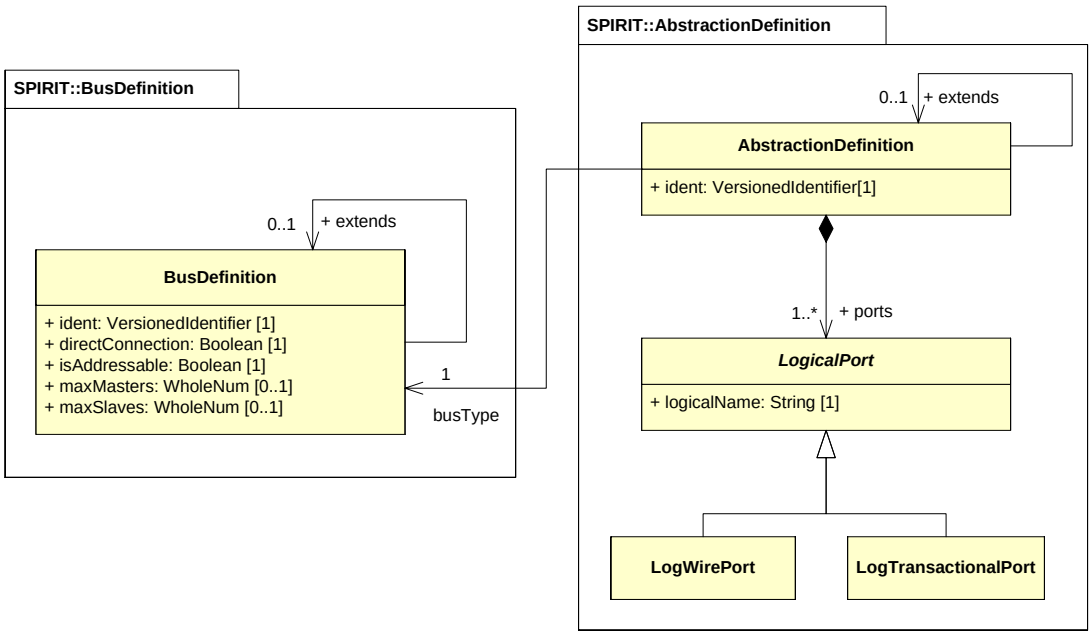


Figure 5.10: IP-XACT Interface Definitions Metamodel.

5.3.2 Abstraction Definition

The AbstractionDefinition model element describes the low-level aspects of a bus or interconnect. Note that using the word *abstraction* for low-level aspects can be somewhat misleading. However, we keep this name to follow the IP-Xact standard. An abstraction definition gives more specific attributes for a given bus definition. There can be several abstraction definitions for the same bus definition, like AHB _rtl and AHB _tlm from the Leon 2 example of IP-Xact specification [SPI08]. An abstraction definition must contain two mandatory elements, busType and ports. The busType attribute gives the reference to the bus definition for which this abstraction definition exists. Just like the bus definition, an abstraction definition can also extend another abstraction definition with some compatibility constraints to enforce. The extending abstraction definition may change the definition of logical ports, add new ports, or mark existing logical ports illegal. Here we have to keep an important point in consideration that if an abstraction definition extends another abstraction definition, then the corresponding bus definitions referred to in the two abstraction definitions must also extend each other. This

can be stated by the following OCL rule:

```
context AbstractionDefinition inv:
  self.extends.busType = self.busType.extends
```

As an illustration, we consider the example given in the IP-Xact specification [SPI08] shown in figure 5.11. Here, the abstraction definition AHB _rtl extends the AHBLite _rtl which implies that the bus definition AHB also extends the basic bus structure AHBLite.

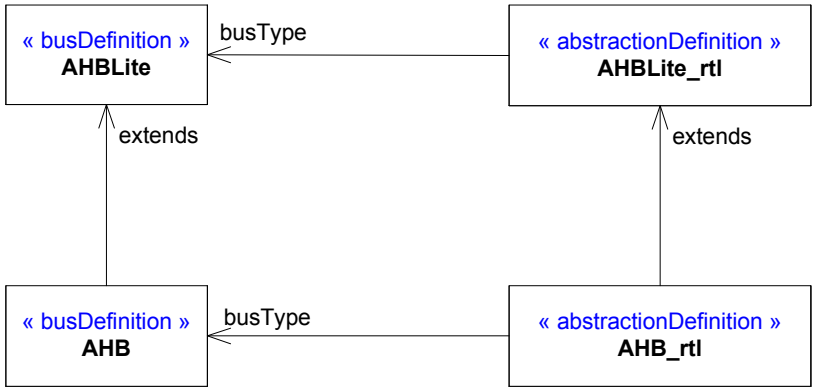


Figure 5.11: Bus Extension Constraint Example.

The extension such as used in bus and abstraction definitions shows one of the weakness of an XSD-based standard. The relationship between the extending and the extended model elements is typically a generalization relationship. The standard speaks about the "extends relation hierarchy tree", along with a concept of inter-connectability, which appears to be close to substitutability. Because IP-Xact ignores generalization, the standard imposes that "all the elements and attributes of the extended bus definition and abstraction definition pair shall be specified in the extending bus definition and abstraction definition pair". This surely leads to verbose models.

The abstraction definition specifies the ports elements and constrains them (type, direction ...). These ports are *logical ports*, contrary to the *physical ports* parts of the Model Component (figure 5.4). In fact, we can say that the abstraction definition is a collection of logical ports at given abstraction level that may appear on a bus interface for a particular bus type. Each logical port must define the logicalName attribute which gives a name to the logical port that can be used later in component descriptions when the mapping is done from a logical abstraction definition port to the component physical port, through *port maps*, already presented in the Component::BusInterface package. The logical name shall be unique within the abstraction definition. Each port also requires a wire element or a transactional element to further describe the details about this port.

A LogWirePort model element represents a port that carries logic values or an array of logic values. This logical wire port contains two sets of optional constraints for a wire port, to which it is mapped inside a component's or abstractor's bus interface. First one is the qualifier which indicates that this is an address, data, clock or reset port. The second set contains the optional constraints like presence of the port on the bus interface, its direction and width and is defined separately for master, slave and the system. It indicates the behavior of the port with respect to its containing bus interface's mode type. A LogTransactionalPort carries information that is represented on a higher level of abstraction. This logical transactional port may provide optional constraints for a transactional port, to which it is mapped inside a

component's or abstractor's bus interface. The optional constraints for the transactional ports are little different from the wire port. The qualifier element indicates that the port is address or data port. There is no provision for the clock or reset as this port is at transaction level. Also, instead of the direction and width elements of the wire port, transactional ports have service attribute describing the behavior of the port. Further details of these constraints can be consulted in the IP-Xact specification [SPI08].

Bus and abstraction definitions for the Acquisition system

In Bus Definition

direct connection Master/Slave = true
maximal number of Masters = 1
maximal number of Slaves = 1

In Abstraction Definition

Acq			
Logical port		On Master	On Slave
SAMPLE	w	out	in
TRANSFER	w	in	out

Figure 5.12: Definitions of the `Acq` bus type.

Figure 5.12 contains several information relative to bus interface Acq used in components Sensor and Processor of the Acquisition system. The Bus Definition specifies that the connection master/slave is 1 to 1, and direct. The abstraction definition introduces two wire logical ports (SAMPLE and TRANSFER), and their respective directions according to their mode (master or slave).

In Bus Definition

direct connection Master/Slave = false
maximal number of Masters = 4
maximal number of Slaves = 16

In Abstraction Definition

Save			
Logical port		On Master	On Slave
ADDR	w	out	in
DATA	w	out	in
BREQx	w	out	illegal
GRANTx	w	in	illegal
DONEx	w	in	illegal
SELx	w	illegal	in

Figure 5.13: Definitions of the `Save` bus type.

Bus interface Save is a bit more complex because, according to the mode, some logical ports are forbidden. This is shown in figure 5.13 with the keyword `illegal`. The logical ports are wire ports. The bus definition says that only non-mirrored to mirrored type connections are allowed. The maximum numbers of masters and slaves has been taken as 4 and 16 arbitrarily.

5.4 Design

An IP-Xact *design* is the central placeholder for the assembly of component objects. It represents a system or a sub-system defining the set of component instances and their interconnections, also shown in figure 5.14. The interconnections may be between interfaces or between ports on a component. Thus a design description is analogous to a schematic of components.

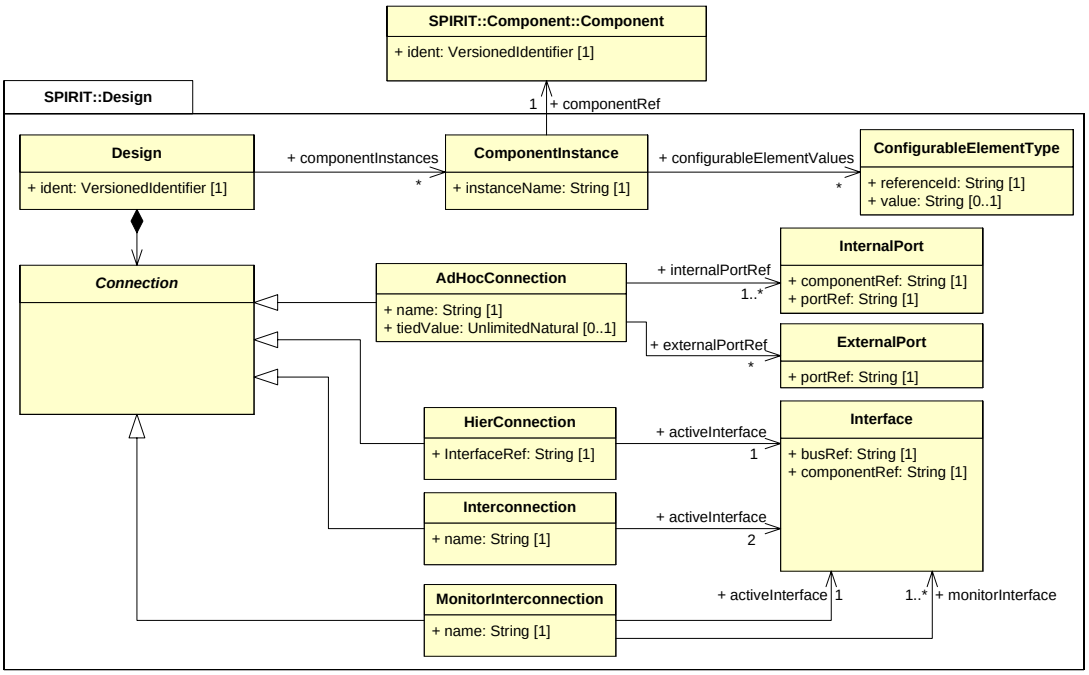


Figure 5.14: IP-XACT Design Metamodel.

The design componentInstance refers to the original component description. As this design element represents a system, these component instances can be configured to the specific design needs like providing clock specifications (period, offset) or providing addressable memory range, *etc.*. These configurable values are given using the configurableElementValues element for providing the value of a specific component parameter. The designer can then connect the components using various types of connection elements. The key types of connections are the interconnection, monitor interconnection, ad-hoc, and hierarchical connections. Interconnection is the point-to-point connection of bus interfaces from two sibling components and hence, is the standard connection type used to connect the components in the same design. It specifies exactly two bus interfaces that are part of the interconnection using the activeInterface element having the type Interface. Interconnections can only connect two bus interfaces and broadcasting of interconnections is not allowed. MonitorInterconnection is a special kind of interconnection which specifies the connection between a component bus interface and a list

of monitor interfaces (a kind of bus interface mode, discussed earlier) on other component instances. AdHocConnection connects two ports directly, wire ports but also transactional ports, without using a bus interface. It can also connect the component instance ports and ports of the encompassing component (in the case of a hierarchical component). It must contain at least one reference to the internalPort specifying the component and its desired port. On the other end of the connection, there can be another internal port or an externalPort which exists outside the design hierarchy and only specifies the target port without any reference to the containing component. Lastly, the hierarchical connections (HierConnection) connect components from different hierarchical levels (*e.g.,*a parent to one of its children). A hierarchical connection represents a hierarchical interface connection between a bus interface on the encompassing component and a bus interface on a component instance of the design. It contains a single mandatory reference to a component bus interface (activeInterface) which is then delegated to the containing component.

Acquisition system

The design of the acquisition application contains component instances and the interconnections between bus interfaces. An interconnection between bus interfaces is specified by a pair of *activeInterfaces*. An activeInterface (an IP-Xact element) consists of a component instance reference and a bus interface reference. As usual in IP-Xact, these references are made by names.

- component instances: `s1:Sensor`, `s2:Sensor`, `p1:Precessor`, `p2:Precessor`, `m1:Memory`, `m2:Memory`, `m3:Memory`, and `b:ABus`
- interconnections: for instance, Bus interface `M_A` of component instance `p1` is connected with bus interface `S_A` of component instance `s1`.

Figure 5.15 shows a partial view of the repository for a design of the Acquisition system. Rounded-corner rectangles with dashed outlines are not part of the instance model, they only group objects whose specifications are in a same location (*e.g.,*component specification, design specification, etc.).

5.5 Abstractor

System level design (SLD) is the design methodology to deal with the complexities of Systems-on-Chip (SoC) by using the higher abstraction levels. Today, the system designs are usually formulated at high abstraction levels like TLM and then are automatically or manually transformed into low level models like RTL, finally paving the way for the chip synthesis. In actual, this transformation does not occur in a holistic way and the discrete blocks of models are treated separately. Usually, a single independent IP block (like a communication bus) is transformed into the low level design whereas the rest of the system works at the same high level. This ensures the avoidance of complexity in the design transformations by dividing them into smaller parts and that each block can be verified independently. For this form of discrete step-by-step transformation, we need the *transactor* elements which are used to bridge the gap between the communication interfaces at different abstraction levels. A transactor works as a translator from TLM function calls to sequences of RTL statements, hence providing the mapping between transaction-level requests, made by TLM components, and detailed signal-level protocols on the interface of RTL IPs [BDF08]. This transactor translations are two way phenomenon and also includes the mappings of the structural design architectures. A transactor will break down the TLM level channel communication into several wires and buses,

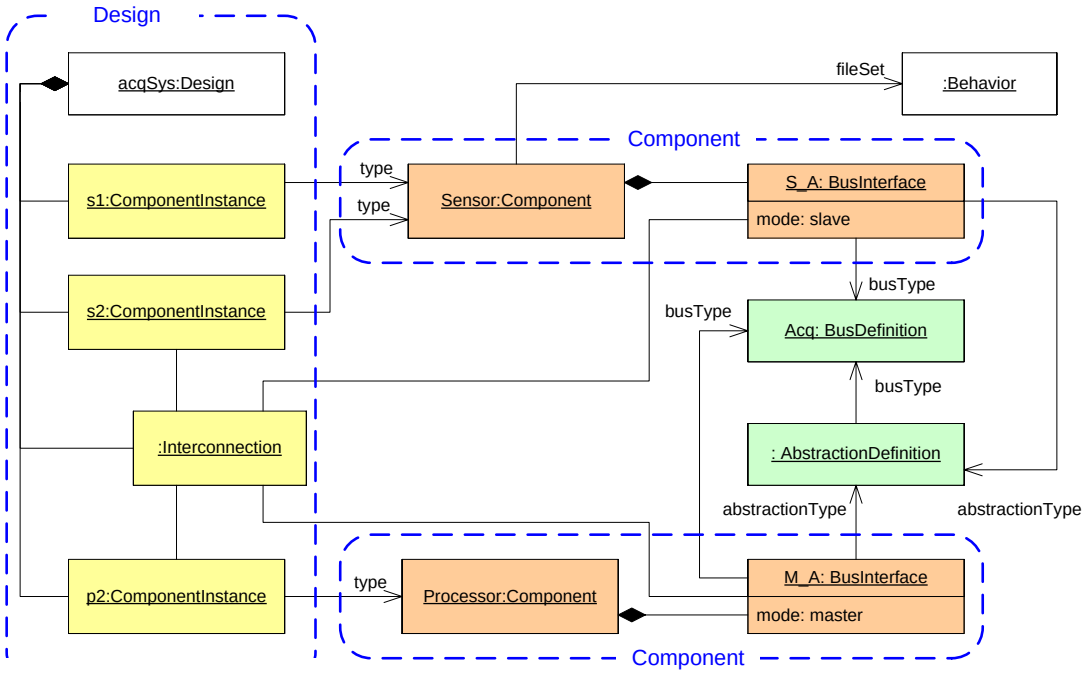


Figure 5.15: (partial) repository of the Acquisition system Design.

introducing many new signals like clocks, reset, *etc.*which did not existed at higher abstraction level (like TLM-PV, which does not contain timing information). Thus, the transactors often introduce valuable information into the design of a system and for the functional verification of such models, it is absolutely necessary to verify these transactors also.

In IP-Xact, the concept of transactors is represented by the *abstractors*. Abstractor is the top level IP-Xact element used to convert between two bus interfaces having different abstraction types and sharing the same bus type (figure 5.16). Designs that incorporate IP models using different interface modeling styles (*e.g.,*TLM and RTL modeling styles) may contain interconnections between such component interfaces using different abstractions of the same bus type. An abstractor contains two interfaces, which shall be of the same bus definition and different abstraction definitions. Unlike a component, an abstractor is not referenced from a design description, but instead is referenced from a design configuration description. Abstractor consists of the mandatory elements of abstractorMode, busType and the abstractorBusInterface. The bus type, just as in bus interface, refers to the unique bus definition associated with the abstractor.

An abstractor contains two mandatory abstractorInterface elements. Each abstractor Interface element defines properties of this specific interface in an abstractor. It contains an obligatory reference to the abstraction definition for this abstractor interface. Abstractor interfaces can also contain a list of optional portMap elements taken from the bus interfaces. Each port map element describes the mapping between the logical ports, defined in the referenced abstraction definition, to the physical ports, defined in the containing component description (see bus interface for details). The abstractorMode element is some what similar to the interfaceMode element of the bus interface. But it contains four values master, slave, direct and system representing the four possible modes of interconnection between the two mandatory abstractor bus interfaces. The *master* mode specifies that the first abstraction

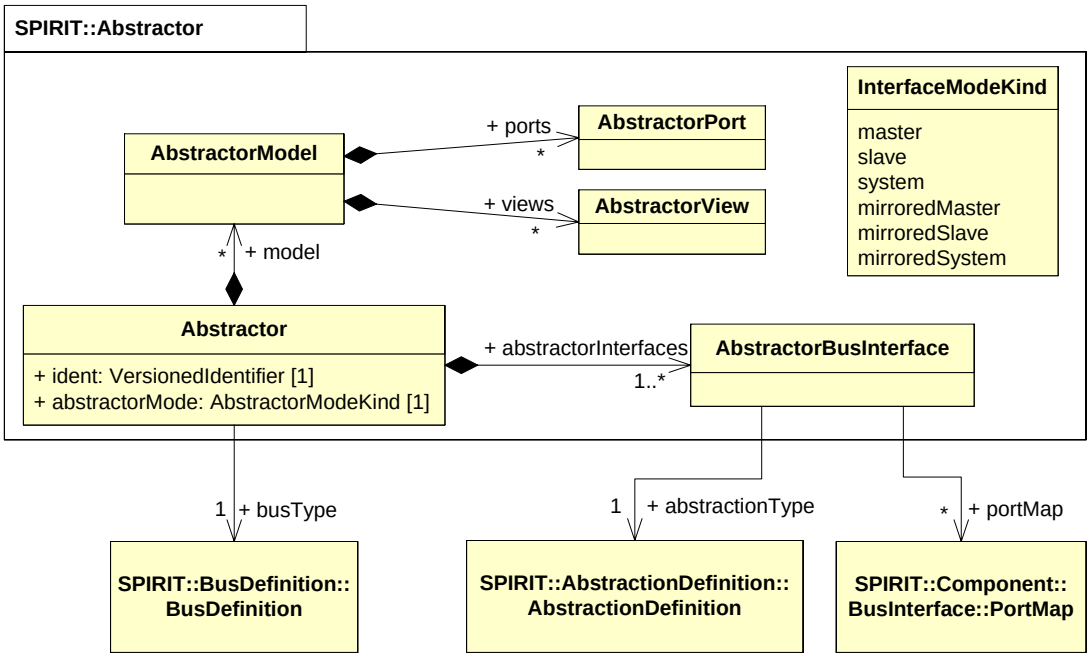


Figure 5.16: IP-Xact Abstractor Metamodel.

definition interface connects to the master, whereas the second connects to the mirroredmaster and similarly for the other modes. The abstractor model element is exactly similar to the bus interface model element other than the naming terminology used. It contains the abstractor views and the abstractor ports. We advice to refer to the bus interface subsection for further details on it.

5.6 Conclusion

At several places in this chapter we have shown that IP-Xact is syntactically rich but complex. The scattering of information has been demonstrated through the acquisition system example. Even if the IP-Xact specification can be exploited by tools, it remains hard to understand and build for a human user. This provided the idea to extract the essence of IP-Xact to a metamodel, with the intent to propose a simpler front-end to IP-Xact specifications.

We have taken advantage of opportunities offered by the metamodeling concepts to simplify the representation of many of the mechanisms provided by the IP-Xact standard. The proposed metamodel can now function as the reference metamodel for the design of IP-Xact systems even without the use of modeling languages like UML. Moreover, our domain view also gives a better understanding of IP-Xact concepts involving multiple hierarchies which is not possible to comprehend directly from the IP-Xact specification or by seeing the IP-Xact XML description files. Putting on the concepts of IP-Xact in metamodeling allows us to benefit from the enormous work done by the Spirit consortium for modeling electronic components. It positions our metamodel as a comprehensive state of the art initiative and the center point for the future work on design and modeling of IP-Xact concepts. In the following chapters, we illustrate how to use this information to model IP-Xact in UML and to automate the design flow of systems on chip by relying on techniques of MDE.

Chapter 6

Modeling IP-XACT in UML

Contents

This chapter describes the representation of IP-X*act models in* UML*. We use a combination of* UML *concepts,* MARTE*, and our own profile in our attempt to specify* IP-X*act models in* UML*. We have defined* UML *stereotypes with parsimony and have also introduced a model library to provide a set of data types equivalent to* IP-X*act primitive types. The stereotypes and the model library are gathered within a new profile named* UML *profile for* IP-X*act.*

6.1 Introduction

The analysis of the IP-Xact metamodel in the previous chapter leads us to the need to introduce a *profile* for IP-Xact facilitating the integration of approaches based on UML in the context of IP-Xact. To fulfill this requirement, we introduce our UML profile for IP-Xact. Here we bring the concepts of the metamodeling in IP-Xact to the UML syntax through the mechanism of profile. In this way, the metamodeling concepts are applied to use the graphical and tooling capabilities of UML (graphical modular designing, component reuse, different representation views, *etc.*) and the user-defined profile. This chapter describes the representation of IP-Xact models in UML. We use a combination of UML concepts, MARTE and our own profile in our attempt to describe the IP-Xact models in UML. Following B. Selic [Selic 2007], we have tried to define stereotypes with parsimony and to create new ones when no equivalent concepts were available in UML or in MARTE. In addition to stereotypes, we have also defined a *model library* to provide a set of data types equivalent to IP-Xact primitive types. The new stereotypes and the model library are gathered within a new profile named UML profile for IP-Xact. We will explain each of the concepts introduced to clarify the reasons that why we have defined them. Moreover, we justify the way in which we have transformed the UML models into IP-Xact models and compare our approach with the other approaches.

There have been several propositions to use UML in SoC Design [CSL+03, Sch05] including usage of profiling mechanisms (UML for SoC [MM05, OMG06], Omega-RT [GOO06] and UML for SystemC [RSRB05]). There are also some combined UML/SysML-based environments to support analysis and produce SystemC outputs [VSBR06]. However, our work specifically focuses on the interoperability among IP-Xact models and makes an extensive use of the MARTE profile and its time model. Some preliminary works [ZBG+08, AMMdS08, SX08] have considered solutions to model IP-Xact designs in general purpose modeling languages like UML with or without the support of MARTE profile. These approaches mostly focus on structural aspects, whereas we also consider behavior and time information of IPs. The UML profile for ESL proposed by Revol [RTT+08, Rev08] supports bidirectional transformations between UML and IP-Xact as well as the generation of SystemC code skeletons based on the register map information provided by IP-Xact. This profile focuses on TLM models and abstracts away all the RTL-related information. It was designed to provide a good integration with ST Microelectronics TLM design flow. Our work is complementary as Revol focuses on the structural aspects while we also look at the relationship with the behavior.

In this chapter, firstly we identify the key elements of our profile. Then, in section 6.3, we again go through the main IP-Xact concepts and for each of them we explain our mapping rules and justify the creation of required new stereotypes. Thereafter, we illustrate the use of this profile in Section 6.4.

6.2 Mapping IP-XACT Concepts

As stated earlier, our first step of the construction of our profile is to identify the concepts that we desire to manipulate in our UML models. More specifically, as our goal is to have an *interoperability* between IP-Xact based models and other conventional tools in the ESL community, we need to determine the concepts of this standard that we want to represent in UML. Some notions of UML are not expressible in IP-Xact, similarly transforming IP-Xact to the UML concepts is also not easy. Here we get the major advantage from our IP-Xact domain view introduced in the previous chapter. This domain view acts as a bridge between the general IP-Xact concepts described in XSD files and the UML structural modeling. Our end goal is to take advantage of the strengths of these two standards while *ensuring equivalence* (with

respect to the tools) between models expressed in both formats. There are many approaches given by various authors revolving around UML [ZBG+08, RTT+08, SXM09] which we will discuss and compare from time to time in the next two sections. Each approach differs in the way the IP-Xact concepts are perceived in mind and then mapped to UML. One can not draw a line of right or wrong for such approaches and all these attempts, including ours, lie in the gray area, where we consider the bargain of gain and loss by using one particular approach.

IP-Xact represents the structural aspects of a SoC design and hence, at the later stages helps in the better integration of the system. For this purpose, IP-Xact is quite strict in representing and identifying the components, their design constraints and their interconnections. All these issues are not focally covered in the UML and for this purpose, we try to take advantage from our custom built profile. This profile is a mix of the stereotypes introduced along with the new data types and enumerations defined, and stereotypes borrowed from MARTE. The new stereotypes are only considered to be introduced when the existing metaclasses of UML, the available MARTE stereotypes or the modeling diagrams fail to facilitate the concept representation. As an observation, the IP-Xact multiple hierarchy structure is better handled by the UML whereas the specialized IP-Xact concepts (like bus interfaces) are better handled by the new stereotypes (but surely with the help of existing UML structures). Practical examples will further illustrate these points.

Briefly, the profile (stereotypes and model library), the modeling techniques and principles of use make the *effective environment to develop* the IP-Xact models in UML. We take care for each of the concepts introduced to clarify the reasons that led us to define them. In addition, we justify the use of various techniques in UML modeling and profiling by referring to similar approaches found in the related works. Table 6.1 shows the general mapping relation with the key IP-Xact concepts highlighted. Our IP-Xact models are mainly built in UML using the class diagrams and the composite structure diagrams. Class diagrams represent the various bus definitions (the library elements of IP-Xact) whereas the composite structure diagrams represent all other concrete elements present in IP-Xact, like components and abstractors. This partitioning is coherent with the UML concepts in the way that the UML structure diagrams are used to represent the concrete structured elements and instances. This approach also helps us to represent the IP-Xact hierarchical components which are composed of other components. Hence, the IP-Xact models in UML are represented by using UML along with the MARTE profile and our IP-Xact profile. We discuss the use of two profiles separately in the subsequent subsections.

6.3 UML Profile for IP-XACT

6.3.1 UML and MARTE Profile

MARTE profile is the successor of SPT profile for UML. It is dedicated to the design and modeling of real-time embedded systems. It has got special focus on representing hardware as well as software structures and their behavioral models including time. As questioned by many authors recently about the use of MARTE profile in our research work, we wanted to clarify and emphasize few things. The base of UML modeling is the extensibility and re-usability features of designs [Sel98]. UML profiles provide a mechanism that has led to the development of various domain specific profiles. So our use of MARTE profile as the base for our work on IP-Xact profile avoided us from reinventing the wheel. It provided us with hardware components which are equally used by the IP-Xact specification. Also, MARTE profile has got quite acceptance in the MDE community and is supported by the UML modeling tools like Papyrus. Moreover, the MARTE time model is a comprehensive specification dealing with time related properties of a

Spirit IP-Xact	UML	MARTE	Profile for IP-Xact
VLNV			«versionedIdentifier»
Component	Structured Class	«hwResource»	
Hierarchical Component	Structured Class	«hwResource»	
Processor	Structured Class	«hwProcessor»	
Bus Interface	Port		«busInterface»
Bus Interface Mode	Enumeration		InterfaceModeKind
Address Space	Structured Class		«addressSpace»
Memory Map	Structured Class		«memoryMap»
Address Bank	Structured Class		«addressBank»
Address Block	Structured Class		«addressBlock»
Register	Structured Class		«register»
Field	Structured Class		«field»
Memory Usage Type	Enumeration		UsageKind
Memory Access Type	Enumeration		AccessKind
Memory Bank Alignment	Enumeration		BankAlignmentKind
View	Property		«view»
Physical Port	Port		«wirePort», «transactionalPort»
Wire Port Direction	Enumeration		DirectionKind
Transactional Port Initiative	Enumeration		InitiativeKind
Cpu	Property		«cpu»
Communication Bus	Structured Class	«hwBus»	
Channel	Property		«channel»
RAM	Structured Class	«hwRAM»	
Bridge	Structured Class	«hwBridge»	
Timer	Structured Class	«hwTimer»	
Bus Definition	Class		«busDefinition»
Abstraction Definition	Class		«abstractionDefinition»
Logical Port	Property		«logWirePort», «logTransactionalPort»
Port Presence Type	Enumeration		PresenceKind
Abstractor	Structured Class		«abstractor»
Abstractor Interface	Port		«abstractorBusInterface»
Abstractor Interface Mode	Enumeration		AbstractorModeKind
Design	Structured Class		
Component Instance	Part (Property)		
Instance configuration	Property		«configurableElementValue»
Connection	Connector		

Table 6.1: Mapping IP-Xact concepts to UML(Structured Class is an abbreviation to denote the metaclass Class from the StructuredClasses package)

system. In the last part of this thesis, we have used the MARTE time model concepts along with its clock representation language CCSL. In the future, we try to merge our work to represent IP behavior, using CCSL and time model, with the IP-Xact specification. Currently IP-Xact specification is mostly dealing with structural aspects of ESL designs and such a proposal for behavior integration will be a great contribution. Thus, use of MARTE profile as the base of our sub-profile for IP-Xact is helpful not only from the structural view-point but is also of a potential use for behavioral representation. Last but not least, our AOSTE team at INRIA Sophia Antipolis has got major contributions in the development of MARTE profile, so its

visionary to find its possible use next to IP-Xact specification.

Our sub-profile for IP-Xact borrows several concepts and stereotypes from MARTE. These stereotypes are used to represent the IP-Xact components in UML. As discussed in the preceding chapter, the Hardware Resource Modeling (HRM) package from the MARTE profile contains a large collection of hardware element stereotypes. They can be used to represent the vast variety of components represented in IP-Xact. IP-Xact does not explicitly differentiate between hardware elements (like the processor and the RAM both represented by the IP-Xact component). But at a closer look, the internal structure of the various components makes them distinct. For example, the IP-Xact channels are the collection of mirrored bus interfaces (either they are master, slave or system) which normally exist on the bus components only. Similarly, the IP-Xact cpu element can exist on the processing element. Thus using the MARTE profile, we highlight the difference of these components by applying distinct stereotypes. We apply MARTE stereotypes from the HRM package to identify components that must be transformed into IP-Xact components. More specifically, we apply the stereotype HwResource and some of its sub-stereotypes. Components stereotyped by «hwProcessor», «hwMemory» and «hwBus» are all transformed into IP-Xact components but give us the opportunity to organize and represent the information in a better way in UML. This method gives us the advantage to benefit the most from the graphical features of UML. As an example, the different types of memories in IP-Xact are differentiated by the access type (readOnly, writeOnly, readWrite) and usage type (memory, register, reserved) in their properties. But in UML, using the MARTE HwRAM or HwROM stereotypes, we can visually show different types of memories in our UML models even though they are represented by the same IP-Xact component type. Similarly, the HwBridge and HwTimer stereotypes are the respective equivalent of IP-Xact *bridge* and *timer* components. All these mappings between the IP-Xact concepts and the MARTE profile is shown in table 6.1. Illustrations are given in section 6.4 of this chapter. For instance, figure 6.10 on page 94 shows our sample IP-Xact processor component with the HwProcessor stereotype applied from the MARTE profile. Note that we have followed the naming convention imposed by the authors of the Leon2 architecture example given by the Spirit consortium, even though it is generally admitted that class names should start with a capital letter. This was introduced to better present IP-Xact files, similar to the ones given by the authors, after the automatic model transformation. IP-Xact components get more complex when we consider the industrial applications. All such components of a type not even represented in MARTE can be shown by the generic stereotype HwResource (see figure 6.17, page 101 for a hierarchical component).

6.3.2 UML and IP-XACT Profile

Other than the MARTE profile, we use a combination of UML features and our specialized IP-Xact profile for the representation of various IP-Xact model elements. Here we discuss one by one the model elements of the profile for IP-Xact.

Component

Components are represented in the composite structure diagrams by the classes from the StructuredClasses package in UML metamodel. The UML Class of the StructuredClasses package is different from the Class of the kernel package in the way that the former is an EncapsulatedClassifier and a StructuredClassifier (refer back to figure 3.2 on page 26) and so, may have an internal structure and ports. In what follows, we use the terms "Structured Class" to denote class with internal structure and ports, even if it is not the name of a UML metaclass. An IP-Xact component inherently can contain component attributes and ports. This way a Structured Class can perfectly represent any hardware element containing parts, ports, attributes

or interconnections. We refer to figure 5.4 on page 64 from the preceding chapter to show the relationship of IP-Xact components with other elements. The IP-Xact VLNV information is provided by the VersionedIdentifier stereotype which contains the two attributes vendor and version of the IP-Xact component. The component's qualified package name contributes to the VLNV information as library name, along with the component name itself for the rest of the two fields. The component name of the structured class, contributing indirectly to the VLNV, is mandatory as the structured classes are UML NamedElement.

Note that stereotype VersionedIdentifier is *abstract* and extends no UML metaclass. This solution, already adopted for the TimedElement stereotype from MARTE, has been preferred to another solution effectively taking a metaclass (*i.e.,*PackageableElement [SXM09]). The VersionedIdentifier stereotype is used as a generalization for several other (concrete) stereotypes, which all need the information carried by the versioned identifier. Figure 6.1 shows VersionedIdentifier stereotype, useful data types, and some of the IP-Xact component sub-elements implemented as stereotypes extending the UML metaclass Property.

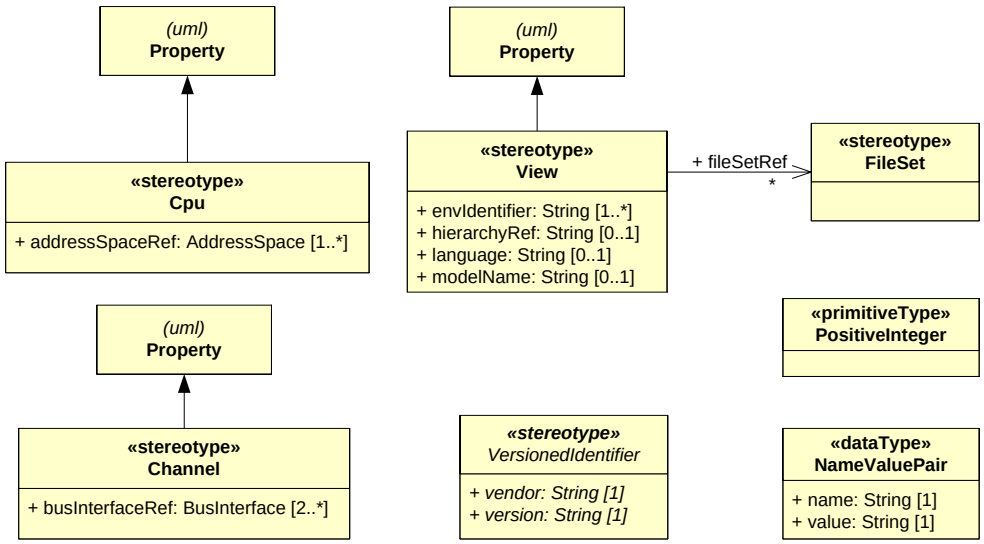


Figure 6.1: VLNV and IP-Xact Component element stereotypes

Just like in our metamodel, we describe the component from the most basic part *i.e.,*its model. In IP-Xact, model is a collection of component views and physical ports. We implement IP-Xact physical ports and views as distinct entities in our UML models for IP-Xact. A View is defined as a stereotype that extends the metaclass Property, as shown in figure 6.1. As these IP-Xact elements and their attributes were thoroughly discussed in the metamodel, so here we only provide their mapping onto UML elements. For any confusion regarding the IP-Xact terms and concepts, one can refer to the preceding chapters. The View stereotype contains the hierarchyRef attribute which contains the name of the design file of the component (discussed in subsection 6.4.3). In the similar way, we represent the Cpu and the Channel stereotypes as properties of the structured classifier (hence, IP-Xact component). The Cpu stereotype contains an optional reference to the component's memory map and specifies all the elements stereotyped with «addressSpace». Similarly, the Channel stereotype refers to the bus interfaces of the component (ports stereotyped as «busInterface»).

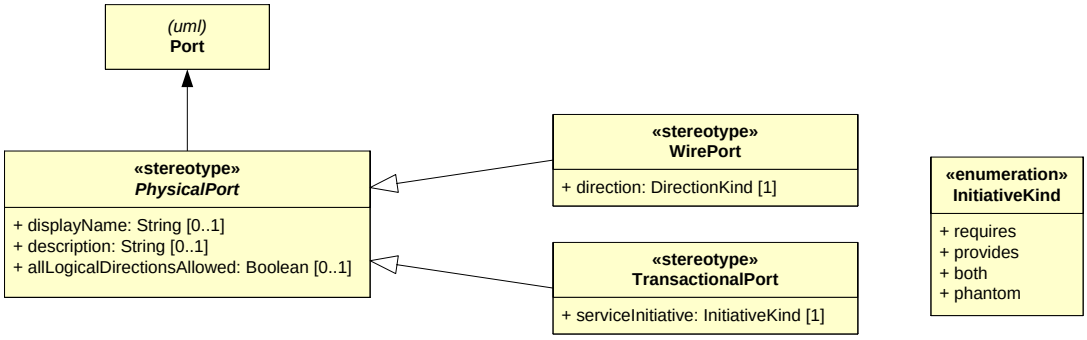


Figure 6.2: Physical Port Stereotypes.

Physical ports

The IP-Xact physical ports are represented by the abstract stereotype PhysicalPort which extends the UML Port metaclass, also shown in figure 6.2. We have chosen to represent the IP-Xact ports as UML port because they can also be used communicate with the environment just like bus interfaces. These ports can then be connected with external ports using ad-hoc connections. In the profile we have declared the PhysicalPort as an abstract stereotype, so effectively we have to use concrete sub-stereotypes: either WirePort or TransactionalPort. The initiative type of transactional ports and the direction of the wire ports are implemented as enumeration in the profile.

BusInterfaces

IP-Xact bus interfaces, as defined previously, are the *collection of similar ports* adhering to some common protocol. They are the main element to communicate with other components. As they are similar to the physical ports, we again use the UML ports to represent them. The applied stereotype BusInterface (figure 6.3) differentiates these ports from the IP-Xact physical ports. This BusInterface stereotype extends the UML Port metaclass. Some of the approaches [ZBG+08] have used UML interfaces to represent IP-Xact bus interfaces but we prefer to use UML ports to represent IP-Xact bus interfaces, as they are merely the collections of similar physical ports.

The interfaceMode for the bus interface is defined in the profile as UML enumeration InterfaceModeKind having six mode values. A master interface mode (also known as an initiator) is the one that initiates transactions whereas a slave interface mode (also known as a target) responds to transactions. A system interface mode is used for some classes of interfaces that are standard on different bus types, but do not fit into the master or slave category, like clock or reset bus interfaces. Interface mode has always been a source of confusion for the researchers trying to model IP-Xact in UML. Various authors [ZBG+08, RTT+08, SX08, SXM09] tried to introduce a separate class or stereotype for each of these interface modes. Whereas some other approaches [ASHH09] and our initial attempt [AMMdS08] for modeling IP-Xact introduced the enumerations to deal with interface mode of bus interfaces. We kept focus on design simplicity and the general rules of parsimony while introducing new stereotypes [Sel07]. One of the thing that compelled authors for the former approach was the set of attributes associated with each mode type of bus interfaces, sometimes mandatory while optional usually. For example,

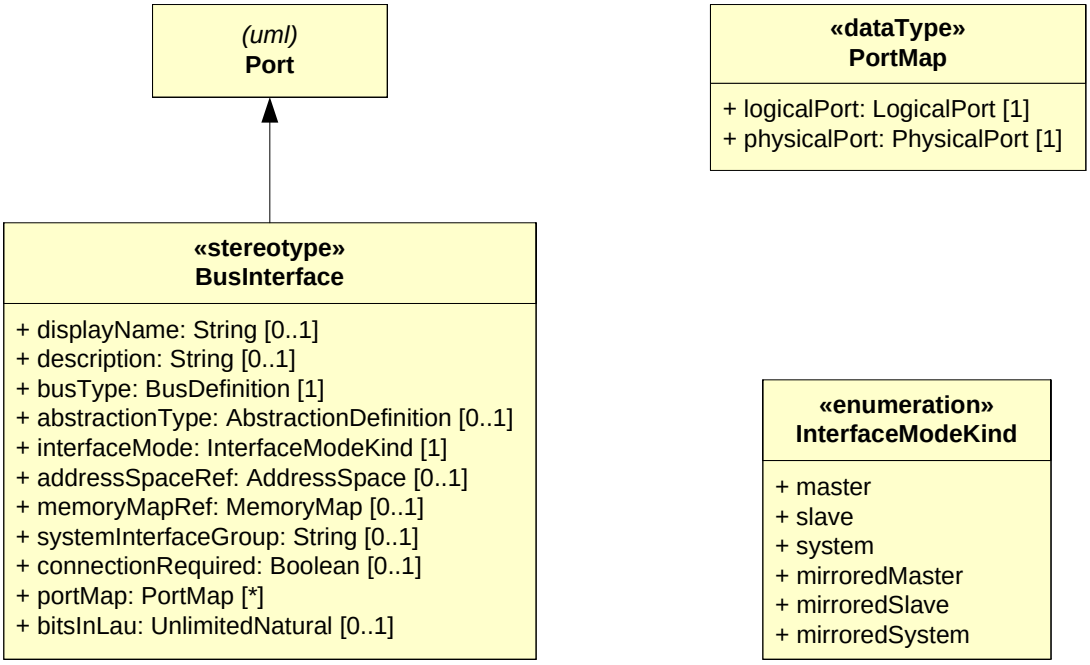


Figure 6.3: Bus interface stereotype.

the master and slave bus interfaces contain a mandatory attribute referring to their address spaces and memory maps respectively. The system mode bus interface contains a mandatory attribute 'group' specifying which names the group of the interface to which it belongs to like clock ports, reset ports *etc.*. We have combined all these attributes in our busInterface stereotype and have added OCL constraints to channelize their use. For the above group attribute, we have added the systemInterfaceGroup attribute and it must match a group name present on one or more ports in the corresponding abstraction definition. In this way, the 'system' bus interface is associated with the underlying logical ports. The system interface group name must not be available to the designer if the interface mode is not system. In UML, we apply the following OCL constraints to the BusInterface stereotype:

```
context BusInterface
inv ifMode:
  (self.interfaceMode = InterfaceModeKind::master)
      implies (self.addressSpaceRef->size()= 1) and
  (self.interfaceMode = InterfaceModeKind::slave)
      implies (self.memoryMapRef->size()= 1) and
  (self.interfaceMode = InterfaceModeKind::system)
      implies (self.systemInterfaceGroup->size()= 1)
```

Similarly, the optional presence of the bus interface model library also contains the PortMap data type that is used to represent the IP-Xact port maps. Just like the models, port map is also a collection with references to the logical and physical ports defined elsewhere. But unlike models, we prefer to introduce data type for the port maps because they are used to specify the mapping between the logical and the physical ports and neglecting this container will also loose this mapping information.

Memories

Summarizing the description given before about the memories, a memory (*i.e.,*address space and memory map) of an IP-Xact component is a collection of register locations of an addressable area of the component as seen from its bus interfaces. Address spaces are reserved for the master bus interfaces and can contain local memory maps. Memories (of any type) are made up of address banks and address blocks as their building blocks. Moreover, memories can be of the type 'registers' composed of smaller units called fields. All of the seven described memory types are implemented as stereotypes extending the UML Class metaclass shown in figure 6.4. In the memory profile library we also introduce two data types, one for the register reset value and the other for the field values. All the options regarding memory unit's access, usage and alignment are implemented as UML enumerations. IP-Xact memories are hierarchical and are dealt in UML using the class hierarchies as shown in the component model figure 6.11 on page 95.

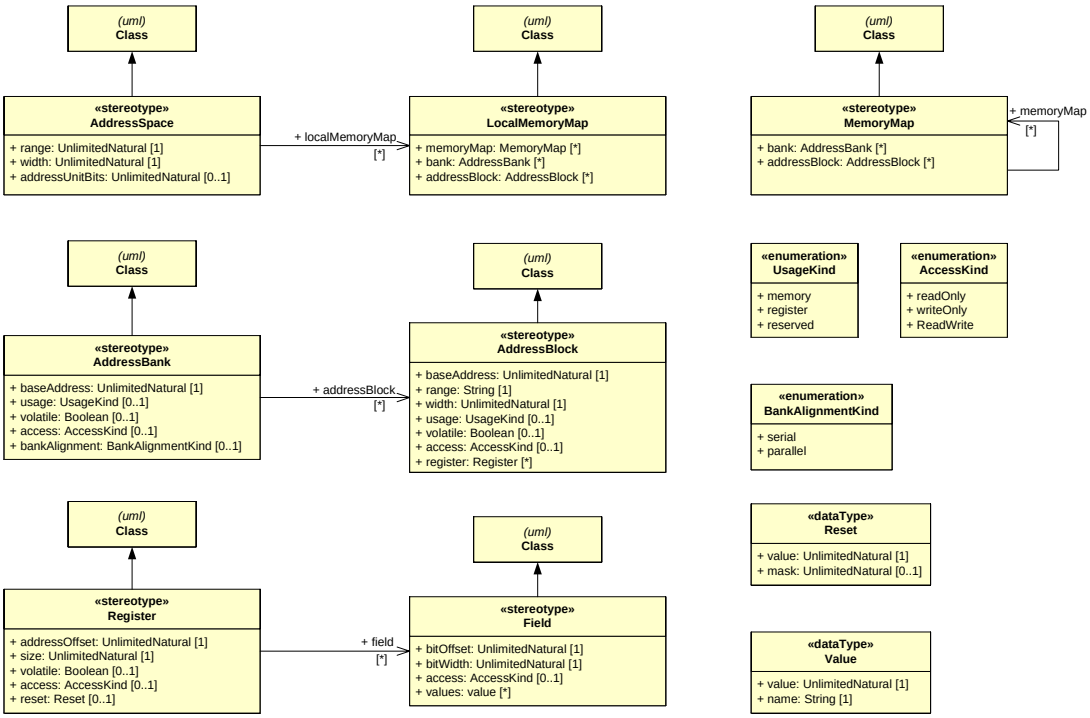


Figure 6.4: Component's Memory Model: Address Spaces and Memory Maps

Interface Definitions

To represent the abstraction definitions and the bus definitions (also called collectively as interface definitions in the standard), we use stereotypes extending the Class metaclass and specializing the VersionedIdentifier abstract stereotype. We do not use the structured classes to represent these interface definitions as these definitions do not contribute directly (by providing any concrete element) to the system design environment, and hence just work as the libraries to be referred to for concerned attributes and constraints. This approach is also used by the

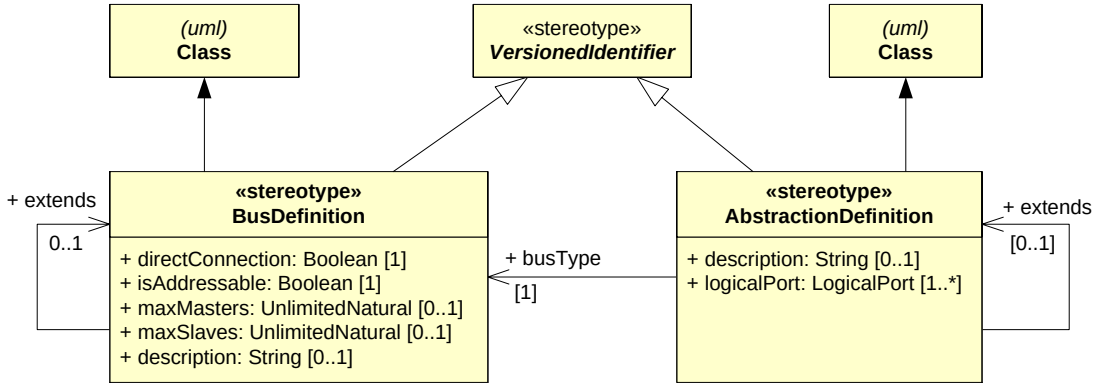


Figure 6.5: Bus definition and Abstraction definition stereotypes.

professional tools dealing with IP-Xact like *coreAssembler* from the Synopsys coreTools[1]. This coreAssembler tool also requires the buses definitions as a library but does not use them as an object in the assembly project. This all seems quite natural also because the two definitions are merely the set of constraints and the configuration attributes of their respective buses. The two definition stereotypes are shown in figure 6.5. Both the bus and the abstraction definitions specialize the abstract stereotype VersionedIdentifier, thus introducing the identifier attributes to these stereotypes. For both interface definitions, the extends attribute is used to refer to the other definition files (but of the same type) used as the base for this definition description (shown also in figure 6.20 and 6.21). Additionally, for the abstraction definition we specify the mandatory busType attribute to mention the underlying bus definition.

Abstraction definition also refers to a collection of *logical ports*, as shown in figure 6.6. The logical ports are represented by the Property metaclass extended by the LogicalPort stereotype. These ports represent the set of constraints applied to the underlying physical ports. LogicalPort is an abstract stereotype and hence the two stereotypes that can be applied to the model are the LogWirePort and the LogTransactionalPort. Both the port stereotypes contain set of attributes, implemented as data type in the profile, called Qualifier which tells us the type of information that this port carries. The Qualifier for logical transactional port consists of two Boolean attributes isAddress and isData showing whether the port is an address port or a data port. The wire logical port also contains the additional Boolean attributes of isClock and isReset showing if these ports are dedicated for the clock and reset control signals respectively.

Logical ports can have distinct set of constraints for the respective physical port depending on its presence on a master, a slave or a system bus interface. This way, we define the behavior of the port based on its environment. For example, the presence attribute, specifying the existence of the port, can have three values one each for master, slave and system and hence depicts that if the physical port belongs to a master bus interface then this logical port is present or not. The serviceInitiative and the presence optional attributes are implemented as UML enumerations. The service initiative attribute tells that the port implements the function itself (provides) or it requires the target component to implement it (requires). This initiative concept is somewhat similar to the SystemC concepts of sc_port and sc_export. The group attribute is used to group system ports into different groups within a common bus. These groupings are then obligatorily referred from the system mode bus interfaces, if they exist.

[1]https://www.synopsys.com/dw/doc.php/ds/o/coretools_ds.pdf

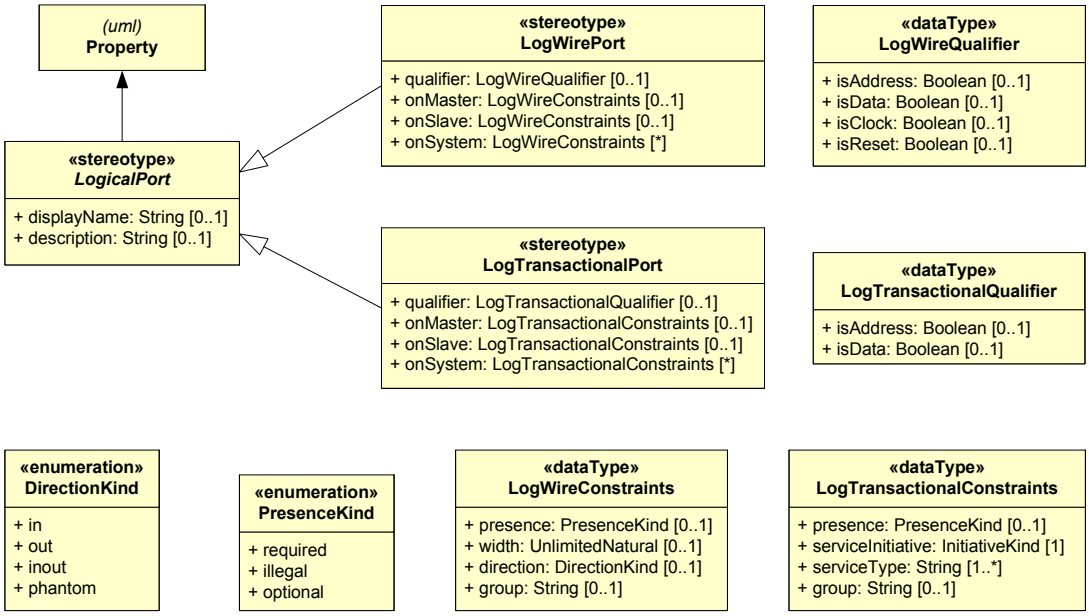


Figure 6.6: Logical Port Stereotype.

So the group attribute is restricted (using OCL constraints) as mandatory when the onSystem constraints are chosen whereas it is not allowed for other constraint types, also enforced by OCL constraints, as shown below:

```
context LogicalPort
inv logPortBehavior:
  (self.onMaster.group->size()= 0) and
  (self.onSlave.group->size()= 0) and
  (self.onSystem.group->size()= 1)
```

Abstractor

An IP-Xact abstractor component acts as a bridge between the two components at different levels of abstraction. The Abstractor stereotype extends the UML::StructuredClasses::Class metaclass and specializes the abstract VersionedIdentifier stereotype (as shown in the figure 6.7). An Abstractor contains a mandatory link to a bus definition. It also references exactly two AbstractorBusInterface, each referring to an abstraction definition element. Thus in total, an abstractor must exactly refer to a bus definition and two abstraction definition elements. The AbstractorBusInterface stereotype extends the UML Port metaclass and hence can be applied to the ports of the abstractor structured class. The attribute abstractorMode defines the mode for the interfaces on the abstractor. For master mode, one interface connects to the master while the other connects to the mirroredMaster bus interface. For slave mode, one interface connects to the mirroredSlave while the other connects to the slave bus interface. For direct mode, one interface connects to the master while the other connects directly to the slave bus interface (hence direct connection, bypassing mirrored interfaces). For system mode, one interface connects to the system while the other connects to the mirroredSystem bus interface. For system

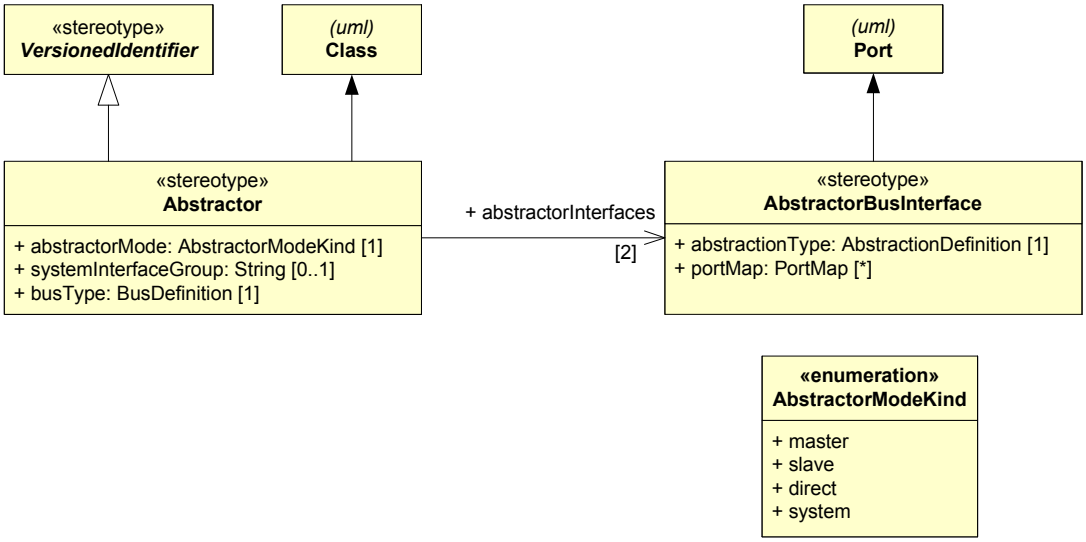


Figure 6.7: Abstractor stereotype.

abstractor interface mode, the systemInterfaceGroup attribute is obligatory to be specified and is ensured by the following OCL constraint:

```
context Abstractor
inv abstMode:
  (self.abstractorMode = AbstractorModeKind::master)
      implies (self.systemInterfaceGroup->size()= 0) and
  (self.abstractorMode = AbstractorModeKind::slave)
      implies (self.systemInterfaceGroup->size()= 0) and
  (self.abstractorMode = AbstractorModeKind::direct)
      implies (self.systemInterfaceGroup->size()= 0) and
  (self.abstractorMode = AbstractorModeKind::system)
      implies (self.systemInterfaceGroup->size()= 1)
```

Hierarchical Components

The component instances in the IP-Xact top level design or the hierarchical components are stereotyped by ComponentInstance. This stereotype, as shown in the figure 6.8, contains two attributes: designViewRef and configurableElementValues. The designViewRef refers to the view element of the component that refers to this component instance. This is needed to bound a component instance with a design view, as a component can have several views and each referring to its own internal design structure. This additional attribute did not existed in our metamodel for IP-Xact and its introduction is justified in the section 6.4.3. The other attribute is a collection of configurable element values whose type is ConfigurableElementType. This data type allows the designer to give, if needed, custom parameters with the first argument giving its identifier and the second with its value.

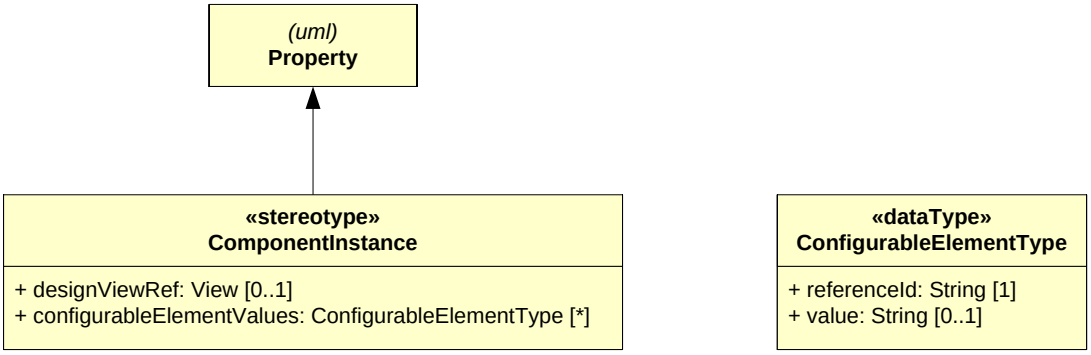


Figure 6.8: Hierarchical Design Model Library.

6.4 IP-XACT Models in UML

After introducing our IP-Xact profile for UML, we now apply it to use in modeling IP-Xact components. While modeling IP-Xact in UML, we use the profiling feature to introduce new concepts as well as use the existing UML modeling facilities like elements from class diagrams and composite structure diagrams. Thus describing the IP-Xact profile for UML is the half job done and rest of the task is taken into account in the UML models. In what follows, we use *class diagrams* to represent interface definitions (bus and abstraction definitions) and all other IP-Xact components and abstractors are represented using the *composite structure diagrams.*

In this section, we discuss the modeling of different IP-Xact elements. Firstly, in the next subsection we give an overview of the Leon II example also used by the Spirit consortium in the IP-Xact specification. Later, we explain the modeling of IP-Xact components in the following subsections.

6.4.1 Leon II Architecture based Example

The Leon Architecture is a complete open source model available to the general public. It was initially developed by Jiri Gaisler while working for the European Scape Agency (ESA). Currently Gaisler Research (or Aeroflex Gaisler)[2] is maintaining (and enhancing) the model. The core component of the Leon architecture consists of Leon II processor which is a synthesizable VHDL model of a 32-bit processor with SPARC V8 instruction set. Gaisler research recommends to use the new Leon III processor core enhanced applications but still the full source code of Leon II is freely available for the development of SoC devices. Communication base of the architecture consists of AMBA buses AHB and APB. This allows to easily add any new AMBA compliant devices to the system.

Main components of the Leon architecture consist of processor core (with cache, floating-point and co-processor), DMA, memory module, timers, UART, interrupt controller, AHB and APB AMBA buses, and the AHB to APB bridge. It also contains separate modules for clock and reset controllers. Our research work mainly focuses on the processor, memory, timers, interrupt controllers, AMBA buses, and the bridge. These components fulfill our minimum requirements to create a simple read/write system starting from the processor and ending at the timers or interrupt controller. This system is then used as a platform to test and verify our

[2]http://www.gaisler.com

research work for both structural and functional aspects. Spirit is also using Leon architecture to present an implementation of their IP-Xact standard. Presently as IP-Xact's Leon based representation is the only comprehensive example freely available, it also motivated us for the use of Leon architecture. Moreover, Leon architecture implementation is highly configurable and is well-documented, supported by online help (in terms of forums) available for the issues encountered.

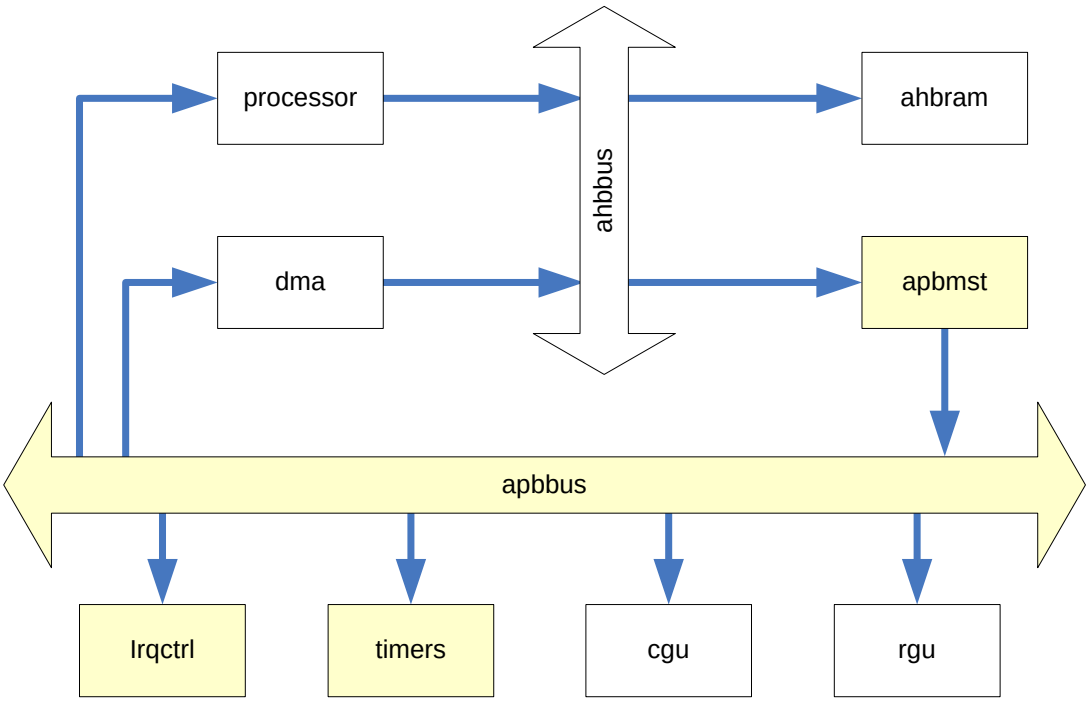


Figure 6.9: Leon 2 based system architecture.

Figure 6.9 shows the Leon architecture as used by the IP-Xact specification. At the base of the architecture's communication are the two buses from the ARM's open standard *Advanced Microprocessor Bus Architecture* (AMBA) [ARM99]. AMBA specification defines the on-chip communication standards for designing efficient embedded devices. AMBA specification defines three distinct buses: *Advanced High-performance Bus* (AHB), *Advanced System Bus* (ASB), and *Advanced Peripheral Bus* (APB). ASB is obsolete now and is mostly replace by the AHB. The recent AMBA 3 specification introduces new types of buses: AHBLite, ATB, and AXI. The last two are not considered in the Leon II example, so we will not present them.

AHB is a high bandwidth bus intended to address the high-performance system designs. It supports multiple bus masters with arbitration features. The two bus masters for our design case are the processor and the DMA. Memory module `ahbram` and the APB bridge `apbmst` are the two slave units on the AHB. An AHB *Master* is the unit that is able to initiate read and write operations by providing address, data, and control information. At any particular moment only a single bus master can command the AHB bus and is decided by the AHB arbiter based on some predefined protocol. An AHB *Slave* is a sort of passive component which can not initiate any transaction on its own. It responds to the read or write operations directed

towards its address space range.

APB is a low bandwidth bus intended for minimal power consumption and reduced interface complexity. This bus contains only one bus master, the APB bridge (apbmst). As APB bridge is also a slave on the AHB, its primary function is to convert system bus transactions into APB transactions. On an AHB transaction request, it latches and decodes the address, selects the destined peripheral, and drives the data onto the APB bus. For a read transaction, it transfers APB data onto the system bus.

In our sample architecture we have got four APB slave modules: interrupt controller, timers, clock and reset generation units. APB slaves have simple and flexible module interfaces. In the figure 6.9, the yellowish module units are combined together as a sub-component called `APBSubSystem` in the IP-Xact specification example. This demarcation allows to organize high-bandwidth and low-bandwidth components separately. Moreover, it also allows to depict the hierarchical approach used by the IP-Xact specification.

6.4.2 Component

A typical IP-Xact component consists of bus interfaces, memory maps, ports, views, and the file set sub-units. If the IP-Xact component is of type processor, then it can have an `Cpu` module, mentioning the name of the processor. If the component is a bus, then we can have a `Channel` sub-module.

IP-Xact components are represented by the structured classifiers in the composite structure diagrams. These structured classes allow us to represent both the ports and the properties of the components. Moreover, the structured classes gave us the advantage to efficiently represent the IP-Xact hierarchical component (which we discuss in section 6.4.3). In the UML component representation, above the component name is the qualified package name of which this component is a part. Here it is worth mentioning that the component name and the names of all other elements that are identified in IP-Xact by the VLNV (VersionedIdentifier stereotype) are mandatory to be provided. Taking advantage of modeling in UML, we have divided the view of IP-Xact components over several diagrams, like ports shown in one diagram and the memory maps in another. These all diagrams show the different sub-modules of the same UML component like processor. We show all these diagrams one by one along with their descriptions.

The IP-Xact physical ports are represented by the UML ports of structured classes, ports that are stereotyped as «physicalPort», as shown in figure 6.10-A. These ports can then be connected with external ports using ad-hoc connections. For commodity, the service initiative type of transactional ports and the direction of the wire ports (implemented as enumeration in UML IP-Xact profile) are shown graphically on the component ports by icons (see table 6.2).

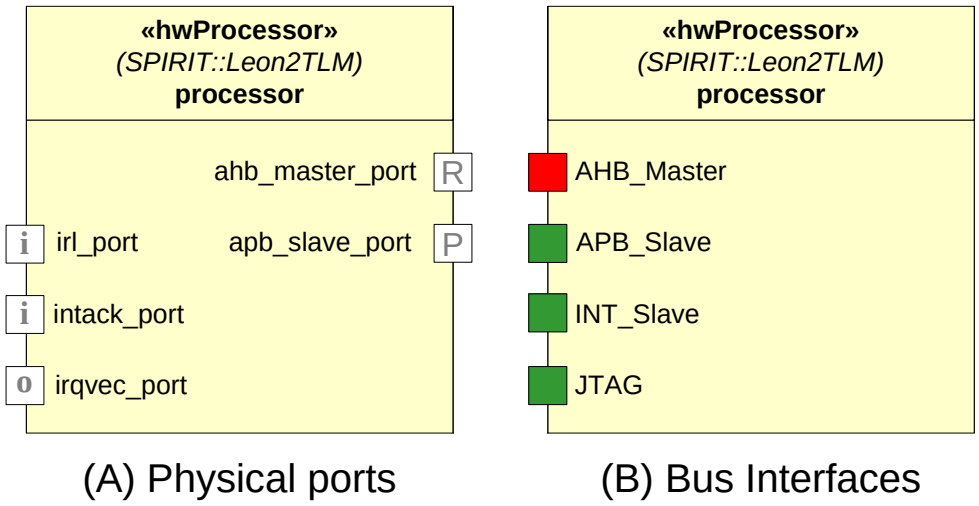


Figure 6.10: Processor Component Definition in UML: Ports and Bus Interfaces.

«physicalTransactionalPort»		
serviceInitiative	Graphical Notation	Description
requires	R	Requires port implementation
provides	P	Provides port implementation
both	B	Can require or provide port implementation
phantom	X	An abstract port
«physicalWirePort»		
direction	Graphical Notation	Description
in	i	Input port
out	o	Output port
inout	b	Bi-directional port
phantom	–	An abstract port

Table 6.2: Iconic representation of Ports.

Similar to the ports, the IP-Xact bus interfaces are also represented as UML ports on the components stereotyped with the «busInterface». The interface mode values of the bus interface are reflected in the model with different colored icons in the UML ports (as shown in the table 6.3), as proposed in the IP-Xact TLM examples application note [SPI08] (shown in figure 6.10-B). Thus finally we get the component models with the colored UML ports representing the IP-Xact bus interfaces whereas other non colored ports representing the IP-Xact physical ports. All these graphical representations for the bus interfaces and ports are shown in Tables 6.2 and 6.3. Both the bus interfaces and physical ports contain most of the information in the applied stereotypes. This information is not shown on the model diagram purposely and can be displayed if needed.

«busInterface»		
interfaceMode	Graphical Notation	Description
master		Interface in master mode
slave		Interface in slave mode
system		Interface in system mode
mirroredMaster		Mirror of master Interface
mirroredSlave		Mirror of slave Interface
mirroredSystem		Mirror of system Interface

Table 6.3: Color notation for Bus Interface Modes.

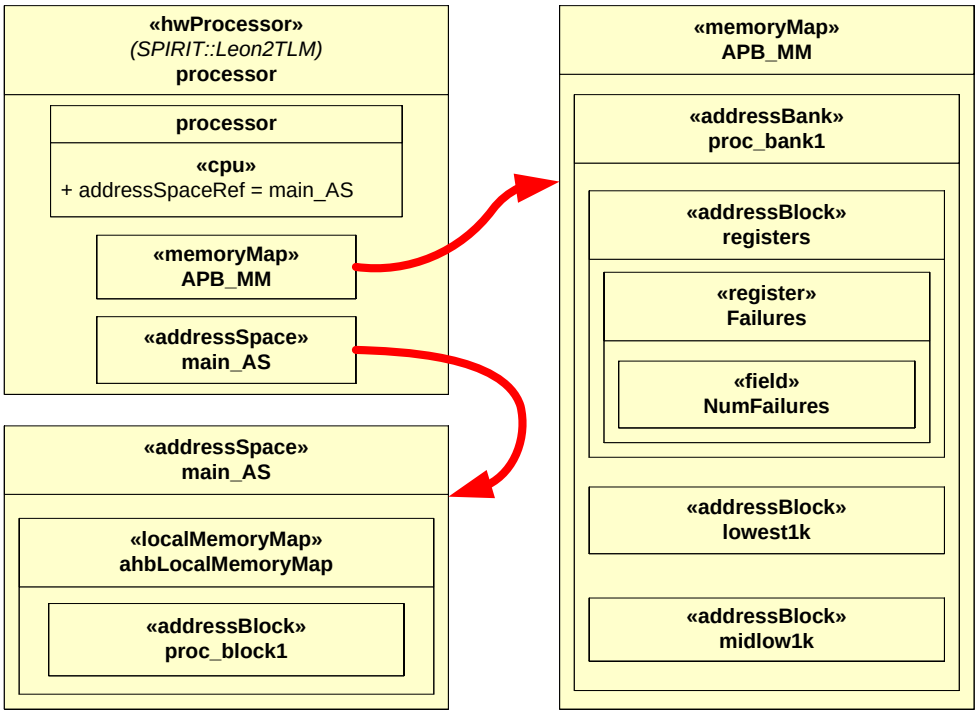


Figure 6.11: Processor Component with memories.

An IP-Xact component contains two types of information relative to memories: address spaces and memory maps, just as shown in figure 6.11. Note that the red arrows are not part of the specification, they only indicates the refinements. Memories are represented using the hierarchical classes in their respective components. Here in the component model, the address space `APB_MM` contains an address bank, several address blocks, and a register containing a field (shown in figure 6.11, right-hand side). These memory units are created in hierarchy just as in IP-Xact files and are stereotyped appropriately. Then in the stereotype attributes, these memory units are properly referred to. For example, the address bank proc_bank1 consists of

three address blocks. All these blocks are represented as nested classes inside the address bank and then they are referred to in the address bank stereotype by the addressBlock attribute (refer back to figure 6.4 on page 87). This is the common approach widely used in this work to tackle with the hierarchy concept of IP-Xact. These memory units could have been referred to by the stereotypes without the use of class hierarchy but that would make these models untidy, difficult to comprehend and most importantly would not be a good representation of IP-Xact models. One thing to remember here is that IP-Xact as a whole is a very complex standard with large set of attributes for various object. We have focused our efforts on the key elements only and did not tried to map each and every thing. Figure 6.11 also shows the `Cpu` component with a reference to the name of the processor.

Figure 6.12 shows the file set and model view sub-modules of the component. IP-Xact uses file sets to refer to external behavior files, like 'processor.cc' and 'processor.h' shown here. The component `view` defines the environment for the IP and the language used for the behavior specification of the IP. Moreover, it contains the name of the design file, if the component is hierarchical.

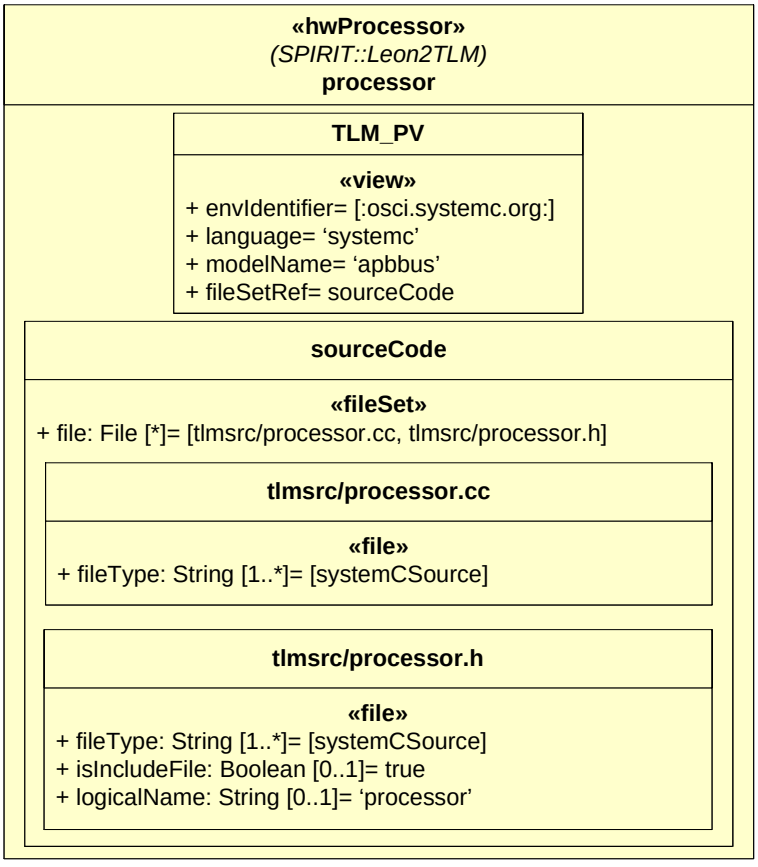


Figure 6.12: Processor Component Definition in UML: Associated code.

Figure 6.13 shows another IP-Xact component represented in UML. This component is ahbbus22, an AMBA AHB bus. It contains the structural information about the bus along with

the source code files representing the bus behavior (like arbitration). The bus component differs from the other components in the way that it usually contains the mirrored bus interfaces. Moreover, it also has a Channel element which contains the logical collections of bus interfaces present on the component. All these interfaces in the channel must be mirrored interfaces.

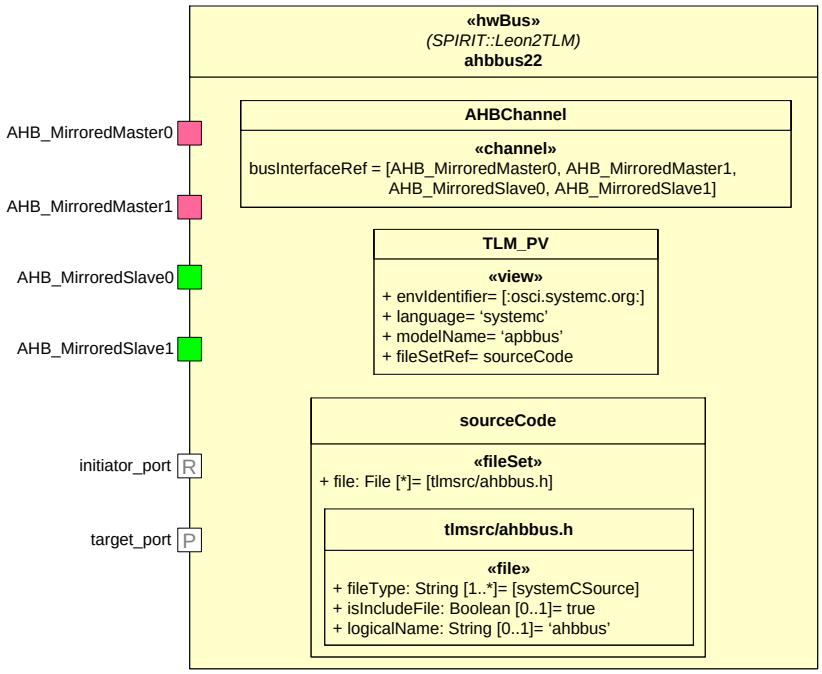


Figure 6.13: AMBA AHB Bus Component in UML.

6.4.3 Design and Hierarchical Components

After discussing the IP-Xact components and their sub-elements, we shift our focus to the hierarchical components. Hierarchical components are same as the simple IP-Xact components but contain the internal design consisting of instances of other components and their interconnections. In IP-Xact, hierarchical components are represented using two files, one for the component definition itself (IP-Xact component file) and the other for the internal design of that hierarchical component (IP-Xact design file). In the subsequent subsections, we discuss the different views points of representing IP-Xact hierarchical components followed by the modeling of Leon II example hierarchical component as a UML model for IP-Xact.

Discussion

Hierarchical components and their internal designs are a controversial (and most important) issue amongst the researches working on the link between IP-Xact and UML models. Because hierarchical components consist of instances of other components, some authors [Rev08, SXM09] have preferred to represent IP-Xact component designs as object diagrams instantiating other components. The use of object diagrams shows that the authors consider the design files to be at the *instance level* instead of the *model level*. This also raises questions in the mind that how

come the elements at different modeling levels can communicate with or refer to each other, as the views of components refer to their respective design files. Some others [ZBG+08] have interconnected the UML components (representing IP-Xact components) directly and have not given any explicit solution to deal with the hierarchical components. Whereas some authors (including us) [ASHH09, AMMdS08] have used UML composite structure parts to represent IP-Xact component instances. In one of our previous attempts to represent IP-Xact [AMMdS08], we used component diagrams to represent the IP-Xact components and the composite structure diagrams to represent the design structure. In that approach we had produced two models to represent an IP-Xact hierarchical component, just as is the case of IP-Xact specification.

To resolve this issue, we have to closely observe the IP-Xact design files and their related components. Each IP-Xact hierarchical component is represented by two IP-Xact files: a component file and a design file. The hierarchical component files define all their interfaces (either bus interfaces or ports). They are just like any other non-hierarchical component with the only difference that the hierarchical components contain the reference to their respective design files using the hierarchyRef attribute of their views. Design files instantiate the components (other than the parent componen that refers to it) and then interconnect them with each other and to the interface of the container hierarchical component. This design file does not define the container component's interfaces and directly uses them. Thus this design file is of no use without the help of the original container component. This whole concept is easy to understand for the people having the know-how of VHDL or Verilog models, as IP-Xact structural representation is very close to these hardware description languages. In VHDL, the design file is the one that instantiates other components and interconnects them. It exists at the same modeling level as other component files and just encapsulates the component entities. One more thing that confuses the whole matter is the use of ConfigurableElementValues to add specific attributes to the component instances. At an initial look it looks as if an object has been initialized but in fact this is not the case. This configuring is just usual in VHDL design files where we can parameterize the component instances by providing the instantiation values. One such VHDL instantiation example from our work is:

```
u_Ccsl_R_sampling: Ccsl_R_sampling port map
  (
    clk, rst,
    A => rEdge,
    B => msti(1).hready,
    O => O,
    KIND => '1',
    KIND_data => Ccsl_PrecKIND_NONSTRICT
  );
```

where the signals clk, rst, A, B, and O are linked to their respective targets while the KIND and the KIND_data signals are used to parameterize/initialize the component instantiated with the values of 1 and a predefined constant `Ccsl_PrecKIND_NONSTRICT` respectively. The IP-Xact configurable elements are a more general way to configure IP components and the design file can itself specify the parameter and its value. These configurable elements are helpful to specify such constraints which can not be described in the IP-Xact component itself. *coreAssembler*, the professional modeling tool from Synopsys, also converts the IP-Xact design files into such 'glue' VHDL files which interlink other components. The keywords confusing the IP-Xact audience are instance from UML and instantiate from VHDL, where the latter is used in the concepts of IP-Xact. Considering these arguments, the use of object diagrams to represent IP-Xact design files in UML is not preferred. In this case of VHDL models, the object diagrams represent the run-time behavior of a system.

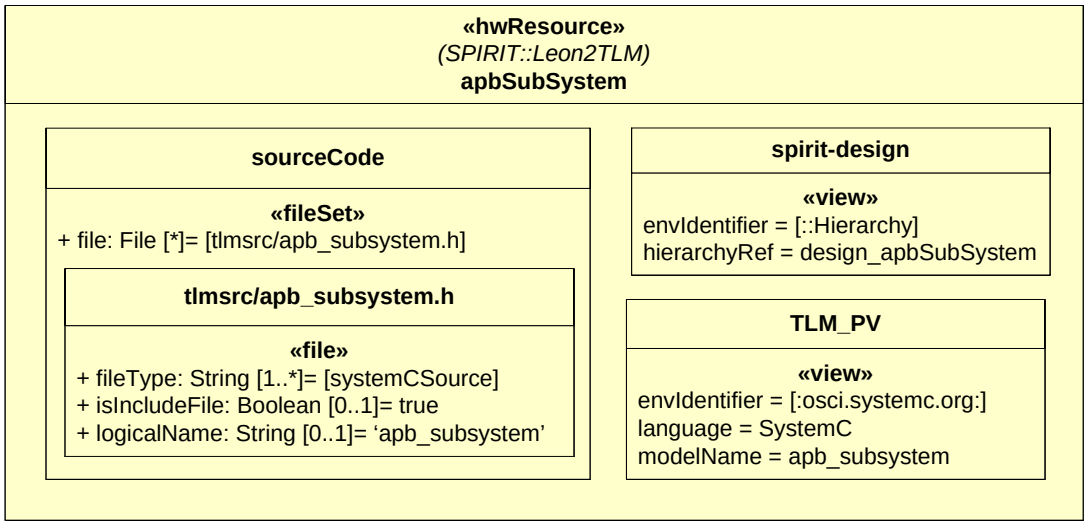


Figure 6.14: AMBA AHB Bus Component in UML: Associated code.

A panel discussion by Bran Selic on 'Modeling of Architectures with UML' and a paper by Y. Vanderperren et al. [VMD08] give a detailed view about the use of different UML diagrams especially in the field of engineering. This paper has a nice discussion on the use of components, structured classes, parts and objects. It clearly states that for engineering applications, *Composite Structure Diagrams* are the most frequently used means to represent hierarchically linked blocks. A Composite Structure Diagram depicts the internal structure of structured classifiers, such as structured classes, by describing the interaction between the internal parts, ports, and connectors. A part represents a set of instances which are owned by an object, for example, or instances of another classifier. Thus we use structured classes to represent each IP-Xact component and design file of a hierarchical component. As stated before, the IP-Xact design files are 'not complete' depending on their container components for the definition of outer interfaces. So we duplicate the IP-Xact component interface (ports and bus interfaces) on the design element. These interface definitions do not exist in the IP-Xact specification of the design file, but we introduce them in the UML models to ease the system designing. Later on during the transformation phase, this redundant information is discarded. This is absolutely necessary as the design files use the component interfaces without defining them but in UML we can not do so. The design connectors directly connect the UML parts with the container component's interface. The design structured class is then referred to, in the hierarchyRef attribute of the view of the component structured class, as shown in figure 6.14. IP-Xact components can have several views (like TLM-PV, vhdlsource from the Leon II example) and each view can refer to an internal design of the component (refer to figure 5.4 on page 64). Using this approach we can achieve any sort of component hierarchy as specified in the IP-Xact. Figure 6.15-A and 6.15-B represent the hierarchical component's bus interface and ports respectively.

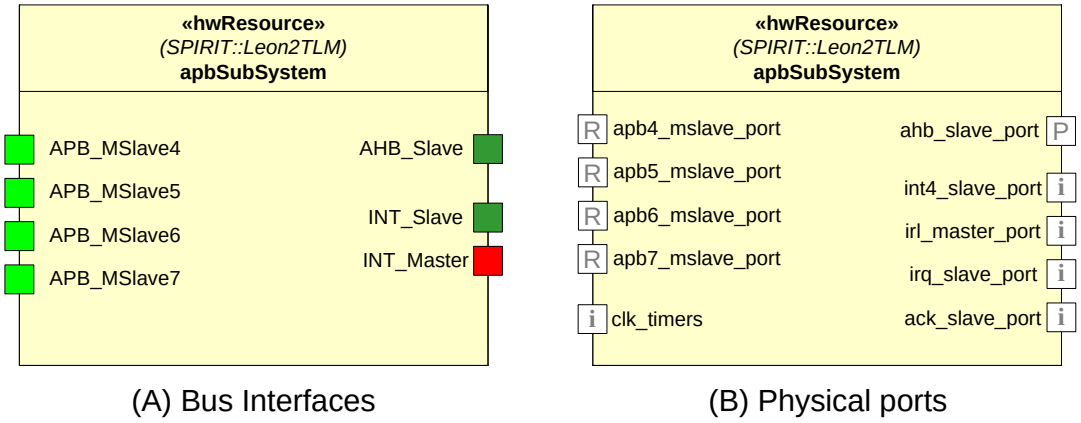


Figure 6.15: AMBA AHB Bus Component in UML.

Modeling Hierarchical Components

The IP-Xact component and design files are represented separately in UML using the structured classes. The 'design' structured class of an IP-Xact component is shown in figure 6.16, representing the TLM view of a component design. Inside the design element, the UML *parts* represent the component instances duly stereotyped with «componentInstance». Here UML gives us the liberty that we can visually hide the ports/bus interfaces that we do not deal with. Also we can graphically show several representations of the same structured classifier (or even a part), which helps us to better organize our models. For example, one representation of the structure classifier may contain only the clock and reset port connections while the other one can have the bus connections. In actual, all these diagrams are only the visual representation of the same element and show a particular dimension of the representation.

There is an important issue with the use of parts as IP-Xact component instances. In IP-Xact, just like VHDL the instance declaration creates a new instance of the original component and we can have several such instances of the same component. But in UML, the composite structure parts represent the property of the structured classifier that relates it to the target classifier. So the parts in UML are not distinct entities and they represent their parent classifiers. So any change on the part (like renaming its port) will be reflected on the component itself. Hence, if we want to create multiple instances of the same IP-Xact component (like processor), then in UML we have to create the structured classifiers representing each of those elements. This task is not of any trouble to a UML designer as the copy/paste of the original classifier will be enough to do the job.

As shown in the metamodel of the IP-Xact design (figure 5.14 on page 75), there are several types of connector elements in IP-Xact including Interconnection, AdHocConnection, HierConnection and MonitorInterconnection. We implement them in our UML models using the UML Connector. In IP-Xact, all these connectors are similar except the source and target elements that they connect to. So in UML we use simple connectors to represent them and later on during the model transformation we judge the source and target elements to determine their type. The Interconnection is a UML connector connecting the bus interfaces of any of the two parts (component instances) in the design whereas the HierConnection (or hierarchical connection) connects the bus interface of a part to the bus interface of the container element. Note that these two connectors only connect the bus interfaces in the design. On the other

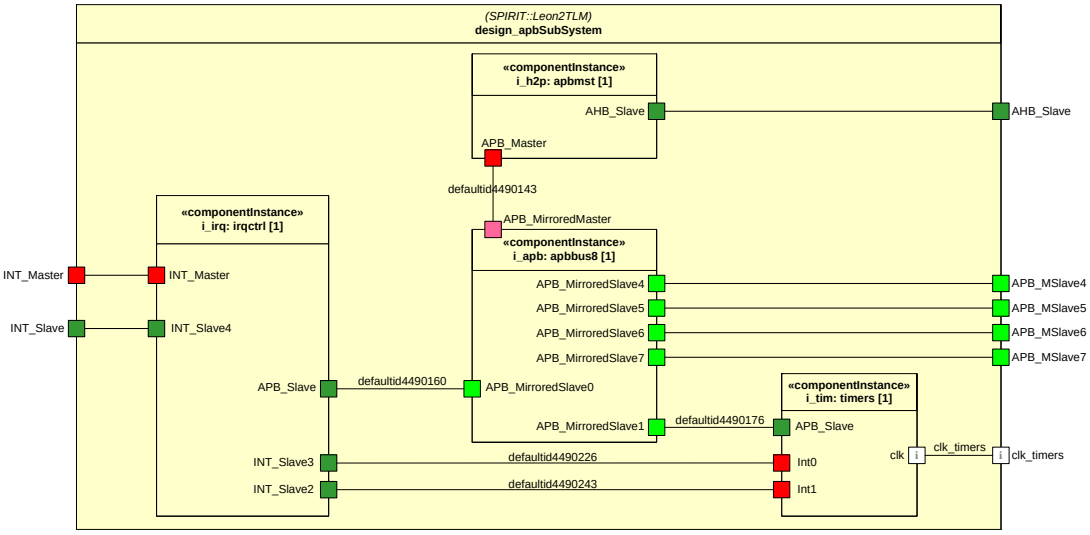


Figure 6.16: Design of Hierarchical Component in UML.

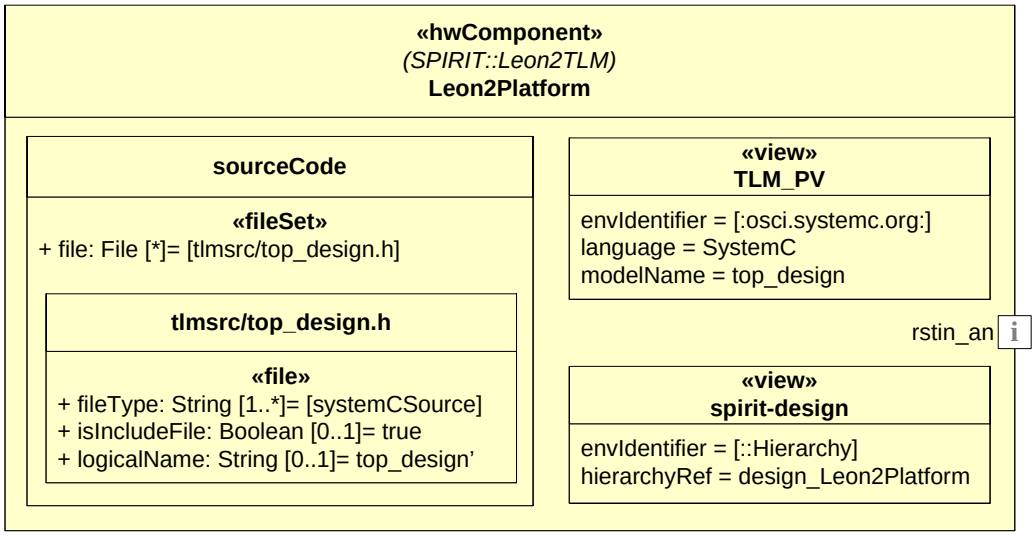


Figure 6.17: IP-Xact Top-level Component.

hand, the AdHocConnection is used to connect the ports within the system. These ports can be on the UML parts or on the container element itself. Hence, the ad-hoc connection represent the combination of interconnections and hierarchical connections but for ports only. The MonitorInterconnection contains only one bus interface at one end while the other end can have one or many monitor bus interfaces connecting to it.

On page 100, figure 6.15-B shows a number of ports on the hierarchical component which

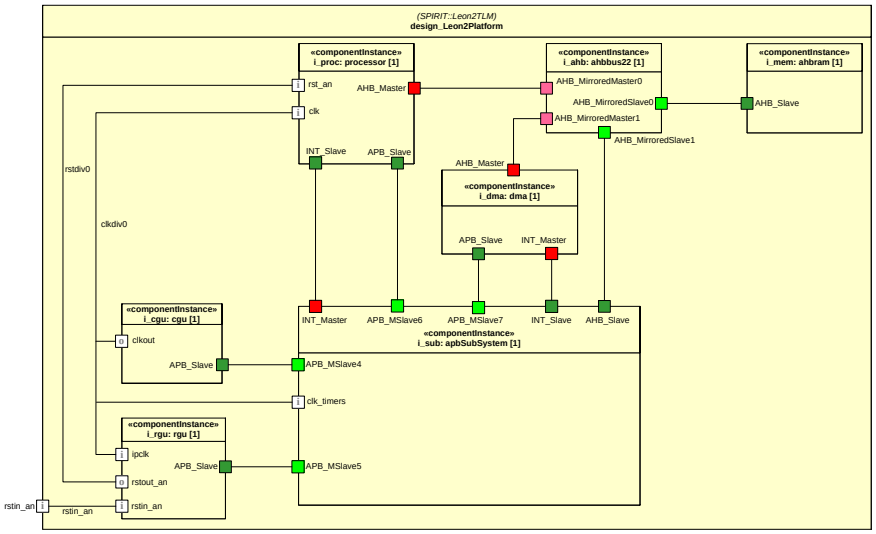


Figure 6.18: Design of IP-Xact Top-level Component.

are defined but are not used in the design. These ports are not connected over here in the diagram but are indirectly used by the bus interfaces of the component. An IP-Xact top level design component is just another hierarchical component with the only difference that it contains minimum number of interfaces (bus interfaces and ports) defined in it. This is quite natural as the top level component is an independent module and does not depend on its environment for execution. Usually the top level component contains the definitions of clock and reset control signal ports, as shown in figure 6.17 and 6.18.

6.4.4 Abstractor

Just like the components and hierarchical components, the IP-Xact abstractor is also represented in UML using the structured classes. The name of the abstractor component is mandatory to be specified along with the vendor and version attributes of the stereotype to uniquely identify the component. The abstractorMode specifies the type of bus interfaces that can connect to the bus interfaces of this component. This abstractor mode is not represented graphically like the ports or bus interfaces of IP-Xact components. The bus interfaces of an IP-Xact abstractor are the UML ports stereotyped with «abstractorBusInterface». The abstractor bus interfaces do not have any interface mode of their own and the abstractor's own interface mode defines their type. These bus interfaces are also not represented graphically and are left as they are. The IP-Xact ports implemented on the abstractor class are just like the ports on the IP-Xact UML components. They are appropriately stereotyped and shown by the icons on the structured class. An abstractor also specifies a mandatory link to the underlying bus definition used by it. Figure 6.19 shows our example IP-Xact abstractor pvapb_2_tacapb bridging the two bus interfaces PV_APB and TAC_APB.

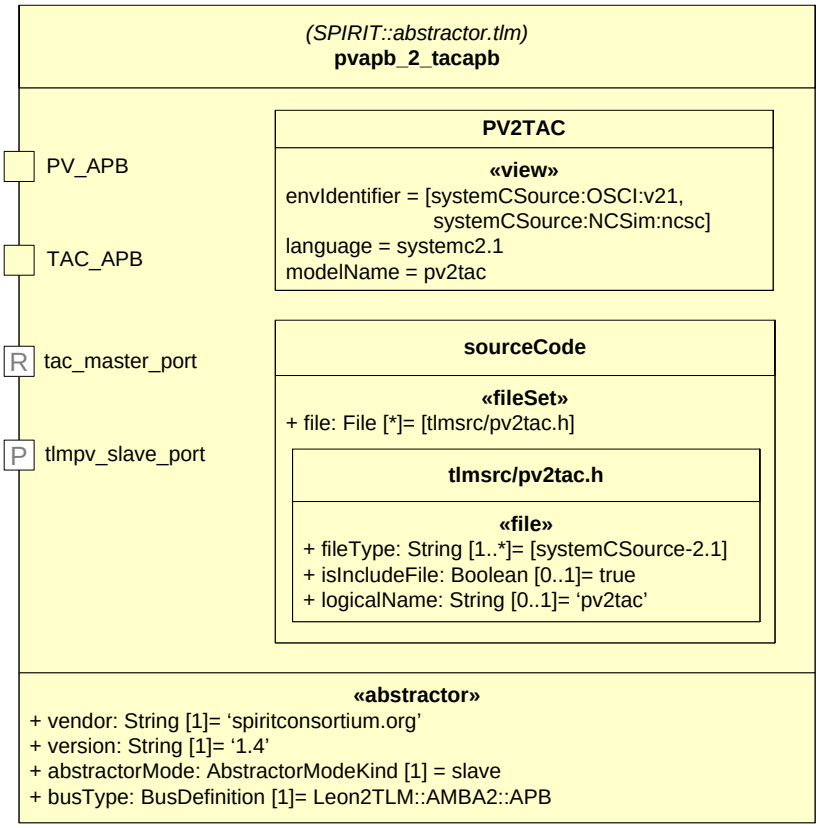


Figure 6.19: An IP-Xact Abstractor Component.

6.4.5 Interface Definitions

Figure 6.20 and 6.21 shows the modeling of interface definitions in UML class diagrams. The interface definitions are implemented by introducing classes with the name of the definition files and stereotyped appropriately. Again just like components, the name and library information about the buses is not saved on the stereotypes and is taken directly from the model, meaning that the name of the classes representing bus definitions and abstraction definitions are mandatory.

The logical ports are the attributes of the abstraction definition class stereotyped with one of the two logical port stereotypes. Here we use the same technique that we used earlier with the IP-Xact component memory to specify the containment relation. Hence, we define logical ports by stereotyping the attributes of the abstraction definition class and then refer to these ports inside the abstractionDefinition stereotype, just as shown in figure 6.20. Unlike the physical ports and the bus interfaces, the logical port attributes (like direction, initiative) are not represented graphically on the models because the logical port behavior varies depending on the underlying physical ports and also because the logical ports do not represent a concrete structure.

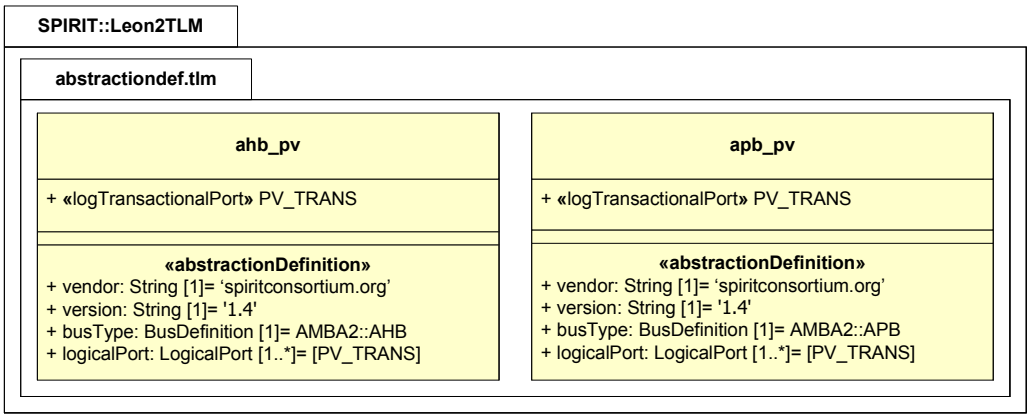


Figure 6.20: Abstraction definition Model Element

As the bus definitions (figure 6.21) could be referenced by the other bus definition (as extension) or the abstraction definitions (as bus type), the best way to model is to initially define all the bus definitions and then create other interface definitions with the appropriate reference to the bus definitions.

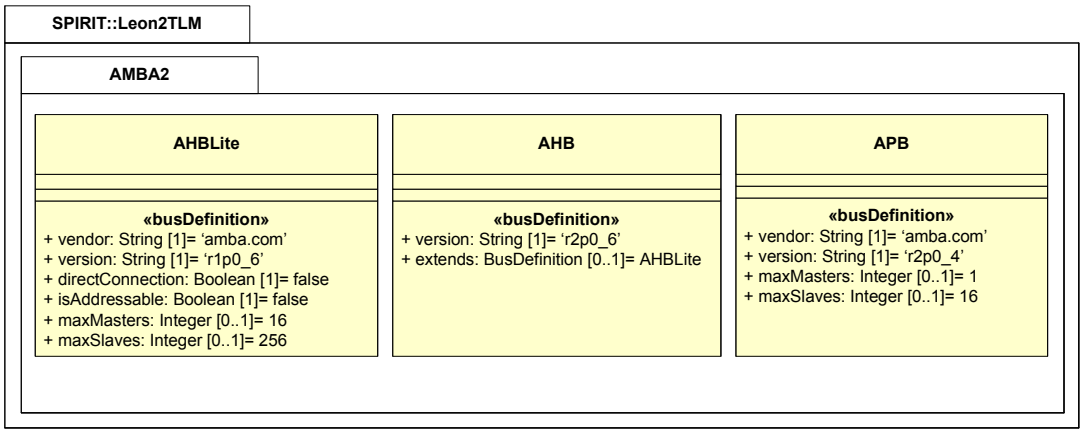


Figure 6.21: Bus definition Model Element.

6.5 Conclusion

In this chapter we discussed the application of our domain view for IP-Xact introduced in the previous chapter. Based on this domain view, we developed our IP-Xact profile for UML based on the existing UML extension of MARTE profile. In the later part of the chapter, we use this profile to model an example of Leon II processor-based architecture which is then used to generate IP-Xact component files. This model transformation technique helps to show the interaction possibilities between the IP-Xact and the ESL profile, illustrating how information

contained in such a profile can be used for a specific implementation.

This chapter concludes our discussion on the possibility of better structural integration and interoperability of IPs. In the next chapters, we focus on the functional aspects of integration and interoperability of IPs.

Chapter 7

Behavior Modeling

Contents

In this chapter, we discuss the issues related to behavior representation at three different abstraction levels using Esterel, SystemC, VHDL. *Then we introduce a more abstract way to represent behaviors related to time through clock constraints specification. Finally we discuss the integration of behavior representations within* IP-Xact.

7.1 Introduction

System Modeling requires representation of both structural and behavioral aspects. In model-based design these aspects, shown on the Y-chart model in figure 2.2 on Page 9, are known as architectural and algorithmic aspects respectively. They are essential parts of the classical SoC design flow.

The structure has been discussed in the previous chapters. The present chapter addresses behavior. Behavior is about the *what* and the *when* of the system. The former concerns what the system have to do (actions, tasks...), the latter specifies partial ordering on these actions. Time-related requirements have direct impacts on the *when*. In classical approaches, time information is often used as extra-functional real-time annotations, giving for instance the duration of an execution or the deadline of an event occurrence. This kind of time information is mostly used in simulation or can be input to (real-time) scheduling analysis tools. In SoC design, time information has more functional intents. During the design, *Multiform logical times* are a better choice than "physical" time. For instance, in a data flow application, we can specify that the activation clock of a producer component is twice as fast as the activation clock of a consumer component, and that the latter component consumes every other produced item. This time specification is clearly functional and should be exploited by circuit synthesizer/compiler and model transformations.

UML being a general-purpose specification language, proves to be a strong candidate for architectural description providing great diversity in designing due to its profile extension mechanism. SysML [OMG08b, Wei08] is one such UML profile dedicated for systems engineering domain, providing UML with the features to model hardware elements. On the behavior representation side also, we prefer to use UML as it utilizes the syntactical notations for its diagrams which are also used by synchronous languages and formal models. UML offers various diagrams like activity, block, state machine and sequence diagrams to represent the system behavior. These behavioral diagrams provide a support for describing *untimed* algorithms, augmented by UML profile for MARTE providing the support for logically or physically timed behaviors. The MARTE extension for UML specifically targets real-time and embedded systems. Using the UML for structural as well as functional representation of a system helps us to create our *golden model* which acts as a reference for all other tools.

The introduction of *Clock Constraint Specification Language* (CCSL) [Mal08] provided UML with the explicit and formal semantics to model the behavioral description of embedded systems. CCSL, introduced in the annex of the MARTE specification (recently adopted by the OMG), provides the timed causality model to the given design. It relies on the MARTE time sub-profile for the behavioral description of UML components. CCSL and logical time (explained before) work in close relation.

In this chapter, we discuss various techniques of behavior representation supported by our running example. In Section 7.2, we discuss the behavior modeling in synchronous languages, followed by VHDL and SystemC representations. Then in section 7.3, we discuss our behavior modeling approach using CCSL specification on our running example.

7.2 Reactive behaviors

In subsection 4.3, we have specified an acquisition system and proposed different formalisms to represent its structural aspects. Now, we turn to the behavioral specifications. The presentation is limited to the behavioral models we have effectively used in this thesis: Esterel (subsection 7.2.1), models with delta cycles (subsection 7.2.2), and CCSL (section 7.3).

7.2.1 Synchronous languages

Imperative synchronous languages support concurrency and offer a wide range of reactive statements. Through examples (components from the acquisition system), we introduce some of the reactive statements of Esterel and explain their behaviors in an informal way. The structural modeling in Esterel given in section 4.3 has only addressed the communications between components. The behavioral modeling demands additional signals that represent interactions of the components with the environment.

Sensor

A sensor gets values of a quantity[1]. The input signal `Val:Data_t` (line 3) represents the obtained value. The measurement takes a non-null amount of time.

```
1  module Sensor:
2    extends ApplTypes;
3    port S_A: mirror AcqT;
4    input Val: value Data_t; // ``physical'' value
5
6    loop
7      await S_A.Sample;
8      await 3 tick; // simulates an acquisition
9      emit ?S_A.Value <= ?Val
10   end loop
11 end module
```

The input signal `Val` stands for the actual value of the sensor (line 4). This value is set by the environment. The keyword **value** indicates that this signal has no presence status: it cannot trigger a reaction. The behavior is specified in lines 6 to 10. A delay as been added (line 8) to simulation the duration of the acquisition. When a measurement completes (at the end of line 8), the sensor sends the valued signal `S_A.Value` that conveys the result of the measurement (line 9).

Processor

The processor initiates two activities: new acquisition and storage of a value. These two activities are triggered by two new signals (line 4). The address where to store a value is determined by a function (line 3) that takes a value as an argument and returns an address. In Esterel the semantics of the function is abstract. It is supposed to be free of side-effects on program signals/variables and executed instantly (0-duration).

```
1  module Processor:
2    extends ApplTypes;
3    function DetermineAddr(Data_t):Addr_t;
4    input StartAcq, StartW;
5    port M_A: AcqT;
6    port M_B: MSaveT;
7
8    // behavior
9    loop
10     await
```

[1] quantity is the *"property of a phenomenon, body, or substance, where the property has a magnitude that can be expressed as a number and a reference"*, definition from the International Vocabulary of Metrology (VIM) [JCG08]

```
11      case StartAcq do
12        emit M_A.Sample; // trigger a new acq
13        await M_A.Value; // returned value
14      case StartW do
15        abort
16          sustain M_B.Breq
17        when M_B.Grant;
18        // now the bus is granted: store data
19        emit {
20          ?M_B.Data <= ?M_A.Value,
21          ?M_B.Addr<= DetermineAddr(?M_A.Value)
22        };
23        await M_B.Done
24      end await
25    end loop
26  end module
```

The behavior is specified in lines 9 to 25. It is a cyclic behavior. For simplicity, acquisition and storage are made exclusive. So, the processor waits (line 10) for `StartAcq` (line 11) or `StartW` (line 14). In Esterel, the multiple cases await statement is fully deterministic. If the two signals `StartAcq` and `StartW` are simultaneously present, the processor enters the acquisition activity (lines 12 and 13) and discards the storage request. This activity is very simple: the processor sends its request (line 12) and waits for the value (line 13). The acquisition activity finishes with this reception. The storage activity is more complex because there is a competition for bus access. The processor has to keep emitting the pure signal `M_B.Breq` (line 16) until the bus is granted ('**abort** ... **when** M_B.Grant' statement, lines 15–17). As soon as the bus is granted, the processor emits the valued signals `M_B.Data` (line 20) and `M_B.Addr` (line 21). The two emissions are concurrent (not ordered) and simultaneous.The processor then waits for the signal `M_B.Done` that indicates the end of the storage activity.

Connections

The Esterel code below is an excerpt from the **module** Application.

```
1  module Application:
2      ...
3      input Val[2]: Data_t, EndAcq[2]; // for sensors
4      input StartAcq[2], StartW[2]; // for processors
5      ...
6      signal
7          port SP[2]: AcqT,
8          port MMP[2]: MSaveT,
9          port MSP[3]: SSaveT
10     in
11         run S1/Sensor
12             [
13                 SP[0]/S_A,
14                 Val[0]/Val,
15                 EndAcq[0]/EndAcq
16             ]
17     ||
18         run P1/Processor
19             [
20                 SP[0]/M_A,
```

```
21                      MMP[0]/M_B,
22                      StartAcq[0]/StartAcq,
23                      StartW[0]/StartW
24                  ]
25          ||
26              ...
```

Lines 3 and 4 contain the newly introduced signals. They are arrays of dimension two (to deal with the two sensors and the two processors). The connection between a processor and a sensor is direct. In the Esterel program we use a local port `SP` (declared in line 7). `SP[0]` is connected on the one side to the port `S_A` of the sensor `S1`, and on the other side to the port `M_A` of the processor `P1`.

The behavior of the Bus and the Memory are given in Appendix B.1. We will check in section 8.2 that the acquisition protocol is respected by this implementation.

7.2.2 Formalisms with microstep simulation semantics

Most modeling languages that support concurrent evolutions have been given a *simulation semantics*. Many, like VHDL and SystemC, adopt an *event-driven* simulation. In these simulations several iterations (microsteps or delta-cycles) can be necessary to compute the result of concurrent evolutions at a given point in time.

Microsteps are found in several StateCharts semantics, for instance, the "asynchronous time model" in the Statemate semantics of StateCharts [HN96]. *Delta-cycles* have been introduced in HDL and are also present in system-level design languages like SystemC. We briefly describe the simulation semantics of VHDL and SystemC.

VHDL simulation semantics

Elaboration and execution of a VHDL model are specified in the VHDL Language Reference Manual [IEE00, chapter 12]. *Elaboration* is the process by which declarations become effective. The elaboration of a VHDL design hierarchy results in a collection of processes interconnected by nets, named by the standard as *model*. This model can then be *executed* in order to simulate the design. A simulation consists of executions of interacting user-defined processes. These executions are coordinated by an event driven *simulation kernel*, also known as the VHDL simulator. Remind that in VHDL, any *concurrent signal assignment* is equivalent to a process that contains only this signal assignment and is sensitive to all the signals occurring in the right-hand side part of the assignment.

The simulation is a cyclic process. It comprises a sequence of *simulation cycles*. A global clock holds the *current simulation time*. This time is not decreasing and is incremented by discrete steps. Usually, several simulation cycles, called *delta cycles*, are executed at the same simulation time. The delta-delay, which separates the successive delta cycles, is considered as an infinitesimally small interval of time. The issue is to determine when the simulation time has to (effectively) progress and when a delta cycle has to be performed. The key concept is the activity status of a signal and its associated *driver*.

A signal driver contains the *projected output waveform* of the associated signal. The projected waveform is a sequence of *transactions*. Each transaction[2] consists of a value/time pair. The VHDL transactions are ordered with respect to their time component. The projected waveform represents the expected future values of the signal at precise points in time. For each driver, there is exactly one transaction whose time component is not greater than the current

[2]Note that in VHDL *transaction* has a very specific and somewhat confusing meaning. In the rest of the thesis, we explicitly mention "VHDL transaction" when ambiguity may exist.

simulation time. The value of this transaction is the *current value* of the driver. As simulation time advances, the time component of the next transaction in a driver may become equal to the current time. In this case, the first transaction is deleted and the next becomes the current transaction of the driver. The driver is said to be *active* during this simulation cycle and the associated signal also. Note that this updating does not imply a change of the value, however if it causes the current value of that signal to change, then an *event* is said to have occurred on the signal. This event can then awake one or many processes that are sensitive to this signal. Each awoken process resumes its execution. In turn processes can update signals, and so on.

Thus, a simulation cycle consists of two separate phases: active signal updating and processes executions.

1. Each active signal is updated. This may cause events.
2. For each process P, if P is currently sensitive to a signal on which an event has occurred in this simulation cycle, then P resumes and executes[3] until it suspends.

If any driver becomes active during the simulation cycle, time is not passing and a delta cycle is executed instead. In fact, the current time changes only after a "steady-state" is reached.

Because of the neat separation between updating and processing phases during a simulation cycle, the result of the simulation is deterministic: it does not depend on the order in which processes are executed.

SystemC simulation semantics

Elaboration and simulation semantics of SystemC are specified in the SystemC Language Reference Manual [IEE05, chapter 4]. On many points, they are similar to the VHDL ones. *Elaboration phase* consists of the instructions coming before the sc_start() function call including the initialization of data structures and connectivity of interfaces. These steps then lead to the *execution phase*, where the control flow is transfered to the SystemC simulation kernel which then performs the step-by-step execution of process threads to emulate concurrency. The *SystemC simulation kernel* is event driven. Processes are scheduled based on their sensitivity to events. Contrasting with VHDL, SystemC can model software and non-deterministic behavior. Thus, a process may call function **notify** to emit an event that will be used immediatly, in the next delta cycle or at some future simulation time. A simulation cycle consists of two separate phases:

1. *evaluation phase:* the scheduler selects a process P from the set of runnable processes. It triggers or resumes the execution of P. P executes without being preempted up to a point where it either returns or calls the function **wait**. An executing process may call function **request_update**, which will cause a function **update** to be called during the very next update phase. The evaluation phase terminates when there is no eligible process left.
2. *update phase:* any and all pending calls to function **update** are executed.

When the update phase terminates, if time-outs exist or delta event notifications have been issued in the current cycle, then the simulator executes a delta cycle. Otherwise, it advances to the next simulation time that has pending events.

Finally, when no further evaluation cycle is left, the simulation terminates by calling the *cleanup phase* (this includes the class destructors). figure 7.1 sums up the different phases.

[3] Actually only a *non postponed* process executes.

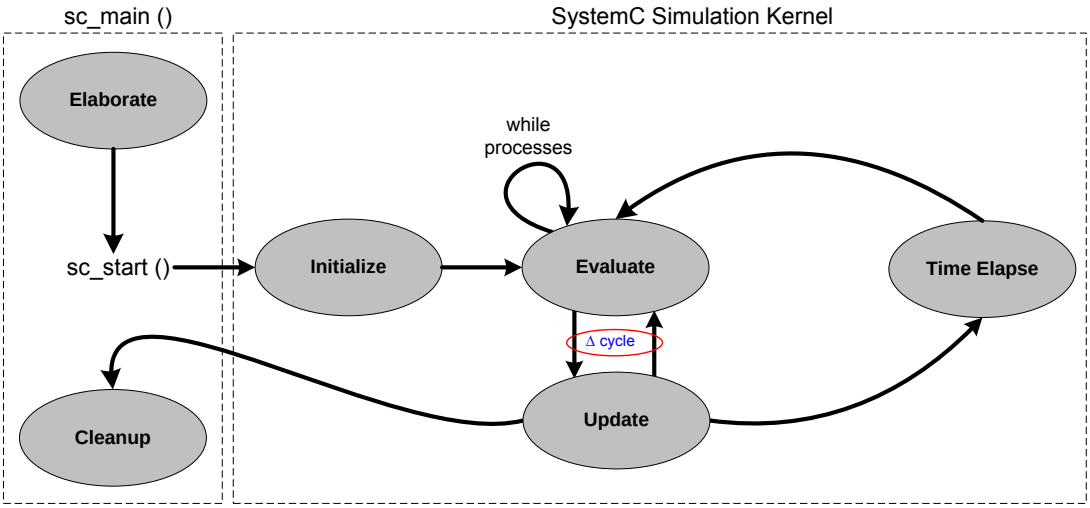


Figure 7.1: SystemC simulation phases

Being based on C++, SystemC programs can be non-deterministic. Even though SystemC can be used in a restricted way to model hardware systems at RTL level, the immediate notification (**notify**()), which causes an event to be generated and used in the very same evaluation phase, may lead to unexpected behaviors.

Microstep vs. perfect synchronous evolutions

Even if in VHDL (and in some SystemC programs) delta cycles allow deterministic simulations, they may cause *glitches* (*i.e.*,a false or spurious transient signal variations). This is usual in combinatorial circuits and is generally harmless. This is not the case when the glitch triggers some visible effect. Through a simple example we explain the relationship between delta cycles and glitches in VHDL. We then explain why the same circuit modeled in Esterel is free of glitch. Finally, we give a solution to eliminate glitches in VHDL models.

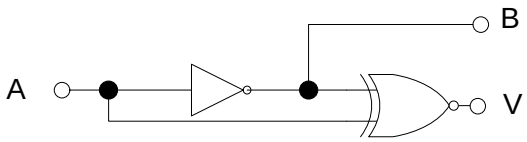


Figure 7.2: Simple circuit with transient signal.

The circuit (figure 7.2) has one input signal A and two output signals B and V. The logical equations are:

$$B = \neg A \tag{7.1}$$
$$V = (A \Leftrightarrow B) \tag{7.2}$$

Therefore, $V = \big(A \Leftrightarrow (\neg A)\big) = false$, that is, V should always be '0'.

The following VHDL code represents this circuit. An assertion (line 6) says that V should always be '0'. When a violation of this assertion occurs a warning is emitted.

```
1  entity CheckExclusion is
2     port ( A: in Bit; B,V: inout Bit);
3  begin
4     process(V) is
5     begin
6        assert V = '0'
7            report "Violation" severity warning;
8     end process;
9  end entity CheckExclusion;
10
11 architecture  CExc of CheckExclusion is
12 begin
13    gate1: B <= not A;
14    gate2: V <= A xnor B;
15 end architecture Cexc;
```

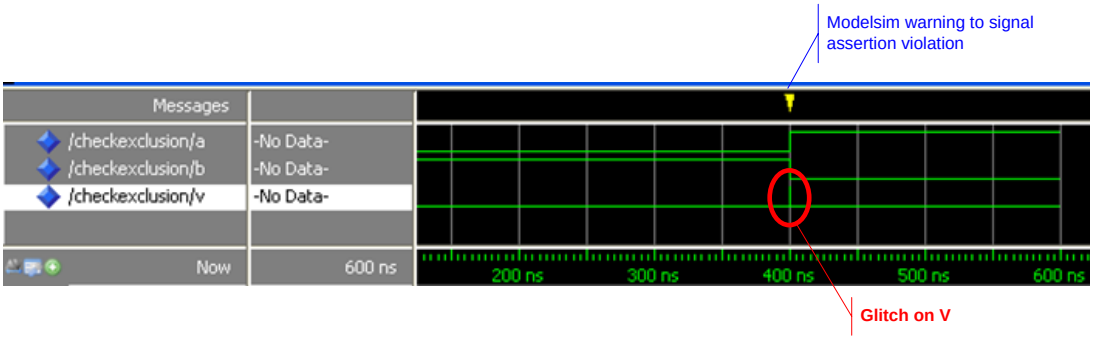


Figure 7.3: VHDL Simulation.

Figure 7.3 shows a simulation trace. At the simulation time 400 ns a violation occurs and a 0-width glitch appears on V. The expression "0-width glitch" means that zooming in the waveform will not increase the width of the glitch: its duration is 0, or more precisely one or several consecutive delta cycles. This can be explained as follows. At the simulation time 400 ns, A goes from '0' to '1'. Just before this time A was '0', B '1', and V '0'. So, A is active and an event is generated on A. The (implicit) processes associated with gate1 (line 13) and gate2 (line 14) are sensitive to A so that they compute the new values of B and V: '0' and '1' respectively. Since B becomes active, a new simulation cycle (a delta cycle) is executed. During the update phase, B and V are effectively updated to '0' and '1' respectively, while A is left unchanged to '1'. During the execution phase a new value ('0') is computed for V. Since V has become active, a new delta cycle is launched. V is updated to '0'. Since no process is sentitive to V the simulation cycle terminates. The circuit is in a new steady-state: A set to '1', B to '0', and V to '0'. It appears that V has got a transient '1' during the first delta cycle.

Now, consider an Esterel program for the same circuit:

```
1  module CheckExclusion:
2     input A;
3     output B, V;
4     sustain {
5        V if not (A xor B),
6        B if not A
```

```
7       }
8   end module
```

The statement **sustain** { ... } (lines 4 to 7) is an infinite loop that executes its inner statements at each instant. The order in which the inner statements are written is irrelevant. The statements are conditional. B is emitted whenever A is absent. V is emitted whenever A and B are both either present or absent. An execution trace (figure 7.4) shows that there is no glitch on V.

The difference of behavior between the VHDL and the Esterel simulation is due to the different underlying semantics. Esterel does not rely on microsteps. It is forbidden for a signal (say V in our example) to have different status during a reaction. Whenever A is present, B is absent (line 6) and V is absent (line 5). The compiler determines an execution ordering that respects the *coherence rules* (section 2.3.2, on page 15): B, then V so that no microsteps are needed.

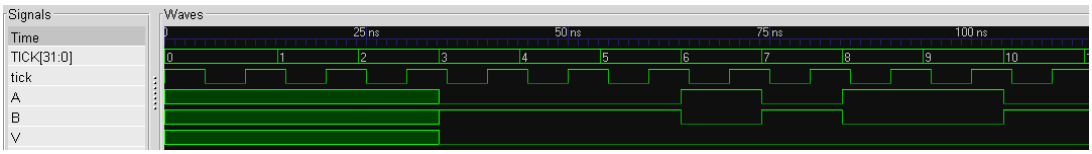


Figure 7.4: Esterel Simulation.

0-width glitches could be avoided in VHDL if only steady-states were considered. VHDL'93 has provided a new facility that is useful in delta delay models. The keyword **postponed** allows deferred executions of a process during delta cycles. During an execution phase, a postponed process is not executed even if it is currently sensitive to a signal on which an event has occurred. Instead, it waits for the end of the *last* delta cycle of the current simulation time to execute. Of course, a postponed process must not cause a new delta cycle. In the following VHDL code, the assignment of V is **postponed** (line 14). This way, there is no glitch on V as shown in the execution trace (figure 7.5).

```
1  entity CheckExclusion is
2     port ( A: in Bit; B,V: inout Bit);
3  begin
4     process(V) is
5     begin
6        assert now = 0 fs or V ='0'
7            report "Violation";
8     end process;
9  end entity CheckExclusion;
10
11 architecture  CExc of CheckExclusion is
12 begin
13    gate1: B <= not A;
14    gate2: postponed  V <= A xnor B;
15 end architecture CExc;
```

The capability of Esterel and VHDL (with **postponed**) to check signal values in steady states will be used in our property observers (section 8.2).

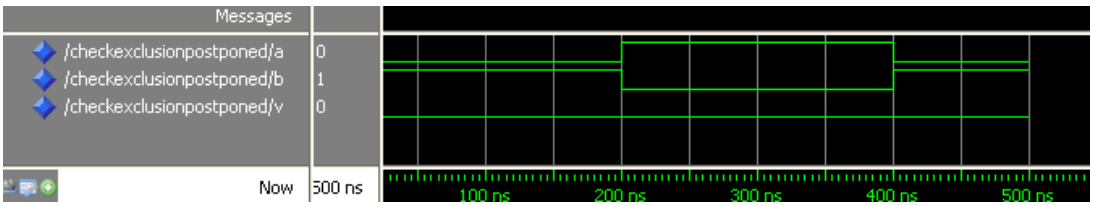


Figure 7.5: VHDL Simulation with postponed process.

7.2.3 Transaction Level Modeling

In the domain of embedded systems, SystemC along with its transaction level modeling libraries are used for the early validation of embedded softwares. This early validation is possible due to the ability of SystemC to represent both hardware and software through the use of virtual platforms. The main elements of an embedded system are modeled as asynchronous processes communicating with each other. As expressed before, the Esterel and VHDL descriptions rely on signals whereas the TLM description in SystemC uses transactions or function calls for communication. Here we describe the behavior of the master, slave and top-level modules for our running example of acquisition system with the focus on data transactions. This focus will help us to better understand the TLM style of modeling electronic systems.

Processor

```
1  #include "processor.h"
2
3  processor::processor(sc_module_name module_name):
4    sc_module(module_name),
5    M_A("M_A"),
6    M_B("M_B"),
7    Start("Start")
8  {
9    SC_THREAD(Compute);
10 }
11
12 processor::~processor() {}
13
14 void processor::Compute() {
15   tlm::tlm_status status;
16   Command_t lCmd;
17   while(1) {
18     wait(eStart);
19     eStart.cancel();
20     wait(SC_ZERO_TIME);
21     lCmd = Start.read();
22     switch(lCmd)
23     {
24     case ACQ: // read sensor
25       status = M_A.read(0, lData); // only 1 address
26       if (status.is_ok()){
27         cout << processor::name() << ": Data Acquired"
28           << endl;
29       } else {
```

```
30            cout << processor::name() << ": Data Acquire Failed"
31              << endl;
32          }
33          break;
34        case SAVE: // save the value contained in lData
35          lAddr = calcAddr(lData);
36          status = M_B.write(lAddr, lData);
37          if (status.is_ok()){
38            cout << processor::name() << ": Data Sent" << endl;
39          } else {
40            cout << processor::name() << ": Data Sending Failed"
41              << endl;
42          }
43          break;
44        }
45      }
46    }
```

The programming in SystemC is modular, based on threads (SC_THREAD) and modules (SC_MODULE). Here for the processor module we use thread function named `Compute` (lines 9, 14–45) to process the transaction commands sent to the sensor and the bus modules. As usual in SystemC, the thread contains an infinite loop (lines 17–44) to process the instructions repetitively. The header of the class contains the component definition along with interface bindings (lines 3–7). For simulation purposes we have introduced a signal and a SystemC event in the processor component's header file, just as given next.

```
sc_in<Command_t> Start;
sc_event eStart;
```

The role of these simulation signals is very simple and is almost similar to the ones used in Esterel program. The `Start` SystemC input port is used to get the input command value which is then used to decide the operation of the processor. Input commands are defined in the `types.h` file as enumeration literals:

```
enum Command_t {ACQ, SAVE};
```

Hence, for an input command of 'ACQ', the processor sends a request to the sensor for data acquisition while the input command 'SAVE' causes the processor to send the stored data ('0' if nothing is stored previously) to the appropriate slave memory module. The `eStart` signal is used to detect the arrival of new commands by using in the wait statement (line 18) inside the process thread. Once an `eStart` event is detected by the wait statement, we use `cancel()` SystemC routine to prepare for the next event notification. The `calcAddr` function is a local function called (line 35) in the `Compute` thread to calculate the address of the data received from the sensor. For our experimentation, this address calculation is based on the value of data received. In line 20, we use the wait statement for a time period of SC_ZERO_TIME, which is the delay of delta time allowing the signal values to reflect the change occurred in the same simulation cycle. This concept is previously described in figure 7.1 on page 113. The data reading and writing is performed through the simple `read()` (line 25) and `write()` (line 36) function calls. Here the main difference lies between the programming at TLM level from the Esterel and VHDL programming: we use function calls to initiate or to respond to data transactions.

Memory

Here we consider the slave side memory module (same applies to sensor module) where these data transactions terminate. In the constructor of the memory module we bind the slave port

to the class itself (line 8) followed by an initialization of the memory cells (line 9). The write and read operations are the essence of the slave module functionality. In the write function, we store the `data` argument passed to the specified `addr` address location (line 23). Note that in the case of write function, the `data` and `addr` arguments are passed as read-only arguments, while in the case of read function, the `data` argument is passed-by-reference (line 29). Lastly, `tlm_status` data type defined in both the functions is a class definition from the TLM library which ensures about the validity of the transactions occurred and also communicates the transaction status between master and slave modules.

```
1  #include "memory.h"
2
3  memory::memory(sc_module_name module_name):
4    sc_module(module_name),
5    pv_slave_base< Addr_t,Data_t >(name()),
6    S_B("S_B")
7  {
8    S_B( *this );
9    for (int i=0; i < 0x1000; i++) iMemory[i]=0;
10 }
11
12 memory::~memory() { }
13
14 tlm::tlm_status memory::write(
15   const Addr_t &addr, const Data_t &data,
16   const unsigned int byte_enable,
17   const tlm::tlm_mode mode,
18   const unsigned int export_id)
19 {
20   tlm::tlm_status status;
21   printf ("\%s : Writing DATA = \%x at ADDRESS = \%x \n",
22   memory::name(),data,addr);
23   iMemory[addr] = data;
24   status.set_ok();
25   return status;
26 }
27
28 tlm::tlm_status memory::read(
29   const Addr_t &addr, Data_t &data,
30   const unsigned int byte_enable,
31   const tlm::tlm_mode mode,
32   const unsigned int export_id)
33 {
34   tlm::tlm_status status;
35   data = iMemory[addr];
36   printf ("\%s : Reading DATA = \%x at ADDRESS = \%x \n",
37   memory::name(),data,addr);
38   status.set_ok();
39   return status;
40 }
```

Simulation

Finally in the top level module (`top.h`), we instantiate the components and the `Simulation()` thread (as explained before in section 4.5 on page 56). This thread does not contain any

`while(1)` infinite loop and hence executes only once during the simulation. It inputs the required data values to run the system as desired. First step of the simulation consists of writing 'ACQ' to the `myStart0` and trigger `eStart` event by the `notify()` SystemC call (line 10). This command directs the processor to request the sensor for data acquisition. The second (and the last) step instructs the processor for storing this data in a memory location attached to the bus (line 15). There are several `wait()` statements introduced in the code to trigger different transactions.

```
1  void Simulation() {
2    // Start Simulation
3    wait(50,SC_NS);
4
5    // System Reset
6    cout << endl << "@Top: Resetting System at time "
7                 << sc_time_stamp() << endl << endl;
8    // System Initialize
9    cout<< "  start0 = ACQ  " << endl;
10   myStart0.write(ACQ);  i_proc_0->eStart.notify();
11   ...
12   wait(50,SC_NS);
13
14   cout<< "  start0 = SAVE " << endl;
15   myStart0.write(SAVE); i_proc_0->eStart.notify();
16   ...
17   wait(50,SC_NS);
18 }
```

7.3 Modeling with logical clocks

7.3.1 Multiform logical time

Leslie Lamport [Lam78] introduced logical clocks in the late 70's. The logical clocks associate numbers (logical timestamps) with events in a distributed system, such that there exists a consistent total ordering of all the events of the system. These clocks can be implemented by counters with no actual timing mechanisms. In the 80's, the synchronous languages [BB91] introduced their own concept of logical time. This logical time shares with Lamport's time the fact that they need not actually refer to physical time. Logical time only relies on (partial or total) ordering of instants. In what follows, we consider logical time in the sense of synchronous languages. In the synchronous language Signal, a *signal* s is an infinite totally ordered sequence $(s_t)_{t\in\mathbb{N}}$ of typed elements. Index t denotes a *logical instant*. At each logical instant of *its* clock, a signal is present and carries a unique value. Signal is a multi-clock (or polychronous) language: it does not assume the existence of a *global clock*. Instead, it allows multiple logical clocks. Signal composition is ruled by operators which are either mono-clock operators (composing signals defined on a same clock) or multi-clock operators (allowing composition of signals having different clocks).

Logical time is widely used in electronic system design. The activity of a processor makes reference to its "clock". This clock can be closely linked to physical time, when the clock is for instance generated by a quartz oscillator. However, with power-aware systems, the period of the clock can be dynamically changed. Thus a logical clock, whose ticks are associated with (not necessarily evenly interspaced) clock pulses, is a far better time reference than usual

chronometric clocks for this kind of applications. Note that hardware description languages (HDLs) [BK08] also refer to logical clocks in their simulations.

Indeed, a logical clock can be associated with any event. This point of view has been adopted in the MARTE time model [OMG08c, Chap. 10]. A logical clock "ticks" with each new occurrence of its associated event. Synchronous languages like Esterel exploit this property. In an Esterel program, time may be counted in seconds, meters, laps... (see the Berry's RUNNER program [Ber00a] which describes the training of a runner). This variety of events supporting time leads to the concept of *multiform time*. More technical examples can be found in automotive applications. For instance, the electronic ignition is driven by the angular position of the crankshaft rather than by a chronometric time (see a study of a knock controller in a 4-stroke engine [AMPF07]).

In this thesis, we consider a *multiform discrete logical time*. We briefly introduce discrete logical clocks and relationships between instants of different clocks. Then, clock relations are defined and the specification language for clock constraints is presented.

Relations between instants

We introduce the concept of instant relations through a simple system with three events A, B, and C. Its behavior is as follows: Every n^{th} occurrence of A ask for an execution of action C. The request for execution is the event B. Any execution of C must be done before the next request (*i.e.*,the next occurrence of B).

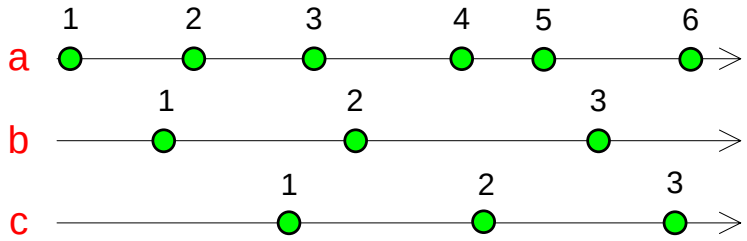


Figure 7.6: Three independent logical clocks.

We associate a clock with each event (clock a with A, clock b with B, and clock c with C). These clocks are *a priori* independent (figure 7.6). The instants of each clock are strictly ordered and indexed by natural numbers.

In fact, dependencies exist between instants of the different clocks. We introduce two kinds of relationships between instants: *precedence* (denoted $\prec$) and *coincidence* (denoted $\equiv$). Figure 7.7 shows these additional relations in a simple case ($n = 2$).

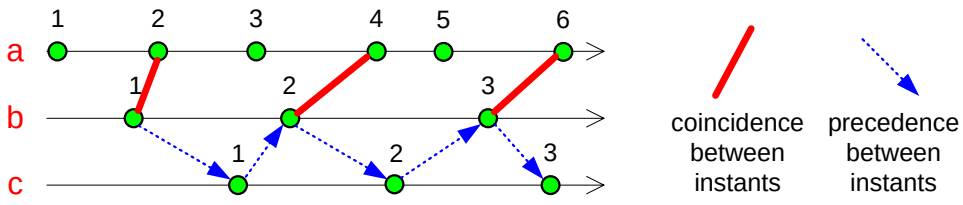


Figure 7.7: Imposing relations between instants.

Let $X[k]$ denote the k^{th} instant of clock X. The relations contained in the figure correspond

to equations:

$$\forall k \in \mathbb{N}^\star,\ b[k] \equiv a[2k] \tag{7.3}$$
$$\forall k \in \mathbb{N}^\star,\ b[k] \prec c[k] \prec b[k+1] \tag{7.4}$$

Eq. 7.3 states that the k^{th} instant of b and the $2k^{\text{th}}$ instant of a are coincident (*i.e.,*the two clocks tick jointly). Eq. 7.4 expresses that the k^{th} instant of b is before the k^{th} instant of c, which is itself before the $(k+1)^{\text{th}}$ instant of b (*i.e.,*ticks of b and c alternate).

These relations result in the *time structure* shown in figure 7.8. It is a partial ordering on instants. Note that coincident instants received a special treatment: they are grouped together. In fact, the time structure is a POset on equivalence classes of the coincidence relation [AMdS07].

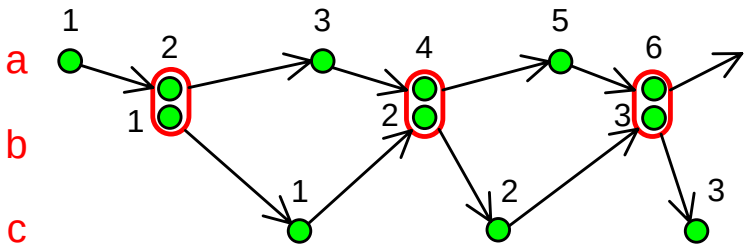


Figure 7.8: Resulting structure.

There exist two other relations on instants: the *non strict precedence* (denoted $\preccurlyeq$) and the *exclusion* (denoted $\#$). They are respectively defined by $\preccurlyeq \triangleq \prec \cup \equiv$ and $\# \triangleq \prec \cup \prec^{-1}$. Table 7.1 sums up the instant relations and their notations.

instant relation	symbol	graphical representation
strict precedence	$\prec$	
(non strict) precedence	$\preccurlyeq$	
coincidence	$\equiv$	
exclusion	$\#$	

Table 7.1: Instant relations

The precedence relation often represents causal dependency. The coincidence relation may reflect strong physical coupling (*i.e.,*revolutions of the camshaft vs. revolutions of the crankshaft), strong synchronizations (*i.e.,*rendez-vous), or simple design choices.

Specifying a full time structure using only instant relations is not realistic. Moreover a set of instants is usually infinite, thus forbidding an enumerative specification of instant relations. Hence the idea to extend relations to clocks. CCSL has been defined for this purpose. It is presented in the next section.

7.3.2 Clock Constraints

CCSL has been introduced in an annex of the MARTE specification [OMG08c, Annex C.3]. It is a language to specify *clock constraints*. This language is non normative (the MARTE profile implementors are not obliged to support it). The semantics of CCSL given in the specification is

informal. A first formal semantics, based on mathematical expressions has been proposed in a paper [Mal08] and a research report [AM08], which is an extended version of the paper. In this thesis we adopt this kind of semantics. The expressiveness of CCSL has been compared to two models (Signal and Time Petri nets) [MA09]. The former can easily express the synchronous clock constraints, the latter is well-adapted to asynchronous clock constraints, but none is convenient to represent mixed clock constraints. A precise definition of the syntax of a *kernel* of CCSL along with a structural operational semantics is now available [And09b, AM09a]. This semantics is the golden reference for the CCSL constraint solver implemented in TimeSquare[4], the software environment that supports CCSL and the MARTE time profile.

A CCSL specification consists of clock declarations and a set of binary clock relations. These relations apply to clock or to clock expressions. This section presents the basic (*i.e.*,part of the kernel) clock relations and a selection of clock expressions, reused in the thesis. These definitions are borrowed from the previously mentioned papers on CCSL.

Clock relations

Let a and b two clocks. Five primitive relations on clocks are defined:

- *Equality:* $a \boxed{=} b$ is a typical synchronous clock relation. There is a bijection between instants of a and b. This bijection is order preserving and the instants are point-wise coincident. In a formal way:

$$a \boxed{=} b \Leftrightarrow \left(\forall k \in \mathbb{N}^\star, a[k] \equiv b[k]\right) \tag{7.5}$$

- *Subclocking:* $a \boxed{\subset} b$ is a weaker synchronous clock relation. The mapping f from a to b is injective and order preserving. a is said to be a sub-clock of b, and b a super-clock of a.

$$a \boxed{\subset} b \Leftrightarrow \begin{cases} \forall k \in \mathbb{N}^\star, \exists l \in \mathbb{N}^\star, a[k] \equiv b[l] = f(a[k]) \\ \forall k_1, k_2 \in \mathbb{N}^\star, a[k_1] \prec a[k_2] \Rightarrow f(a[k_1]) \prec f(a[k_2]) \end{cases} \tag{7.6}$$

- *Precedence:* $a \boxed{\preccurlyeq} b$ is an asynchronous clock relation. a is said to be faster than b.

$$a \boxed{\preccurlyeq} b \Leftrightarrow \left(\forall k \in \mathbb{N}^\star, a[k] \preccurlyeq b[k]\right) \tag{7.7}$$

- *Strict precedence:* $a \boxed{\prec} b$ is similar to the previous one but considering the strict precedence instead.

$$a \boxed{\prec} b \Leftrightarrow \left(\forall k \in \mathbb{N}^\star, a[k] \prec b[k]\right) \tag{7.8}$$

- *Exclusion:* $a \boxed{\#} b$ means that a and b have no coincident instants.

$$a \boxed{\#} b \Leftrightarrow \left(\forall j, k \in \mathbb{N}^\star, \neg(a[j] \equiv b[k])\right) \tag{7.9}$$

We mention an often used non-primitive clock relation: the *alternation*. $a \boxed{\sim} b$ is an asynchronous clock relation such that

$$a \boxed{\sim} b \Leftrightarrow \left(\forall k \in \mathbb{N}^\star, a[k] \prec b[k] \prec a[k+1]\right) \tag{7.10}$$

For instance, in figure 7.7, $b \boxed{\sim} c$.

[4] http://www-sop.inria.fr/aoste/dev/time_square

Clock expressions

A clock expression allows the creation of new clocks from existing ones. We present three examples of clock expressions respectively synchronous, asynchronous, and mixed. Others will be explained when needed.

For the first expression we have to introduce the concept of *binary words*. A binary word is a finite or infinite sequence of bits:

- a *finite binary word* is a word of $(0+1)^*$
- an *infinite binary word* is a word of $(0+1)^\omega$
- a *periodic binary word* is an infinite binary word w defined by:

$$\begin{aligned} w &::= u\,(v)^\omega \\ u &::= \varepsilon \mid 0 \mid 1 \mid 0 \bullet u \mid 1 \bullet u \\ v &::= 0 \mid 1 \mid 0 \bullet v \mid 1 \bullet v \end{aligned}$$

u is called the *prefix* of w, v is the *period* of w, and $(v)^\omega = \lim_n v^n$ denotes the infinite repetition of v. Let $\mathcal{BW}$ be the set of the finite or infinite binary words. For any binary word w, $w[k]$ for $k \in \mathbb{N}^\star$ is the k^{th} bit of w. $w \uparrow k$ denotes the index of the k^{th} one in the binary word w.

Example of clock relations

- *Filtering:* $a \blacktriangledown w$, read 'a filtered by w', where a is a clock and w is a binary word, is a synchronous clock expression which defines a sub-clock, say b, of a such that

$$b \boxed{=} a \blacktriangledown w \Leftrightarrow \forall k \in \mathbb{N}^\star, b[k] \equiv a[w \uparrow k] \tag{7.11}$$

  In figure 7.7, $b \boxed{=} a \blacktriangledown (01)^\omega$

- *inf:* $a \wedge b$, read 'inf of a and b', is an asynchronous clock expression which defines a new clock, say c, such that

$$c \boxed{=} a \wedge b \Leftrightarrow \forall k \in \mathbb{N}^\star, c[k] \equiv \begin{cases} a[k] \text{ if } a[k] \preccurlyeq b[k] \\ b[k] \text{ otherwise.} \end{cases} \tag{7.12}$$

  In other words, c is the slowest clock among those which are faster than both a and b. See figure 7.9 for an example. There also exists the dual operator sup, denoted $\vee$, such that

$$c \boxed{=} a \vee b \Leftrightarrow \forall k \in \mathbb{N}^\star, c[k] \equiv \begin{cases} b[k] \text{ if } a[k] \preccurlyeq b[k] \\ a[k] \text{ otherwise.} \end{cases} \tag{7.13}$$

  That is, c is the fastest clock among those which are slower than both a and b.

- *Delay:* $a\ (\ \delta\ )\rightsquigarrow b$, read '$a$ delayed for δ on b', where δ is a non-null natural number, is an example of mixed expression. It defines a new clock, say c, sub-clock of b, such that:

$$\begin{aligned} c \boxed{=} a\ (\ \delta\ ) \rightsquigarrow b \Leftrightarrow \forall k \in \mathbb{N}^\star, \exists l, m \in \mathbb{N}^\star, l > \delta, \\ c[k] \equiv b[l] \wedge b[l-\delta-1] \prec a[m] \preccurlyeq b[l-\delta] \end{aligned} \tag{7.14}$$

  So, ticks of a plays the role of a trigger, and they cause a tick of c after δ ticks of b. Figure 7.10 illustrates a delay for 2 ticks. Note that this delay operator is polychronous (involving two different clocks a and b) contrasting with the monochronous delay operator of the language Signal [MA09].

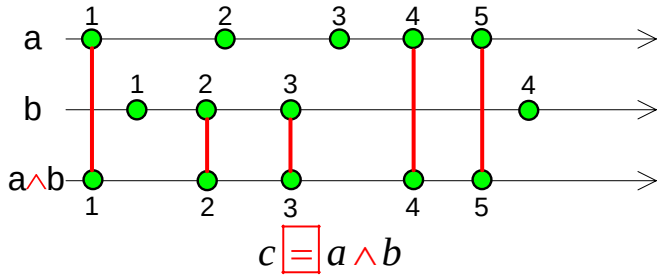


Figure 7.9: The inf expression operator.

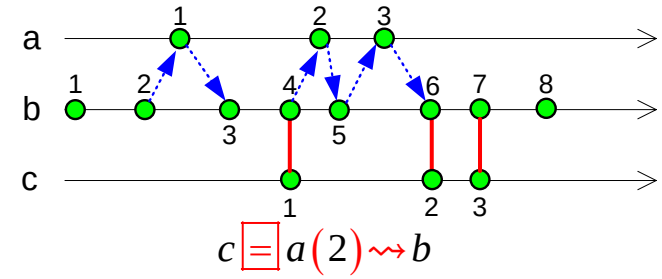


Figure 7.10: The delay expression operator.

7.3.3 CCSL in the Acquisition system

Following the principle that a logical clock can be associated with any event (section 7.3.1), the constraints implied by a protocol can be specified by clock constraints expressed in CCSL. We apply this to the Acquisition system.

Acquisition protocol

The protocol is a simple request/response protocol expressed in CCSL as $req \boxed{\sim} resp$, where req and $resp$ are the logical clocks associated with the events request and response, respectively.

Referring to the Esterel program given in section 7.2.1, we have to specify two instances of the acquisition protocol, where the request is called Sample and the response Value. For simplicity, we adopt the convention that names a clock after the name of the associated event prefixed by a "c_". This leads to:

$$c\_SAP[i].Sample \boxed{\sim} c\_SAP[i].Value \quad \text{for } i = 0, 1 \tag{7.15}$$

Saving protocol

The saving protocol is a generalized request/response protocol with several sources of requests and responses. Moreover a form of exclusion is imposed on the transfers of data.

In the Acquisition example, a bus request (Breq) must precede a bus granting (Grant), which in turn precedes a memory selection (Sel). This precedence chain is repeated forever. To deal with multiple sources, we need the *union* relational operator on clocks.

- *Union:* $a + b$, read 'a union b', where a and b are clocks, is a synchronous clock expression which defines a clock, say c such that

$$c \boxed{=} a + b \Leftrightarrow \forall k \in \mathbb{N}^\star, \exists l, j \in \mathbb{N}^\star, (c[k] \equiv a[j]) \text{ or } (c[k] \equiv b[j]) \tag{7.16}$$

Two local clocks c_G and c_S are used represent any Grant and any Sel, respectively. The exclusion between data transfers is ensured by the alternation of c_G and c_S. The whole protocol is then specified as follows:

$$c\_G \boxed{=} c\_MMP[0].Grant + c\_MMP[1].Grant \tag{7.17}$$
$$c\_MMP[0].Grant \boxed{\#} c\_MMP[1].Grant \tag{7.18}$$
$$c\_S \boxed{=} c\_MSP[0].Sel + c\_MSP[1].Sel + c\_MSP[2].Sel \tag{7.19}$$
$$c\_MSP[0].Sel \boxed{\#} c\_MSP[1].Sel \tag{7.20}$$
$$c\_MSP[1].Sel \boxed{\#} c\_MSP[2].Sel \tag{7.21}$$
$$c\_MSP[0].Sel \boxed{\#} c\_MSP[2].Sel \tag{7.22}$$
$$c\_MMP[0].Breq \boxed{\sim} c\_MMP[0].Grant \tag{7.23}$$
$$c\_MMP[1].Breq \boxed{\sim} c\_MMP[1].Grant \tag{7.24}$$
$$c\_G \boxed{\sim} c\_S \tag{7.25}$$

Equations 7.23 and 7.24 impose behaviors that can be represented by the regular expression $(Breq[i]; Grant[i])^*$. Equation 7.25 imposes the cyclic behavior $(G; S)^*$. Equation 7.18 makes the Grants exclusive; equations 7.20–7.22 do the same with the Sels. Figure 7.11 shows a possible execution trace for this protocol specification. Blue dashed arrows are displayed (on demand) by the TimeSquare viewer. Red annotations have been manually added to the figure.

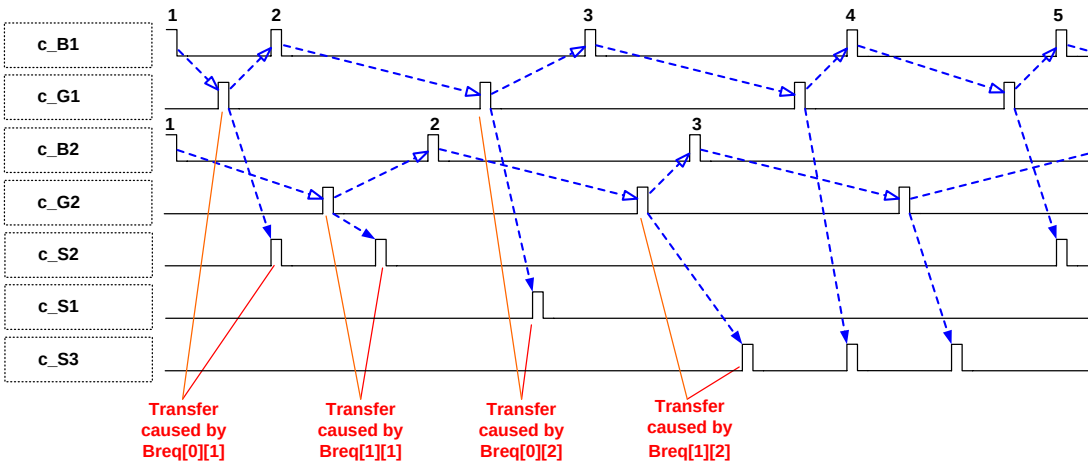


Figure 7.11: A possible execution trace for the Saving protocol.

7.4 IP-XACT and Behavior

7.4.1 Specification of the behavior

IP-Xact offers little support to behavioral specifications. We briefly describe the two supported facilities: expression of time constraints and references to external behavior specifications.

Time constraints

Optional constraints may be provided for *wire ports*. A constraint named requiresDriver specifies whether a driver has to be present in a complete design. Attribute driverType further qualifies what driver type is required. The values `clock` and `singleshot` explicitly refer to time constraints.

Component driver/clockDriver. This element defines the properties of a (repetitive) clock waveform. Four mandatory elements describe the properties of the waveform. A fifth optional element attaches a name to the clock driver.

1. `clockPeriod` (mandatory) specifies the overall length (in time) of one cycle of the waveform. The duration is expressed by a real number and a unit. The default unit is the nanosecond (`ns`) equal to 10^{-9} seconds. Picosecond (`ps`) can be used instead and is equal to 10^{-12} seconds. All other time-related properties adopt the same notation for duration.
2. `clockPulseOffset` (mandatory) specifies the time delay from the start of the waveform to the first transition.
3. `clockPulseValue` (mandatory) specifies the logic value (0 or 1) to which the port transitions. Note that this value is the opposite of the value from which the waveform starts[5].
4. `clockPulseDuration` (mandatory) specifies how long the waveform remains at the value specified by clockPulseValue.
5. `clockName` (optional) attribute specifies a name for the clock driver. If this is not defined, the name of the port to which this clockDriver is applied shall be used.

Figure 7.12 depicts these elements.

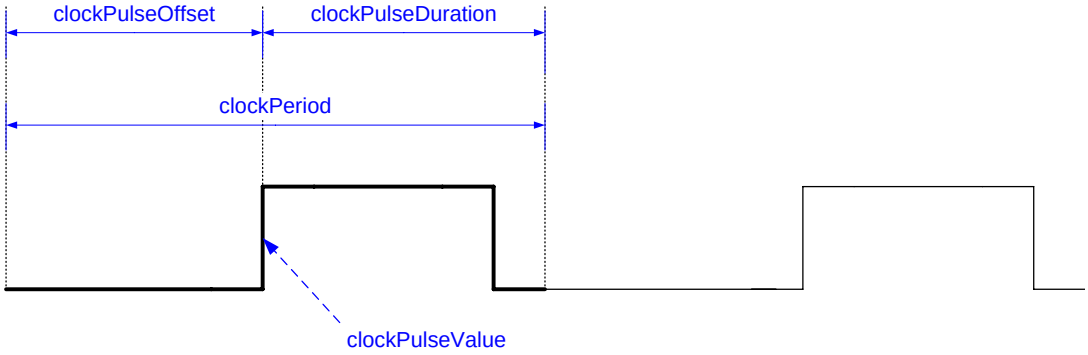


Figure 7.12: ClockDriver elements.

Component driver/SingleShotDriver. This driver is similar to the clock drive but defines non-repetitive waveforms. The durations are specified is the same way as for clock drivers.

The `singleShotDriver` element contains three elements that describe the properties of the waveform (see also figure 7.13).

[5]This leads to a clumsy specification when the pulse offset is 0.

1. `singleShotOffset` (mandatory) specifies the time delay from the start of the waveform to the transition.
2. `singleShotValue` (mandatory) specifies the logic value to which the port transitions. This value is also the opposite of the value from which the waveform starts.
3. `singleShotDuration` (mandatory) specifies how long the waveform remains at the value specified by singleShotValue.

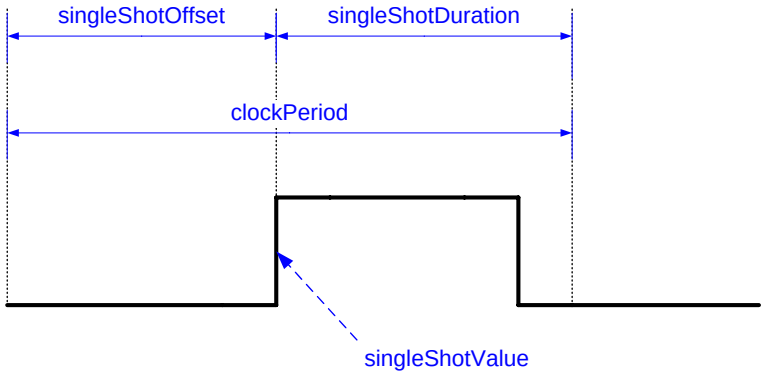


Figure 7.13: SingleShotDriver elements.

Port timing constraints. They appear within wire ports in an abstraction definition. The `timingConstraint` element defines a technology-independent timing constraint associated with the containing wire port of a component or abstraction definition. The values are expressed as a percentage which represents the ratio of the cycle time to be allocated to the timing constraint on the port.

1. `clockEdge` (optional) specifies to which edge of the clock the constraint is relative (either `rise` or `fall`).
2. `delayType` (optional) restricts the constraint to applying to only best-case (minimum) or worst-case (maximum) timing analysis. By default, the constraint is applied to both. The `delayType` attribute may have two values `min` or `max`.
3. `clockName` (mandatory) specifies the reference clock.

An example of IP-Xact timing constraint follows:

```
<spirit:timingConstraint spirit:clockName="hclk">
  50
</spirit:timingConstraint>
<spirit:timingConstraint spirit:clockName="hclk"
  spirit:clockEdge="fall" spirit:delayType="min">
  40
</spirit:timingConstraint>
<spirit:timingConstraint spirit:clockName="hclk"
  spirit:clockEdge="fall" spirit:delayType="max">
  60
</spirit:timingConstraint>
```

Lines 1 to 3 specify that the delay of the contrained element is 50% of the period of clock `hclk`, relative to the rising edge of the clock (default value of attribute `clockEdge`). This values applies to both best and worse case (absence of the attribute `delayType`). Line 4 to 7 specify a delay of 40 % of the clock cycle, relative to the falling edge, and applicable to the best case. The third constraint (lines 8 to 11) applies to the worst case timing.

External specifications

Timer module has already been briefly introduced in the chapter 3 on page 3.4. But that example was more basic and hypothetical. Here we focus on the timer module from the Leon II architecture (presented on page 91). Leon II timer unit implements two 24-bit timers, one 24-bit watchdog and one 10-bit shared prescaler, as shown in the figure 7.14.

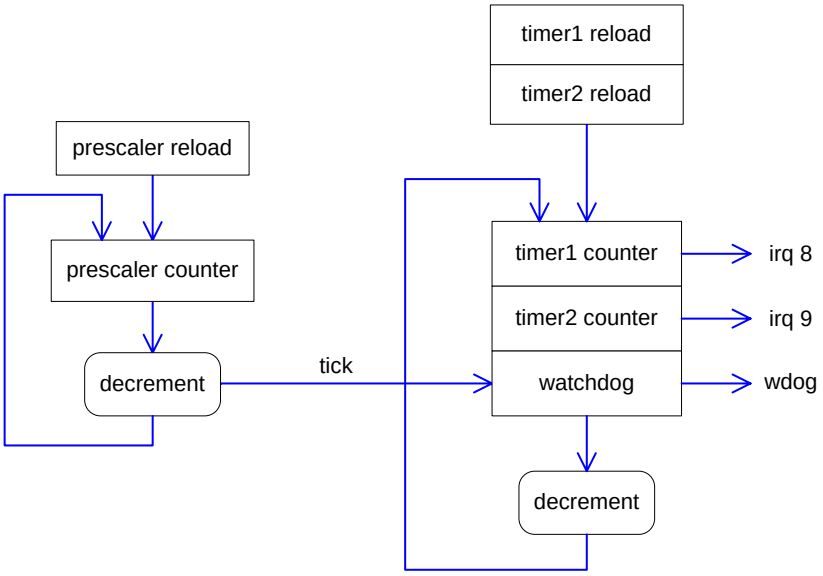


Figure 7.14: Leon II Timer Module Structure

The prescaler module is used to divide the system clock frequency. It produces an output named *tick* that then drives the timer 1, timer 2, and Watchdog modules. The prescaler loads/reloads the count-down value from the prescaler reload register (or can be directly loaded also). This value is then decremented on each system clock event. Once the counter reaches zero, the output tick is produced and the prescaler counter (prescaler value) is reloaded. The effective division rate of the prescaler is equal to the reload register value + 1.

The prescaler module is always running and is independent of the timers 1 and 2 which are controlled through their respective timer control registers. These timer control registers currently consist of only three control bits *enable*, *reload*, and *load*. The *enable* bit enables the timer and it starts decrementing the timer counter on each tick signal coming from the prescaler. If the *reload* bit is enabled, the timer will load the value in the reload register into the timer counter register on the next counter underflow. If reload is disabled, the timer will reload itself with the maximum possible value. The *load* register bit, when set, instantly loads the timer counter with the reload value. This load bit is a kind of write only bit as reading it always returns the value '0'. When the timer counter reaches zero, the respective interrupt is generated. The watchdog counter differs in the way that it does not have a control or the

reload register and is always running. The two timers and the watchdog all share the same prescaler decrement counter's output tick.

This behavior is described in external files reachable through the `fileSet` specification. This is relative to a particular *view*, in the *model* part of the Timer component. Listing 7.2 contains references to VHDL files. Lines 8 to 10 select the language, lines 14 to 18 refer to the source files. Listing 7.3 contains the same kind of information for a SystemC source code.

```
1  <spirit:model>
2    <spirit:views>
3      <spirit:view>
4        <spirit:name>vhdlsource</spirit:name>
5        <spirit:envIdentifier>
6          :designcompiler.synopsys.com:
7        </spirit:envIdentifier>
8        <spirit:language spirit:strict="true">
9          vhdl
10       </spirit:language>
11       <spirit:modelName>
12         leon2Timers(struct)
13       </spirit:modelName>
14       <spirit:fileSetRef>
15         <spirit:localName>
16           fs-vhdlSource
17         </spirit:localName>
18       </spirit:fileSetRef>
19     </spirit:view>
20   </spirit:views>
21    ...
22 </spirit:model>
```

Table 7.2: External references to VHDL files.

```
1  <spirit:model>
2    <spirit:views>
3      <spirit:view>
4        <spirit:name>TLM_PV</spirit:name>
5        <spirit:envIdentifier>
6          :*Simulation:
7        </spirit:envIdentifier>
8        <spirit:language>systemc</spirit:language>
9        <spirit:modelName>timers</spirit:modelName>
10       <spirit:fileSetRef>
11         <spirit:localName>
12           sourceCode
13         </spirit:localName>
14       </spirit:fileSetRef>
15     </spirit:view>
16   </spirit:views>
17  ...
18 </spirit:model>
```

Table 7.3: External references to SystemC files.

7.4.2 Behavior triggering

Fields

IP-Xact defines a *field* as an array of consecutive bits included in a *register* (see our Component Memory Metamodel, in Figure 5.8 on page 70). A unique identifier may be assigned to a field making references easier. Other elements are used to characterize a field. We detail three of them that are linked to behavioral aspects.

access is an optional element which indicates the accessibility of the field and the existence of possible side-effect while reading or writing. Possible values for this element are given in the following table.

Value	Effect
`read-only`	returns a value related to the values in the field
`write-only`	affects the contents of the field
`read-write`	combines both previous effects
`writeOnce`	only the first writing after power up may affect the contents of the field
`read-writeOnce`	like the previous with reading capability

modifiedWriteValue is another optional element which describes the manipulation of data written to a field. This is typical of (hardware) control registers in which writing a value often results in a different value. When this element is omitted, the value written to the field is the value stored in the field.

Value	Effect
`oneToClear`	writing a 1 clears the corresponding bit
`oneToSet`	writing a 1 sets the corresponding bit
`oneToToggle`	writing a 1 toggles the corresponding bit
`zeroToClear`	writing a 0 clears the corresponding bit
`zeroToSet`	writing a 0 sets the corresponding bit
`zeroToToggle`	writing a 0 toggles the corresponding bit
`clear`	all the bits are cleared
`set`	all the bits are set
`modify`	all the bits may be modified by the writing

readAction is an optional element which describes the effects of a read operation on the bits of the field. When this element is omitted, the field is not modifed by a read operation.

Value	Effect
`clear`	the field is cleared after a read operation
`set`	the field is set after a read operation
`modify`	the field is modified in some way after a read operation

Besides possible bit changes, a field access operation can trigger specific actions. This is illustrated with Leon II timers.

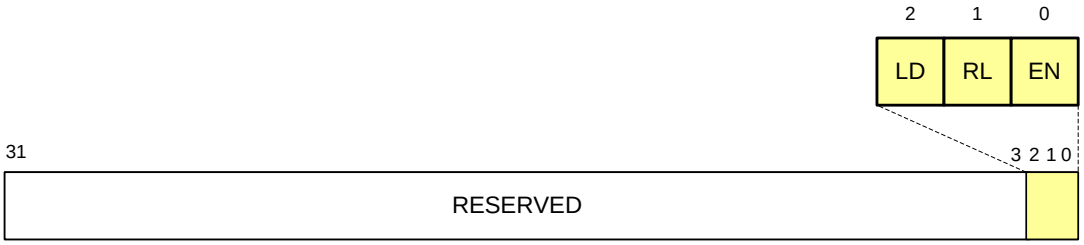


Figure 7.15: Timer Control Register.

Example of the Leon II timers

Figure 7.15 represents the structure of a Timer Control Register. This structure is also described in the XML listing. Lines 9 to 18 specify the field named enable (line 10) and denoted as EN in the figure. Line 14 indicates the offset of the field whithin the register (offset = 0). The width of the field is one bit (line 15). This field can be accessed for both read and write operations (line 17). A description (lines 11–13) gives the purpose of this field. This is an informal textual specification. The other fields of this register are described in a similar way. Note that the loadCounter field, denoted as LD in the figure, does not store what it is given as argument for a write operation. Lines 40–42 make clear that when a '1' is written in LD, its contents (a bit) is cleared.

```
1  <spirit:register>
2    <spirit:name>timerControl</spirit:name>
3    <spirit:displayName>timerControl</spirit:displayName>
4    <spirit:description>Timer Control Register</spirit:description>
5    <spirit:addressOffset>0x8</spirit:addressOffset>
6    <spirit:size>32</spirit:size>
7    <spirit:access>read-write</spirit:access>
8    ...
9    <spirit:field>
10     <spirit:name>enable</spirit:name>
11     <spirit:description>
12       Enables the timer when set.
13     </spirit:description>
14     <spirit:bitOffset>0</spirit:bitOffset>
15     <spirit:bitWidth>1</spirit:bitWidth>
16     <spirit:volatile>true</spirit:volatile>
17     <spirit:access>read-write</spirit:access>
18   </spirit:field>
19   <spirit:field>
20   <spirit:name>reloadCounter</spirit:name>
21     <spirit:description>
22       When set to 1 the counter will automatically
23       be reloaded with the reload value after
24       each underflow.
25     </spirit:description>
26     <spirit:bitOffset>1</spirit:bitOffset>
27     <spirit:bitWidth>1</spirit:bitWidth>
28     <spirit:access>read-write</spirit:access>
29   </spirit:field>
30   <spirit:field>
```

```
31      <spirit:name>loadCounter</spirit:name>
32      <spirit:description>
33        When written with 1, will load the timer reload
34        register into the timer counter register.
35        Always reads as a 0.
36      </spirit:description>
37      <spirit:bitOffset>2</spirit:bitOffset>
38      <spirit:bitWidth>1</spirit:bitWidth>
39      <spirit:access>read-write</spirit:access>
40      <spirit:modifiedWriteValue>
41        oneToClear
42      </spirit:modifiedWriteValue>
43    </spirit:field>
44    <spirit:field>
45      <spirit:name>reserved</spirit:name>
46      <spirit:description>Reserved</spirit:description>
47      <spirit:bitOffset>3</spirit:bitOffset>
48      <spirit:bitWidth>29</spirit:bitWidth>
49      <spirit:access>read-only</spirit:access>
50      <spirit:testable>true</spirit:testable>
51    </spirit:field>
52  </spirit:register>
```

Besides the informal specifications given in the `description` elements, the precise behavior triggered by the write operations in the Timer Control Register is described in external files referenced in the fileSet (see page 129). We propose to go further by linking fields to behaviors.

Extension to the metamodel

We have already used UML and MARTE in our IP-Xact profile. As demonstrated in this section, IP-Xact offers little direct support to behavioral specifications. In constrast, UML has many ways to represent behaviors. Our metamodel of IP-Xact can be easily extended to allow a better connection between structure and behavior. Figure 7.16 contains our unified notion of component: on the left side, the structural aspects covered by IP-Xact; on the right side, the behavioral aspects brought by UML (shown with a hatched background). Writing in fields of control registers may trigger the execution of some behavior, which in turn uses data contained in other fields of registers as input parameters.

Moreover, a UML functional model can be annotated with time information using the MARTE time model. Hence the modified figure 7.17. Considering this metamodel, we propose to extend IP-Xact descriptions with time requirements of IPs. It is neither practical nor desirable to include the whole behavior since it would require adding all UML behavioral model elements to IP-Xact. It is also not practical in most cases since implementations of the same IP at different abstraction levels are usually made by different teams and may result in components that do not even have the same interfaces. For example, a simple read or write communication at TLM level boils down to more complex control signals at RTL level. The same RTL signals can also be shared by completely different transactions. Rather than addressing the whole IP behavior, we focus on their time requirements. These requirements, almost missing in IP-Xact, are usually described as waveforms in datasheets. We specify them as a CCSL specification (see section 7.3). Then, we rely on CCSL operational semantics to execute the specification and automatically produce waveforms. These specifications are then used to generated dedicated *observers* of temporal properties. This is developed in the next chapter.

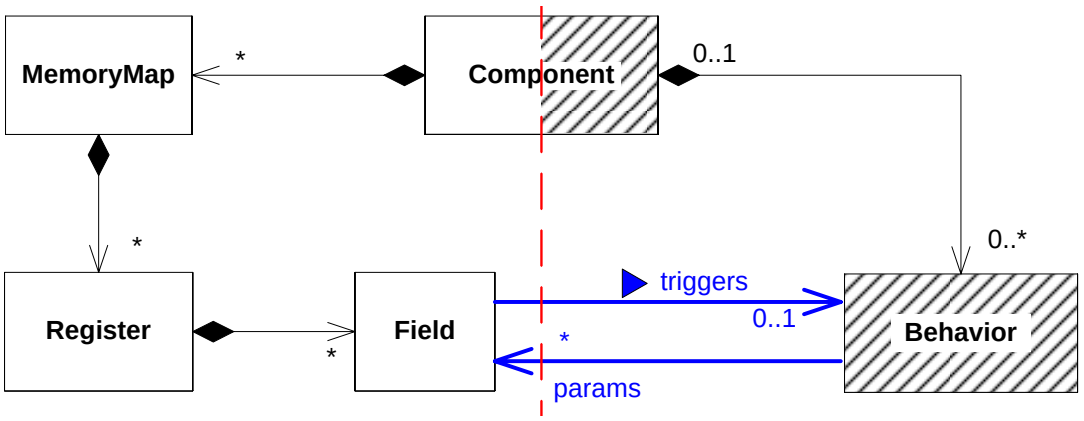


Figure 7.16: Linking structure and behavior.

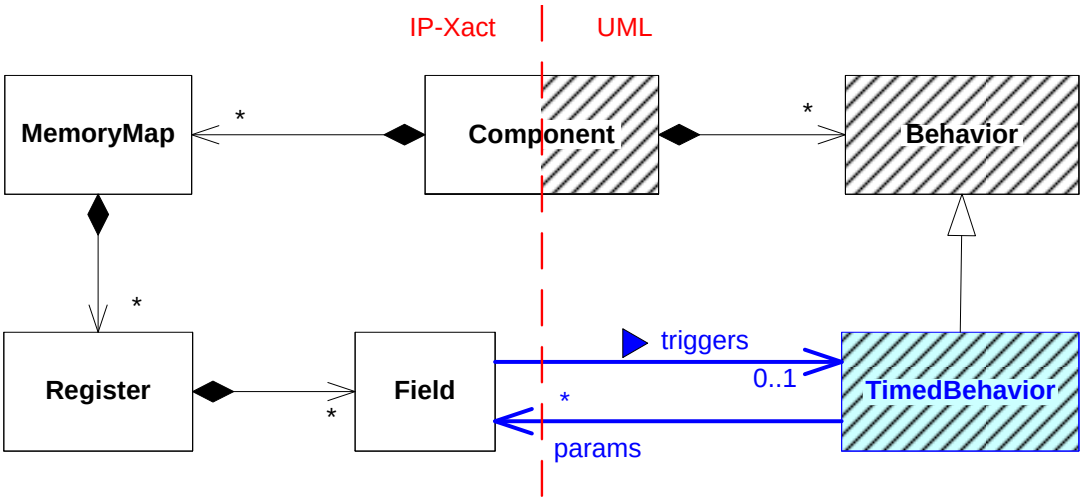


Figure 7.17: Linking structure and behavior.

7.5 Conclusion

In this chapter, we have seen the different implementations of behavior of the IPs at different abstraction levels. Esterel is seen to be the best choice for the abstract behavior representation with the advantage of use of logical clocks. SystemC is also widely used for abstract level modeling but causes the loss of precision while designing at such levels. After describing our UML profile for IP-Xact (in the previous chapters) that builds on MARTE to model IP-Xact designs, here we propose to use the MARTE Time model and its constraint language CCSL to complement IP-Xact specification for selective behavior representation. We also explored the way behavior can be coupled to the structural constructs in the IP-Xact specification. In the next chapter, we use CCSL constructs to practical use by the development of VHDL observers for them and then creating testbenches from those observers.

Chapter 8

Verification and Testbenches

Contents

This chapter continues the discussion on the concepts introduced in the previous chapter regarding behavior representation. We consider different approaches for verification techniques and testbench development. Then we introduce a verification approach relying on CCSL *clock constraints and observers. A library of* VHDL *observers is proposed. Finally, these observers are put to use for a verification of time properties of a Leon II-based system.*

8.1 Introduction

Verification is a process used to demonstrate the functional correctness of a design [Ber02]. It is an act of reviewing and testing, and hence determining and documenting whether the design output meets design input requirements. In the present times, the system verification takes almost 70% time of the total design efforts necessitating the use of new techniques to reduce verification time and efforts. Designing at higher abstraction levels enables us to work more efficiently without worrying about low-level details. At this level, signal and pin details of the architecture are hidden, hence the designer can better focus on the overall functional correctness of the model. Simulating design models is also very fast at this level. Hence, the verification process at the abstract levels is efficient. But we have to note that at higher levels of abstraction, we have loss of information about the design (like pin/signal details) resulting in reduction of control over the system. Therefore using system abstraction for verification purposes must be chosen wisely as it may lead to wrong functional output.

The purpose of the verification process is to ensure that the result of given transformation is as expected. We use the reconvergence model to know what exactly is being verified. The *reconvergence model* [Ber02] is a conceptual representation of the verification process and is used to illustrate what exactly is being verified. For a verification process, the *design-under-test* (DUT) and the verification process must have the same starting point. The verification process is of several types: formal verification, functional verification, and test bench generation, as defined in the subsequent paragraphs.

The *formal verification* is the act of proving the correctness of a system through its underlying algorithms with respect to a certain formal specification or property. Formal verification can be further divided into two sub-categories: equivalence checking and model checking. The *equivalence checking* compares two models to mathematically prove that the source and the output are logically equivalent and that the transformation preserves its functionality. *Model checking* is the verification technique that looks for the violation of user-defined rules about the behavior of the design. It verifies that the assertions or characteristics of a design are formally proven or not. As an example, it can check a system behavior based on automata by checking all states in a design for unreachable, isolated states, or deadlocks.

The *functional verification* ensures that a design correctly implements intended functionality. The functional verification (a design meets the intent of its specification) can only be performed when the specification itself is written in a formal language with precise semantics. Without functional verification, one must trust that the transformation of a design specification into low level code was performed correctly, without misinterpretation of the specifier's intent. Note that in software engineering, functional verification is usually called *validation* [Boe84], and according to Boehm, answers the question "Am I building the *right* product?", while *verification* answers "Am I building the *product* right?".

In HDLs, *testbench* refers to a code structure used to create a pre-determined input sequence to a design and then to optionally observe the response. Testbenches are the piece of code that can generate stimulus to exercise code or expose bugs [Ber02]. Testbenches are used either to increase code coverage or to simulate the design to check if it violates a property. Figure 8.1 shows how a testbench interacts with a DUT. In this testing structure, the testbench initially provides inputs to the design and monitors any outputs. The design and its testbench together create a closed system where no other external stimulus or observation is introduced from the user of the system or the environment. For the design under test, such a system is the model of the universe (or the external environment). The task of the test and verification process is to consider what input patterns to supply to the design and what shall be the expected output of a properly working design.

Generally, there are three distinct approaches to functionally verify a system: black-box,

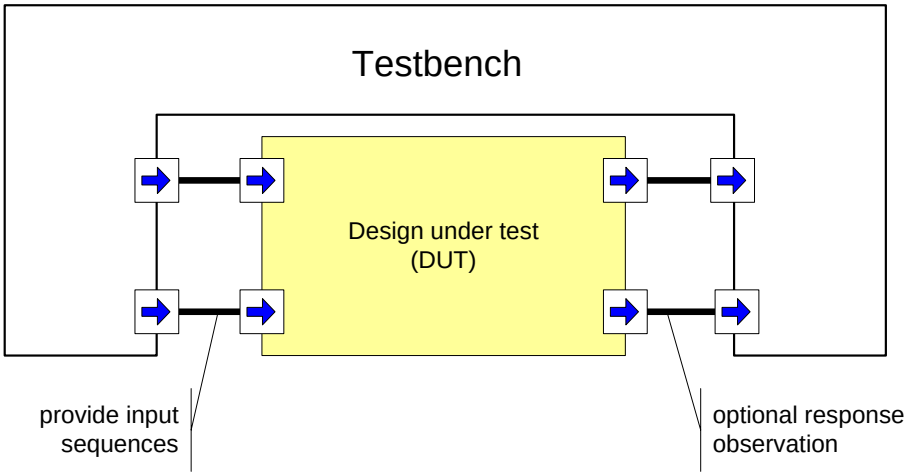


Figure 8.1: Generic Structure of Design Verification.

white-box, and gray-box. The *black-box verification* approach is performed without any knowledge of the actual implementation of a design. Verification is performed by interacting with the available interfaces of the DUT and having no knowledge of the internal structure or behavior implementation. This drawback leads to difficulties in interacting with very large or complex designs. In such complex designs, we then need additional non-functional modifications to provide extended visibility and controllability of the system. Such non-functional modifications can be like adding few testbench accessible internal registers which can then be used to trace the internal state of the design modules. On the positive side, black-box approach does not depend on any specific implementation of the design and hence provides portability.

The *white-box verification* approach has complete visibility and control of the internal structure and behavior implementation of the DUT. Due to such a knowledge of the system, this approach can easily isolate the different modules of the design and can correctly diagnose the problems related to a particular sub-module. White-box verification approach strictly depends on the implementation of design which hinders portability. The *gray-box verification* approach is a mix of black-box and white-box approaches. Just like black-box approach, gray-box approach controls and observes a design through its interfaces but it contains significant knowledge about the system just like in white-box approach.

In the previous chapter, we have discussed in detail that how useful CCSL is in representation of timed behavior patterns. We have also given its practical application on our running example of *Acquisition Protocol*. That example is tested using the TimeSquare tool in Eclipse for simulating CCSL constraints. In this chapter, we have used the black-box approach for our CCSL based constraint observers. This means that our observer modules are only concerned with the interface of DUT and does not need to know the internal structure specification of the IP. This internal structure may or may not be known to the design verifier. Little knowledge of component's internal structure may help to rapidly/easily diagnose the problems indicated in terms of specific constraint violation. The advantages of black-box approach are also there for our CCSL observer modules. Black-box approach provides us greater portability meaning that testbenches implemented through such approach do not depend on the target implementation technology. It means that our observers are equally valid for IP modules implemented either at RTL level (VHDL) or abstract levels like TLM (SystemC) and CP (Esterel implementation). This compliance will exist till the interfaces provided by the various IP

implementations remain same. Hence, one of our goals, of selective behavior pattern matching between various implementations of an IP, is achieved.

In this chapter, we firstly discuss the various aspects of property checking and observers in section 8.2. Then in section 8.3, we focus on the creation of observers in Esterel and VHDL. Finally in section 8.4, we use our developed VHDL observers to test a few properties of the Leon II based system.

8.2 Property checking

8.2.1 Principle

As discussed before, comparing implementations at different levels is very difficult. So instead, we have investigated the possibility to generate a skeleton from the abstract specification of the behavior. However, this is not practical because of huge libraries of legacy IPs and because it would impose one single methodology and design flow to all IP providers. As an example we considered the IP implementations in VHDL and SystemC provided by the Spirit Consortium in the IP-Xact 1.4 specification. We noted that as the IPs at TLM and RTL level are implemented by different teams/group of people, they differ a lot in their behavior implementation and structure. Moreover, IPs at abstract levels contain less information as compared to their counterparts at RTL level. For instance, at RTL level we have a complex set of signals which are replaced by bus/channel structures at TLM level implementations. Hence, practically it is quite impossible to compare the behavior as a whole of the two implementations of the same IP. Moreover, most of the IPs are implemented using black box approach and thus their codes can not be compared directly.

From the behavioral point of view, instead of considering the IP behavior as a whole we decide to focus on specific *time properties* of the IP. We can then prove these properties to be equivalent for the diverse IP implementations. These properties shall be related to time like delay between the input and output, alternating input patterns, or one input always precedes another. Such an information can be directly deduced from the specification of the IP. Traditionally IPs datasheets always contains a separate section for the timing diagrams and state machines, describing a test case to show the response of the IP under a specific environment. One such waveform for the APB bridge IP taken from the AMBA specification is shown in figure 8.16 on page 155. We will discuss this waveform in the last section of the chapter and will show that how this waveform was helpful to define the timing properties that are verified later on the actual IPs. This approach is quite non-intrusive as we do not need to know the internal structure of the IP. All the IP vendors provide timing specifications or datasheets to their third party customers. These IP timing properties finally help us to create CCSL constraints, as shown in figure 8.2. CCSL constraints creation depends on the skills and knowledge of the system verifier. But still this step is not too much difficult as CCSL constraint constructs are quite basic and we can combine multiple constraints to make a meaningful property specifications. While using CCSL constraints two points are to be kept in focus. Firstly, the design verifier needs to have some understanding of CCSL constructs and their functioning to avoid errors while utilizing those constraints. Secondly, the CCSL constraints need to have same input interface for all IP implementations (as marked red in the figure), no matter at TLM or RTL level. For this purpose, we use *transactors* with the IPs to convert the abstraction-level specific interfaces to the ones that are required by our observers. For example at the TLM level, the bus communication takes place in terms of transactions between the ports while the observers require individual signals for their functioning. Hence the role of a transactor over here is to get the signal information from a data transaction. These

signals are then converted into special signals representing CCSL clocks using the *adaptors*.

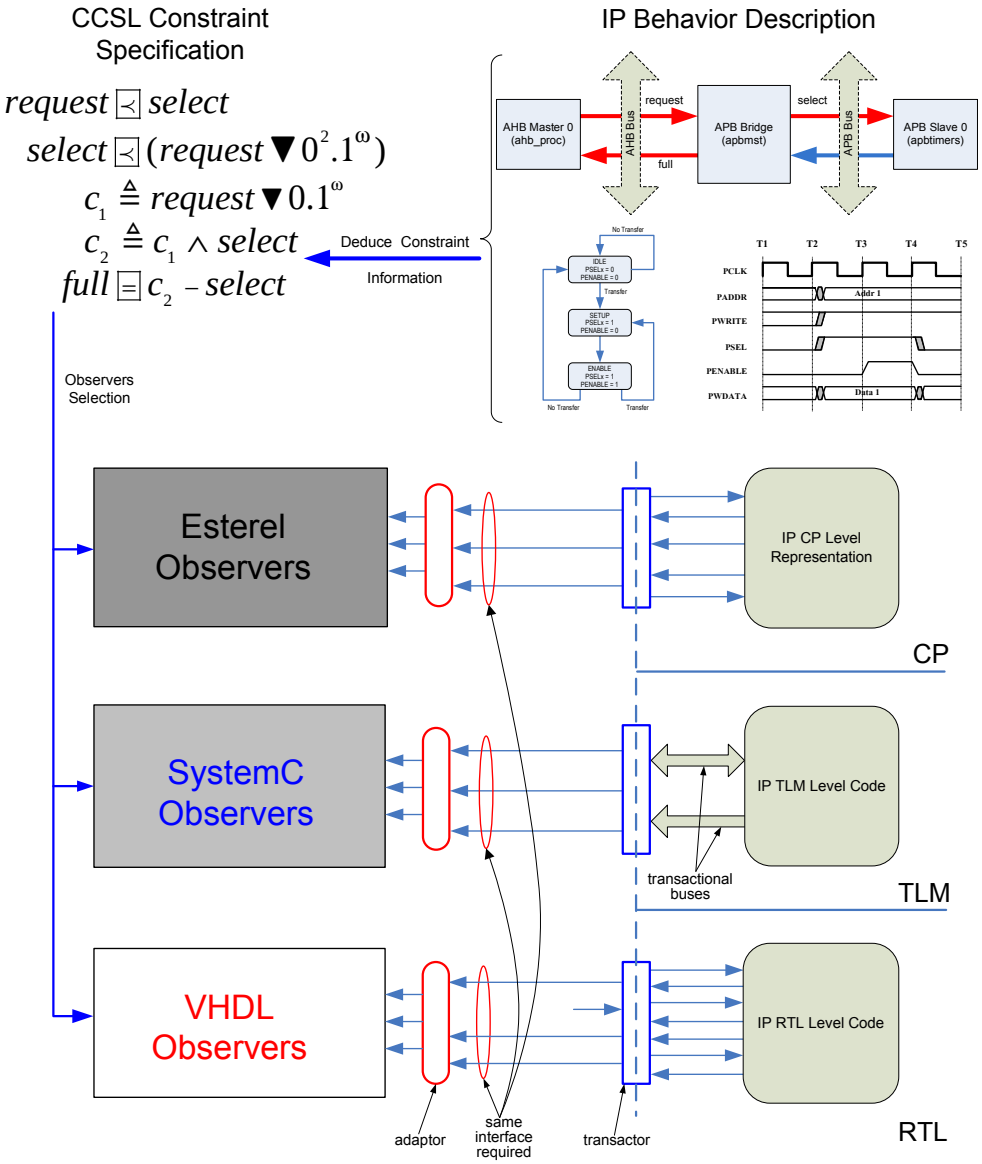


Figure 8.2: Principle of property checking.

Once the CCSL constraints are developed we can simulate them in TimeSquare compiler (plug-in for eclipse) developed by the AOSTE team of INRIA Sophia Antipolis. In this software we provide sample values for the inputs, taken directly from the specification. Once we are satisfied with the selection and the proper representation of timing properties, we proceed to the next phase. Next phase is to develop the testbenches for the IPs under test. These testbenches consist of collection of CCSL constraint implementations called as *observers*. In the further sections we discuss observers and their implementations in detail. These observers are modular objects where each one of them represents a CCSL constraint independently. Moreover, these observers are untimed modules needing no external clock. They react to the input signals

only. Inside the testbench these observers are instantiated along with other helper modules.

Esterel language implementation of observers is already available [AM09b, And09a]. We can use Esterel Studio to generate SystemC observers for the TLM level and VHDL observers for the RTL level IPs from the Esterel observers. The generated code for the testbenches is run against the respective IP implementations. In this way we are sure that the same observer implementation is used to verify time properties at different abstraction levels. Use of Esterel language has got an extra advantage that using the Esterel Studio verification tools, we can *formally verify* the IPs and their corresponding testbenches. Using Esterel Studio, the verification at the CP level is exhaustive so we can guarantee that the CP model will never violate a checked property.

Alternatively we can use the direct implementation of observers being of the same abstraction level as are the IPs under test. We have created such observers for RTL implementation IPs in VHDL. This library allows us to directly check candidate implementations without relying on Esterel Studio code generation tools. These observers do not guarantee that the two implementations of an IP (RTL/TLM) are correct, but it does verify (in simulation) that the timing properties simulated on the CCSL compiler still hold for RTL and TLM implementations. Contrary to Esterel, with lower level implementations the simulation will not cover 100% of the state space (*i.e.*,no model checking).

8.2.2 Observers

Verification by observers is a technique widely applied to synchronous languages [HLR94]. The principle is given in figure 8.3. An *observer* (right-hand box in the figure) is a reactive program expressing a safety property P that has to be verified for a program (middle box). In synchronous language, the observer is put in (synchronous) parallel with the program. The observer has a unique output that signals possible violations of P. The observer receives the same input signals as the program. It also receives its output signals. Thus, the observer is purely passive: it only listen to the program without interfering with it. Often, a property holds only under some contexts. The *assumptions* made on the system environment are represented by another reactive program called *Environment* (left box in the figure). The Environment only generates useful input sequences.

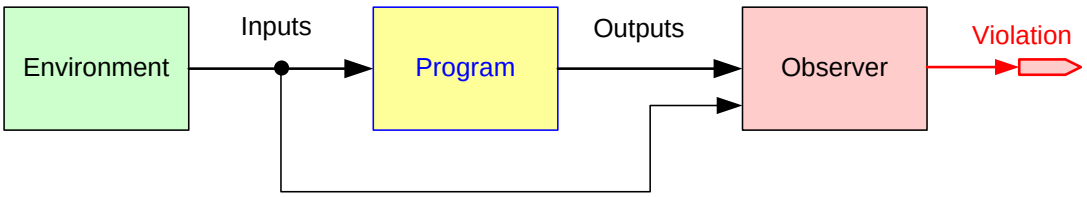


Figure 8.3: Property checking of reactive programs.

The verification consists of checking that the synchronous parallel composition of the three reactive programs Environment, Program, and Observer never emits a violation for any input sequence provided by the Environment. The analysis can be done by standard reachability analysis techniques. If the property is false, an input sequence leading to the violation can be generated. This is called a counter-example.

With the synchronous languages, the observers can be written in the very same language as the program to verify. This is illustrated with Esterel on the Acquisition system in subsection 8.3.1. Non-synchronous languages are also possible targets, this is demonstrated with VHDL in subsection 8.3.2. Before, we give a guide-line for implementing CCSL observers.

Implementing CCSL observers

In order to check properties specified in CCSL with observers, we have first to represent the concept of logical clock in the target language. We have also to represent CCSL clock relations and CCSL clock expressions. For convenience, we propose a modular approach: the observers are built on predefined modules available in a library. The library contains *observer*, *generator*, and *adaptor* modules, which represent clock relations, clock expressions, and clocks, respectively. An observer module signals a `Violation` whenever the associated CCSL relation is violated. A generator module generates a new object which represents the clock defined by the associated CCSL expression. Finally, an adapter module defines the object which represents a CCSL clock. In what follows, we adopt a uniform naming convention for the identifiers: adaptors are prefixed by `Ccsl_A_`, observers by `Ccsl_R_`, generators by `Ccsl_E_`, and clock representations by `c_`.

8.3 Observer implementation

8.3.1 Esterel observers of the Acquisition system

For Esterel, a CCSL clock is represented as a *pure signal*. A library of CCSL observers, generators, and adaptors is available for this language [And09a]. We briefly present the modules effectively used in the Acquisition application.

Adaptors

In Esterel a logical clock can be associated with any signal. This clock ticks whenever the signal is present. Hence, an Esterel adaptor maps an Esterel signal to a pure Esterel signal, standing for a CCSL clock.

For a pure signal `A`, the adaptor code is obvious:

```
module Ccsl_A_pure:
input A;
output c_A;
  sustain c_A if A
end module
```

For a valued signal, the adaptor code is also very simple. It is a generic module which considers only the presence status of the input signal.

```
module Ccsl_A_valued:
generic type T;
input A:T;
output c_A;
  sustain c_A if A
end module
```

Since the presence status of a signal is not persistent, hand-shake in Esterel is often implemented as

```
abort
  sustain resquest
when response
```

The actual event in this case is the "rising edge" of signal `request` (*i.e.*,a signal present at the first instant of the sustain). The *RisingEdge* adaptor represents this behavior (Synchart in figure 8.4).

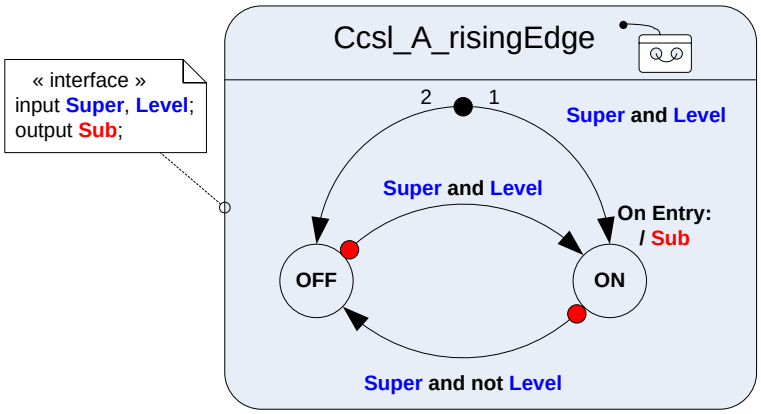


Figure 8.4: SyncChart of the Rising Edge adaptor.

Note that the first two adaptors are so simple that their code is usually in-lined.

CCSL relation observers

Each CCSL relation observer has two input "c_clocks" and an output that signals possible violations. In Esterel, three pure signals are used:

```
interface Ccsl_R_Intf:
    input A,B;
    output Violation;
end interface
```

The CCSL specification of the Acquisition example (section 7.3.3) contains three CCSL relations: equality, exclusion, and alternation. The code of the first two observers is very simple:

```
1 module Ccsl_R_equal:
2 extends interface Ccsl_R_Intf;
3   sustain Violation if (A xor B)
4 end module
```

`Violation` is emitted (line 3) whenever the presence statuses of `A` and `B` are different.

```
1 module Ccsl_R_exclusive:
2 extends interface Ccsl_R_Intf;
3   sustain Violation if (A and B)
4 end module
```

For the exclusion, `Violation` is emitted (line 3) whenever both `A` and `B` are present.

For the alternation, a SyncCharts specification is simpler than an Esterel code (figure 8.5). Note the priority given to the transitions that detect a violation.

CCSL expressions

In the Acquisition application we use only the union of clocks:

```
1 module Ccsl_E_union:
2 input A,B;
3 output C;
4   sustain C if (A or B)
5 end module
```

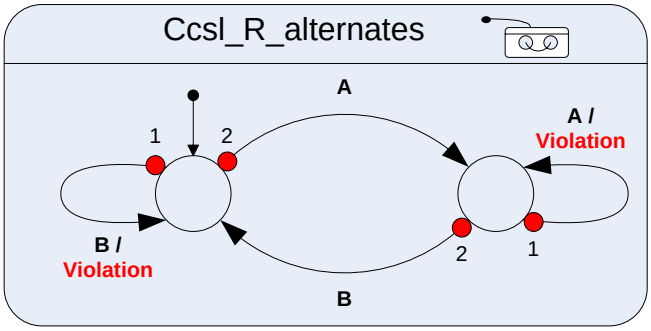


Figure 8.5: SyncChart of the alternation observer.

This module emits `C` (line 4) whenever `A` or `B` is present.

Observation of the Acquisition system

Figure 8.6 shows the observer for the alternation between Breq1 and Grant1 specified in equation 7.23 ($c\_MMP[0].Breq \boxed{\sim} c\_MMP[0].Grant$).

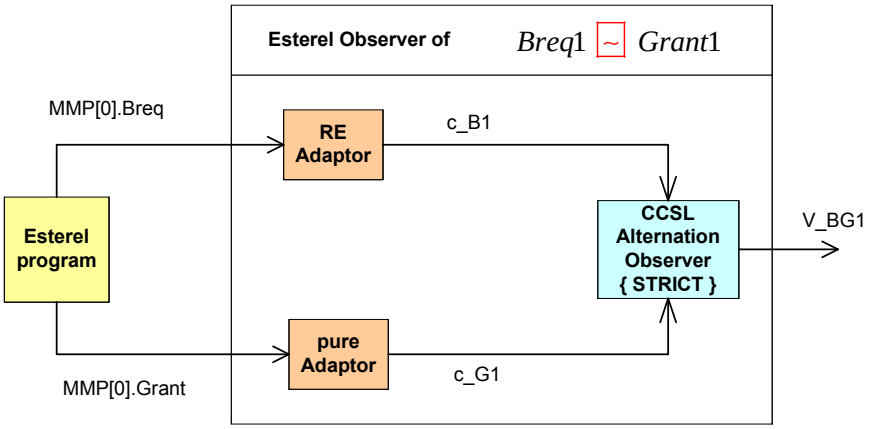


Figure 8.6: Observer for alternation of Breq and Grant.

The observer in figure 8.7 contains a more complex observer that covers a set of CCSL constraints:

- $c\_G \boxed{=} c\_MMP[0].Grant + c\_MMP[1].Grant$ (Eq. 7.17),
- $c\_S \boxed{=} c\_MSP[0].Sel + c\_MSP[1].Sel + c\_MSP[2]$ (Eq. 7.19), and
- $c\_G \boxed{\sim} c\_S$ (Eq. 7.25).

The program and the observers are submitted to the model checking tools provided with Esterel Studio. Figure 8.8 is a commented screen copy of the results of the model checking when applied to the system without an environment module (no left box in figure 8.3). Only two properties are certainly true (the exclusion on the Grants and the exclusion on the Sels). The others are inconclusive. The column message says the reached depth of the exploration

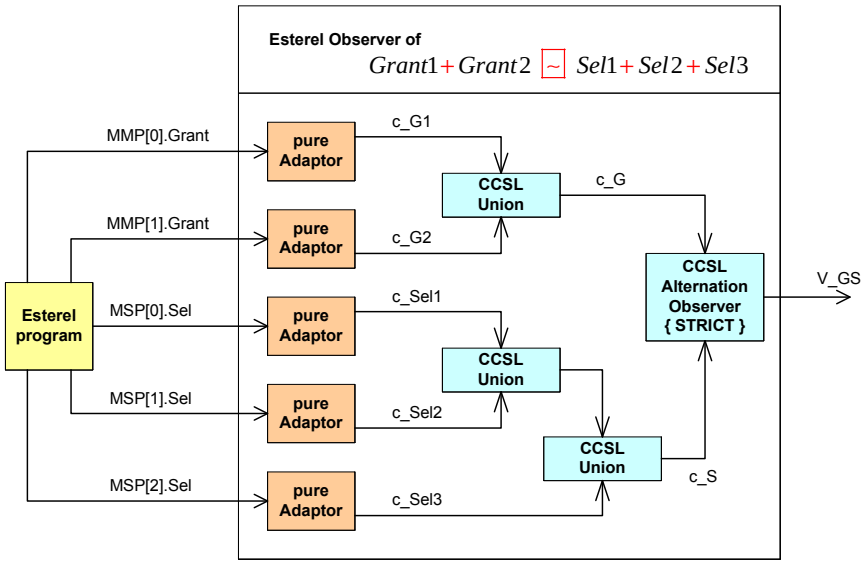


Figure 8.7: Observer for alternation of Grant and Sel.

when the execution has been aborted. The partial conclusion is that there exist no counter-examples for the observed properties, shorter than the depth given in the message. Note that the result for the last property is especially poor.

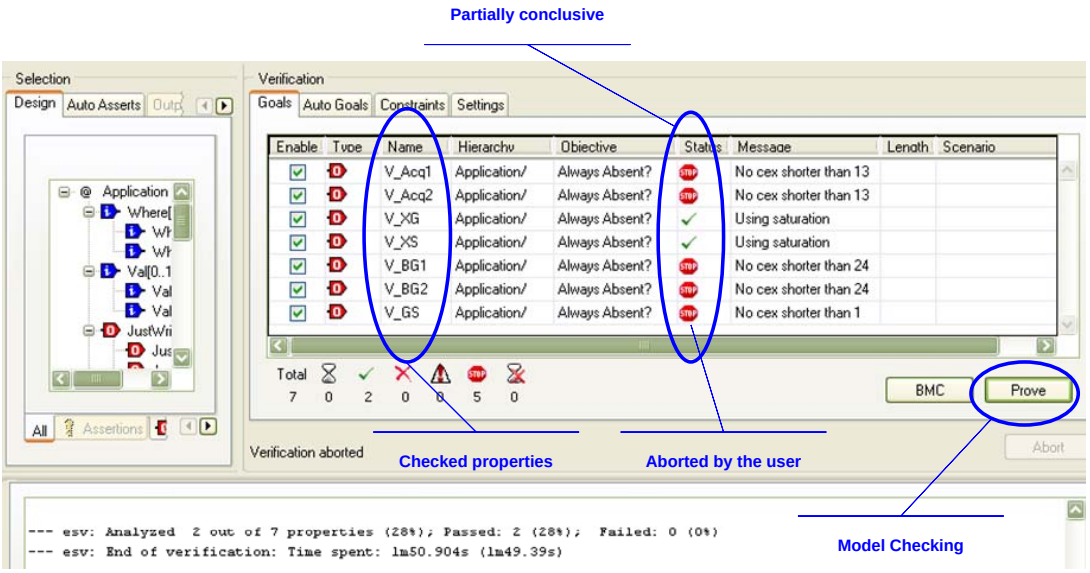


Figure 8.8: Verification using Esterel Studio tools without environment.

In fact, without environment constraints, the model checker, in its exhaustive search, has tried meaningless or at least unrealistic scenarios like making only acquisitions and no savings. Moreover, our programming of the bus gives priority to Breq1 over Breq2, thus there exist

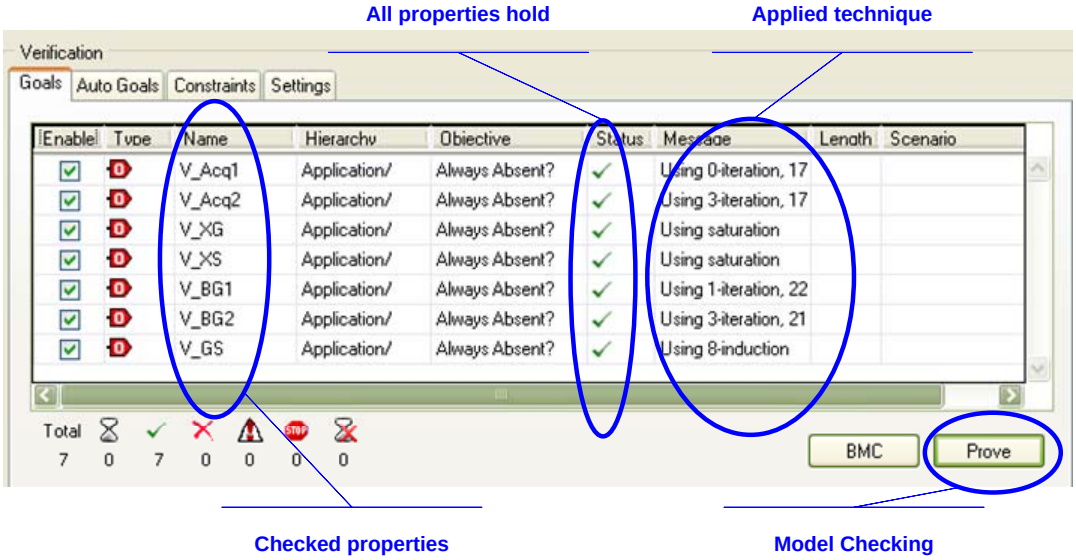


Figure 8.9: Verification using Esterel Studio tools with an environment.

unfair executions (always ignoring Breq2 in the presence of Breq1). To discard these behaviors we have defined an environment module which states that on each processor the requests for acquisition and for saving alternate. Under these assumptions, the model checker has concluded that all the checked properties are true (Figure 8.9).

8.3.2 VHDL observer library

For the practical usage of CCSL constraints in physical embedded devices, we have developed a library of VHDL implementations for CCSL. This library contains observers representing each CCSL constraint and other constructs related to CCSL. This is one of our contributions in the domain of behavior representation using CCSL and its use in property verification.

As discussed in section 8.2.2, CCSL observers, generators, and adaptors represent CCSL relations, expressions, and clocks. In this section, we show the implementation of these modules in VHDL. From all the three types modules, observers are the ones which assess the CCSL clock constraints. All these program implementations are untimed and there is no clock input to these modules.

In this section, we initially define adaptors in the next sub-section. These adaptors are practically shown in use while considering the detailed example in section 8.4. Later in the next sub-section, we discuss VHDL implementations of observers from our CCSL library for VHDL. In the last sub-section, we discuss the VHDL implementation of generators for the CCSL clock expressions.

Adaptors

In VHDL, we represent a logical clock as a pulse-shape signal whose type is `Bit` (figure 8.10). The width of the pulse is as small as possible, but not 0. In VHDL simulation, this means that a "c-clock" pulse has its rising edge and its falling edge in two consecutive simulation instants. The upper waveform in the figure represents an actual VHDL signal `S`. If we consider the event which is characterized by the rising edges of `S`, a CCSL clock S (lower part of the figure) can

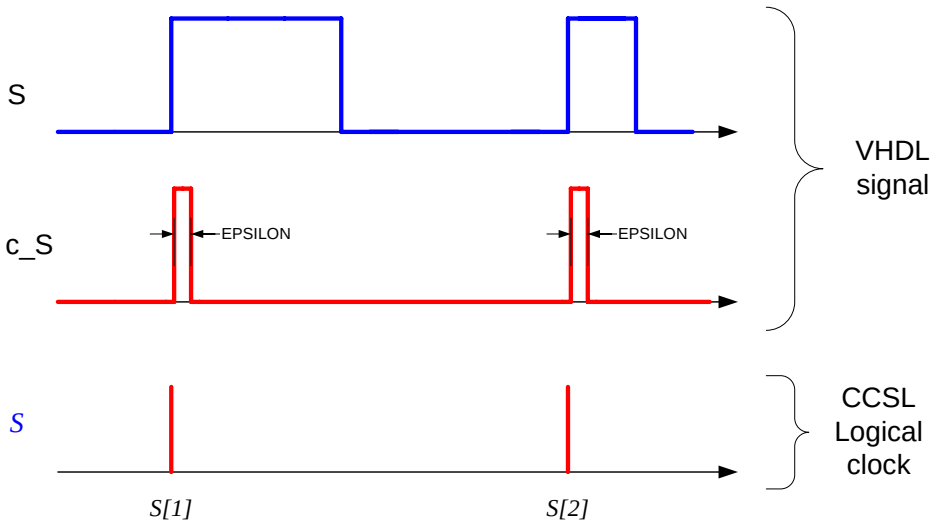


Figure 8.10: Logical clock representation in VHDL.

be associated with this event. This clock is then represented in VHDL as the pulsed signal `c_S` (middle waveform).

The pulse-width of a `c-clock` is `EPSILON`, a constant we have defined in the CCSL package:

```
Package Ccsl is
  constant EPSILON: Time;
end  Ccsl;

Package body Ccsl is
  -- the smallest duration as possible
  constant EPSILON: Time:= 1 fs;
end Ccsl;
```

In all the following VHDL listings, we assume that `EPSILON` is known through a 'use' statement: '**use** Work.Ccsl.**all**;'.

Adaptor `Ccsl_A_risingEdge`, which generates a `c-clock`, is simple VHDL code:

```
1  entity Ccsl_A_risingEdge is
2    port (
3      a: in std_logic;
4      c_a: out bit:= '0');
5  end entity Ccsl_A_risingEdge;
6
7  architecture Ccsl_A_risingEdge_arch of Ccsl_A_risingEdge is
8  begin
9    process
10   begin
11         wait until rising_edge(a);
12         c_a <= '1', '0' after EPSILON;
13   end process;
14 end Ccsl_A_risingEdge_arch;
```

Line 12 generates a pulse of width `EPSILON` whenever the input signal `a` has a rising edge. Note that, according to our hypothesis, `EPSILON` is surely shorter (actually far shorter) than the time during which `a` is high. An adaptor for falling edge is similar, just changing line 11 for '**wait until** falling_edge(a);'.

CCSL Relation Observers

As discussed in the previous chapter, we will define here the observers for the five basic constraint relations including precedence, equality, exclusion, sub-clocking, and alternation. The issue is that the (rising edge of the) `c-clock` signals do not necessarily arrive in the same delta cycle. So, the observer implementations must be *delta delays insensitive*, leading to code more complex than expected at first. To eliminate transient violations (glitch phenomenon due to micro-step semantics and explained in section 7.2.2) we recourse to the VHDL `postponed` statement. This is done in line 9 of the equality observer code below. This instruction is executed only when all the signals are stabilized.

Equality relation :

```
1  entity Ccsl_R_equal is
2    port (
3      c_a,c_b: in bit;
4      v: out bit:= '0');
5  end entity Ccsl_R_equal;
6
7  architecture Ccsl_R_equal_arch of Ccsl_R_equal is
8  begin
9    postponed v <= not(c_a xnor c_b);
10 end architecture Ccsl_R_equal_arch;
```

Observers for exclusion (`Ccsl_R_exclusive`) and for sub-clocking (`Ccsl_R_subclock`) are similar except the code in line 9 which is redefined as
'**postponed** v <= c_a **and** c_b;' for the exclusion, and as
'**postponed** v <= c_a **and not**(c_b);' for the sub-clocking.

Precedence relation :

```
1  entity Ccsl_R_precedes is
2    port (
3      c_a,c_b: in bit;
4      v: out bit:= '0');
5  end entity Ccsl_R_precedes;
6
7  architecture Ccsl_R_precedes_arch of Ccsl_R_precedes is
8    signal d:integer := 0;
9  begin
10   process(c_a,c_b)
11     variable c:integer := 0;
12   begin
13     -- ignore now = 0
14     if now >= EPSILON then
15       if c_a'event and c_a = '1' then
16         c := c + 1;
17       end if;
18       if c_b'event and c_b = '1' then
```

```
19            c := c - 1;
20          end if;
21          d <= c;
22        end if;
23      end process;
24
25      postponed v <= '1' when d<0 else '0';
26    end architecture Ccsl_R_precedes_arch;
```

To deal with precedence, the module has to manage a signal (`d`, declared at line 8) that counts the difference between ticks of the incoming `c-clocks`. Since the rising edges of `c_a` and `c_b` may occur at the same delta cycle, an auxiliary variable `c` is needed.

Module `Ccsl_R_precedes` implements the observer of the *non strict* form of precedence. The *strict* form is more complex and is given in appendix C.2 on page 187.

Alternation relation This relation is not primitive and can be constructed by combining precedence and filtering. Nevertheless, it is a quite usual relation. Its frequent uses demand an efficient implementation of its observer. Thus we propose a direct programming of the alternation observer.

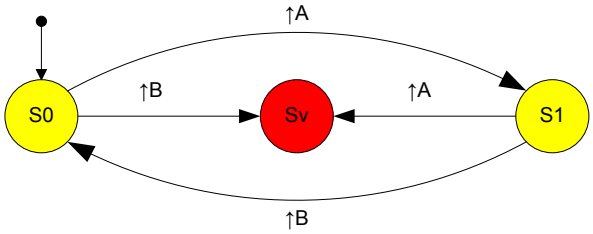


Figure 8.11: Incorrect state machine of the strict alternation observer.

The automaton shown in figure 8.11 is directly derived from the syncChart in figure 8.5. Label $\uparrow X$ denotes a rising edge on X, $\downarrow X$ a falling edge. When implemented in Esterel, there is no problem at all, whereas in VHDL, the behavior is incorrect. The reason is again that $\uparrow A$ and $\uparrow B$ may arrive in any order during delta cycles. For instance, starting from state `S0`, if $\uparrow A$ occurs first, then the next state is `S1`, even if $\uparrow B$ occurs in a next delta cycle. To solve this issue, we propose an automaton (figure 8.12) with transient states (gray background). The steady states (yellow background) are reached only *after* `EPSILON`, when all the delta cycles of the current simulation step are over. This is specified by transitions triggered by $\downarrow A$ or $\downarrow B$.

The code of the `Ccsl_R_s_alternates` observer is available in the appendix on page 189.

CCSL Expressions

Generators also face the issue of possible non-simultaneous occurrences of `c-clock` rising edges. It is even worse because a generator module may produce `c-clock` pulses for other generators or observers, so that the '**postponed**' statement cannot be used.

Inf expression: The generator code relies on three signals (lines 8–10) that count the ticks of the two input `c-clocks` and the output `c-clock`. Knowing the contents of these counters is sufficient to decide whether the output `c-clock` must tick or not.

```
1  entity Ccsl_E_inf is
```

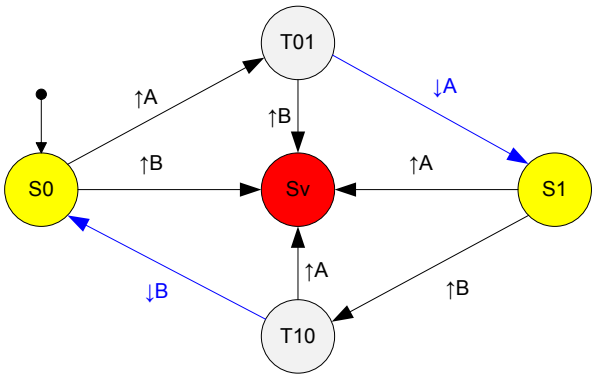


Figure 8.12: Correct state machine of the strict alternation observer.

```
2     port (
3       c_a, c_b: in bit;
4       c_c: out bit:= '0');
5   end entity Ccsl_E_inf;
6
7   architecture Ccsl_E_inf_arch of Ccsl_E_inf is
8     signal cnta: integer := 0;
9     signal cntb: integer := 0;
10    signal cntc: integer := 0;
11    signal do_c: bit := '0';
12  begin
13    process (c_a, c_b, do_c)
14    begin
15      -- ignore now = 0
16      if now >= EPSILON then
17        if c_a'event and c_a = '1' then
18          cnta <= cnta +1;
19          if cnta = cntc then
20            do_c <= '1';
21          end if;
22        end if;
23        if c_b'event and c_b = '1' then
24          cntb <= cntb + 1;
25          if cntb = cntc then
26            do_c <= '1';
27          end if;
28        end if;
29        if do_c'event and do_c = '1' then
30          c_c <= '1', '0' after EPSILON;
31          cntc <= cntc + 1;
32          do_c <= '0' after EPSILON;
33        end if;
34      end if;
35    end process;
36  end architecture Ccsl_E_inf_arch;
```

The local signal `do_c` (line 11) is set when the input `c-clock`, which has the greater index, ticks (lines 19–21 and 25–27). On line 30, the output `c-clock` pulse is generated. Line 32 is

just a delayed re-initialization of signal `do_c`.

Filtering expression: the generator has only one input `c-clock` named `c_sup`, which is the super clock. The output `c-clock` named `c_sub` is a sub-clock of `c_sup`, obtained by filtering according to the periodic binary word specified by two bit vectors: `init` (line 3) for the initial part, and `period` (line 4) for the periodic part. Since there is only one input `c-clock`, there is no risk of critical race. The code is a variant of a shift register. The output pulse is generated in line 14 for the initial part, or in line 19 for the periodic part.

```
1  entity Ccsl_E_filter is
2    generic (
3      init: bit_vector:= "";
4      period: bit_vector:= "1");
5    port (
6      c_sup: in  bit;
7      c_sub: out bit:= '0');
8  end Ccsl_E_filter;
```

```
1  architecture Ccsl_E_filter_arch of Ccsl_E_filter is
2    signal nInit,nPeriod: integer := 0;
3  begin
4    process(c_sup)
5      constant init_size: Natural := init'length;
6      constant period_size: Natural := period'length;
7    begin
8      -- ignore first instant
9      if now >= EPSILON then
10       if c_sup'event and c_sup = '1' then
11         if (nInit < init_size) then
12           -- in initial phase
13           if init(nInit) = '1' then
14             c_sub <= '1', '0' after EPSILON;
15           end if;
16           nInit <= nInit + 1;
17         else  -- in periodic phase
18           if period(nPeriod) = '1' then
19             c_sub <= '1', '0' after EPSILON;
20           end if;
21           nPeriod <= (nPeriod + 1) mod period_size;
22         end if;
23       end if;
24     end if;
25   end process;
26 end Ccsl_E_filter_arch;
```

Minus expression: this CCSL expression has not been defined yet. $c \boxed{=} a - b$ specifies that c is a sub-clock of a, and that c ticks whenever a ticks provided b does not tick at the same instant. The issue with this specification is that it implies a *reaction to absence*. Reaction to absence has a well-defined semantics in synchronous languages like Esterel; it is not the case with most other languages. The Esterel compiler tries to *prove* the absence of a signal during a reaction whereas a classical language compiler determines that a signal is absent only when the reaction is done. This is typically a causality problem.

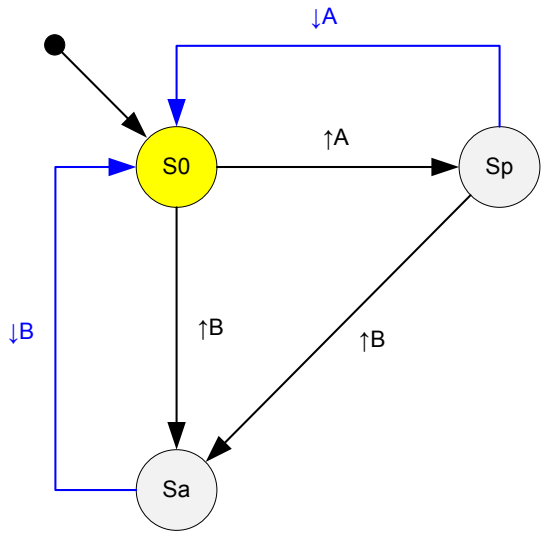


Figure 8.13: State machine of the minus generator.

The automaton shown in figure 8.13 is an attempt to represent the behavior of the minus operator. Unfortunately in VHDL, we have no way to ensure *before the end* of the simulation step that state `Sp` is the current state. The code given below makes the choice to generate a pulse on the output `c-clock` during the transition from state `S0` to state `Sp` (line 20), even if it means that this output can be reset later on (line 31). Thus the `minus` generator can generate a glitch. This is one of the limitations we analyze in the next sub-section.

```
1  entity Ccsl_E_minus is
2    port (
3      c_a,c_b: in bit;
4      c_c: out bit:= '0');
5  end entity Ccsl_E_minus;
6
7  architecture Ccsl_E_minus_arch of Ccsl_E_minus is
8    type State_t is (S0,Sa,Sp);
9    signal State: State_t := S0;
10 begin
11   process (c_a,c_b)
12   begin
13     case State is
14       when S0 =>
15         if c_b'event and c_b = '1' then
16           State <= Sa; -- absent
17         elsif c_a'event and c_a = '1' then
18           -- possibly present
19           State <= Sp;
20           c_c <= '1', '0' after EPSILON;
21         end if;
22
23       when Sa =>
24         if c_b'event and c_b = '0' then
25           -- next simulation instant
```

```
26              State <= S0;
27            end if;
28          when Sp =>
29            if c_b'event and c_b = '1' then
30              State <= Sa;
31              c_c <= '0';
32            elsif c_a'event and c_a = '0' then
33              -- next simulation instant
34              State <= S0;
35            end if;
36
37        end case;
38      end process;
39
40    end architecture Ccsl_E_minus_arch;
```

Limitation of the VHDL observer approach

As just seen before, generators involving reaction to absence may cause glitches and therefore lead to erroneous conclusions in verification with observers. Generator `minus` is the sole module in the VHDL observer library that exhibits such a behavior.

When the `minus` module is directly connected to an observer, it causes no harm because the observer takes account of the stabilized signal values, which are the values at the end of the simulation step. On the other hand, a `minus` module feeding other generators may cause issues by sending erroneous clock pulses. A solution consists in adding delta cycles on input `c_a`, so that a possible rising edge of `c_b` occurs before a rising edge of `c_a`. The number of delta cycles to add can be determined by an analysis of the paths leading to the inputs of the `minus` module. Nevertheless this solution supposes that the observer signals (input to the adaptors) are free of glitches, which can be checked with a *glitch detector* (figure 8.14).

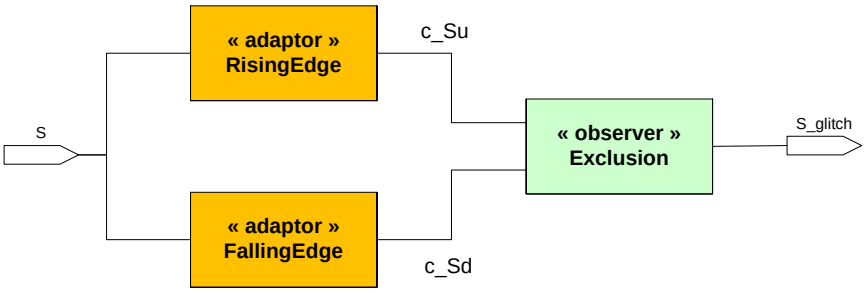


Figure 8.14: A glitch detector.

When observed signals are not glitch-free, we propose an ultimate solution whose drawback is to rely on an external language, namely Esterel. The principle is to program the whole observation chain (generators, observers, and their inteconnections) in Esterel and then to generate a semantically equivalent *sequential code* in VHDL. The Esterel compiler supports this kind of generation. Being a purely sequential code there is no possible critical races. The generated sequential code is put in a VHDL process. To make this solution operational, the execution scheme has to be slightly modified. The verification will now need two phases (each of duration `EPSILON`). The simulation step of the VHDL code under verification is executed first at instant t. The execution of the compiled observation chain takes place at $t + \epsilon$. Auxiliary

signals are reset at $t + 2\epsilon$. The adaptors have also to be modified to adhere to this simulation process. The VHDL code for the rising edge adaptor follows.

```
1  entity Ccsl_A_risingEdge is
2    port (
3      a: in Std_logic;
4      c_a: buffer bit:= '0');
5  end Ccsl_A_risingEdge;
6
7  architecture Ccsl_A_risingEdge_arch of Ccsl_A_risingEdge is
8    type State_t is (S0,S1);
9    signal State: State_t := S0;
10 begin
11   process(a, c_a)
12   begin
13     case State is
14       when S0 =>
15         if a'event and a = '1' then
16           State <= S1;
17         end if;
18       when S1 =>
19         if a'event and a = '0' then
20           State <= S0;
21         elsif c_a'event and c_a = '1' then
22           -- this occurs in the verification phase
23           State <= S0; -- re-initialization
24         end if;
25     end case;
26   end process;
27
28   postponed c_a <= '0' when State = S0 else
29     '1' after EPSILON;
30 end architecture Ccsl_A_risingEdge_arch;
```

8.4 Observers Example

In the previous section, we have discussed various types of VHDL observers for CCSL. We applied the Esterel based observers to our running example of *Acquisition System*. However, one may argue that such demonstrations are usually done on basic/toy examples and do not effectively prove the importance of the concept introduced. To satisfy this purpose, we explain here the use of VHDL observers on some more realistic components. We have chosen the Leon II computer architecture described before in section 6.4.1 on page 91. Leon II is an adequate choice for this demonstration as being one the few free industrial standard IPs available to the research community. Moreover, Leon II is also used with the IP-Xact specification we discussed earlier. It consists of components like processor, memory, bus arbiter, bus bridge, UARTs, and timers. We have tested our property checking methods using observers on the AHB to APB bridge and the AMBA AHB bus arbiter.

In this section, we will focus on the AHB to APB bridge. The discussion ahead gives an informal description of the bridge and later on introduces the use of CCSL observers to verify the properties of our IP as per defined by the AMBA specification. Then, follows a discussion on how to use such a formal specification to compare different implementations of the bridge, possibly at different abstraction levels (TLM, RTL).

8.4.1 APB Bridge Specification

In the Leon II bus architecture, IP components are connected through two AMBA buses. AHB bus is used for high throughput requirements like DMA devices, processor, and memory. APB bus is used for low throughput devices, avoiding the narrow-bus peripherals to load the system bus. In our present work, we have considered the communications between the processor and the timer. The processor is a master on the AHB bus and the communication must go through the bridge to reach the timer attached to the APB bus. Such a communication covers all the aspects of our research interests like functioning of arbiter and bridge (for this section), and functioning of timers (discussed in context of field-address mappings in the previous chapter).

The APB bridge interfaces the AHB to the APB and converts system bus transfers into APB transfers. It buffers address, control, and data from the AHB, drives the APB peripherals and returns data or response signals to the AHB. On a data transfer request, it decodes the address using an internal address map and generates a peripheral select, PSELx. Only one select signal can be active during a transfer. Then the bridge drives the data onto the APB for a write transfer or in case of read transfer it drives the APB data onto the system bus. AMBA specification does not force to implement specific design for APB bridges, and there are many variants of the APB bridge present in the industry. For our research work, the IP implementation from the *Gaisler Research*[1], is designed to operate when the APB and AHB clocks have the same frequency and phase. There are some other design variants also (but we do not have their code) where there are two clock domains in the APB bridge (HCLK and PCLK) for the two buses. This bridge does not perform any alignment of the data, but transfers data from the AHB to the APB for write cycles and from the APB to the AHB for read cycles. For the actual read/write operations, APB bridge generates a timing strobe PENABLE for the read/write clock cycles.

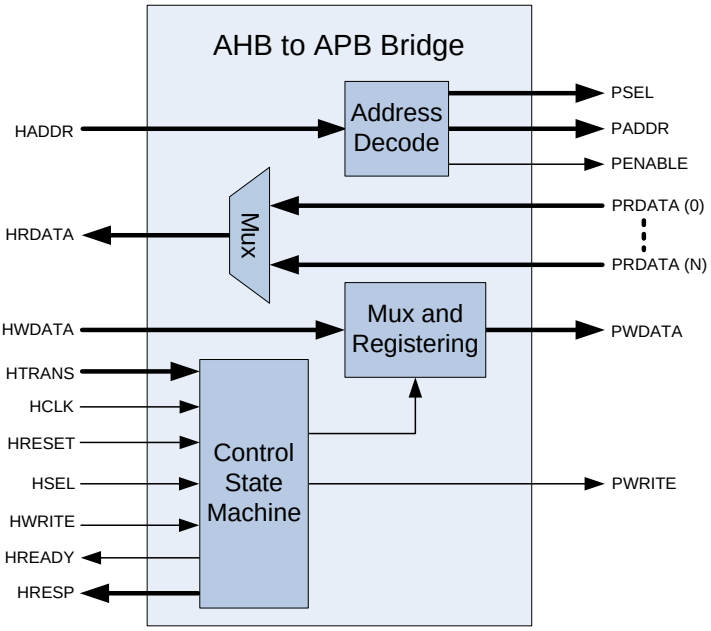


Figure 8.15: APB Bridge Low-level Architecture

[1] www.gaisler.com

Figure 8.15 shows structural description of the bridge. APB bridge acts as a slave for the AHB bus with the AHB side interface signals denoted as Hxxx. For the APB bus, it acts as a master with the interface signals denoted as Pxxx. The initiation of a data transfer starts when the AHB arbiter selects (signal HSEL of) the bridge to indicate a possible data transfer. The bridge then decodes the address available from HADDR and select the corresponding APB slave. There can be as many as 16 APB slaves for the Gaisler's bridge implementation. On write transactions, the APB bridge provides the select (PSEL), enable (PENABLE), write control (PWRITE), address (PADDR), and data (PWDATA) to the targeted peripheral. On read transactions, it multiplexes the targeted peripheral's data (PRDATAx) to the AHB HRDATA. The bridge also returns the signal HREADY back to the AHB master to indicate that it has completed the APB transaction and the slave is ready for the next transaction.

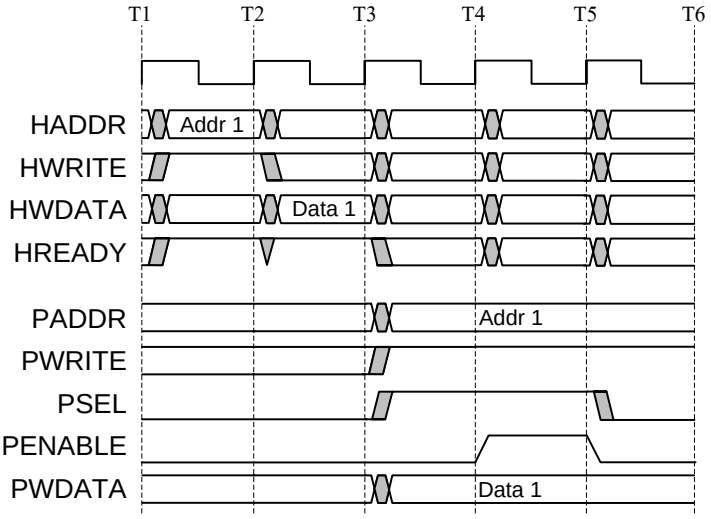


Figure 8.16: APB Bridge Single Write Operation

As per the AMBA Bus Specification [ARM99], the APB bridge implementation must contain *two buffer registers* to store the address (and data) for the next transaction (read or write) while performing the current read/write operation. Figure 8.16 shows a sample *single write* operation as proposed by the AMBA specification. The signal HREADY turns low when the buffer is full, informing the AHB master to pause the transfer. When one operation is performed and one address can be discarded, the signal HREADY goes high again. To illustrate our approach of using CCSL observers for property checking, we have focused on this aspect of the communication.

Gaisler's Leon II example is a running practical example consisting of multiple VHDL modules at different hierarchical levels. It is freely available from the Gaisler's site[2]. The description of the functioning of these modules is given in the Gaisler's IP *cores user's manual* [Aer09]. To understand how to run this project in ModelSim tool (on both Windows and Linux) is explained in the Gaisler's IP *library user's manual* [GHC09]. For most of the modules like that of the processor or RAM, we leave them intact and will focus on the modules of our interest only. The most important module in the whole architecture is the mcore module which groups and instantiates all other important modules. It contains the module representing the AHB to APB bridge called apbmst. Moreover, it contains the module ambacomp (short for

[2]http://www.gaisler.com/cms/index.php?option=com_content&task=view&id=156&Itemid=104

AMBA components) which gathers the definition headers of the components instantiated in the system.

8.4.2 Applying CCSL Constraints

As we know, CCSL constraints can specify selected temporal behaviors of a system. What is the criteria of selection of those properties depends upon the relationship of different signals. This has been explained earlier using the example of *Bus Request* and *Bus Grant* signals which alternate with each other. In this sub-section, first we introduce adaptors for the signals used from the APB bridge. Then, we explain the selection and use of some constraints applied to the APB bridge.

Adaptors

Table 8.1 shows the signals that we use in our constraints. The signal names are complex and also refer to the component instance they belong to. Signal `hsel` that we use for observation is an input signal of the second slave module (`ahbsi(1)`) of AHB bus (which is APB bridge). Similarly, signal `penable` is the output of APB bridge connected to the sixth slave module (`apbi(5)`) of the APB bus (which is timers module). For the ease of use, in the text further, we will use the short form of signals names that we introduced.

Signal Name Used	Actual Design Signal	Description
`clk`	clk	System Clock.
`hsel`	ahbsi(1).hsel	Select probe of the AHB Bridge.
`hselD`		`hsel` delayed for 1 instant.
`hselD2`		`hsel` delayed for 2 instants.
`invhready`	ahbso(1).hready	Ready signal falling edge used.
`penable`	apbi(5).penable	Signal showing transaction cycles on APB.
`penableD2`		`penable` delayed for 2 instants.

Table 8.1: Mapping of signal notation used to the actual design signal.

To convert these VHDL signals into `c-clock`s, we use the rising clock edge policy except for the HREADY signal, which we will explain later. The time duration `EPSILON` of the `c-clock` pulses is preferably 1 femto second, but as the VHDL simulator we use (viz. ModelSim) does not allow such a value, we use 1 nanosecond instead. The instantiations of the adaptor modules are given next. We describe their use, in the next sub-sections.

```
iRising_clk: Ccsl_A_risingEdge
  port map (a=>clk, c_a=>c_clk);

iRising_hsel: Ccsl_A_risingEdge
  port map (a=>ahbsi(1).hsel, c_a=>c_hsel);

iRising_penable: Ccsl_A_risingEdge
  port map (a=>apbi(5).penable, c_a=>c_penable);

iFalling_hready: Ccsl_A_fallingEdge
  port map (a=>ahbso(1).hready, c_a=>c_invhready);
```

Constraint 1

The first CCSL constraint is a simple one and verifies that any APB bridge transaction on the APB bus is always as a result of a transaction command from the AHB bus. Thus, it tests that the signal HSEL always *strictly precedes* the output signal PENABLE. We could not use the PSEL signal as PSEL is unique for each APB slave device. Contrarily, the PENABLE signal is not specific to one slave, and is always functional whenever there is a transaction call (HSEL) on the bridge. We test the property relation of equation 8.1.

$$hsel \boxed{\prec} penable \tag{8.1}$$

To check this property, we input `c-clocks` `c_hsel` and `c_penable` to the precedence observer instantiated in the `mcore` main module of *Leon II Architecture*.

```
cstr11_s_precedes: Ccsl_R_s_precedes
  port map (c_a=>c_hsel, c_b=>c_penable, v=>v1);
```

Where `v1` is a local bit signal defined to trace the violation occurred. Here note that in the equation we use CCSL clocks *hsel* and *penable* while in the VHDL programming, we use the implementation equivalent of those clocks *i.e.,*`c_hsel` and `c_penable` respectively. This nomenclature for CCSL is used through out the presented research efforts and are used reciprocally in the text.

Constraint 2

Second property that we verify is also related to the HSEL and PENABLE signals. It ensures that before the current transaction is completed, at most one new request can be sent by the AHB bus master. This constraint is as per the AMBA specification of APB bridge. It is to ensure that the master module does not overflow the APB bridge with chunk of data. This property is not meant for burst mode transfers, as that would be a violation of this property. Gaisler's Leon 2 Bridge implementation also does not support burst mode for data transfer. Stated in another way, any instant i of PENABLE output signal must always precede the instant $i+2$ of HSEL input signal, as represented in equation 8.2.

$$penable \boxed{\prec} \big(hsel \blacktriangledown 0b00(1)\big) \tag{8.2}$$

This constraint 8.2 can be split into two sub-constraints 8.3 and 8.4.

$$hselD2 \boxed{=} \big(hsel \blacktriangledown 0b00(1)\big) \tag{8.3}$$

$$penable \boxed{\prec} hselD2 \tag{8.4}$$

These two constraints can then be implemented using a *filter* generator and a *precedence* observer module. Here first we created a delayed version of the HSEL signal denoted by a local clock `c_hselD2` using the filter generation function with an initial pattern '00' and the periodic pattern '1'. Later, this delayed version of the original clock is compared with the PENABLE clock for a strict precedence relation.

```
cstr21_filter: Ccsl_E_filter
  generic map (init=>"00", period=>"1")
  port map (c_sup=>c_hsel,  c_sub=>c_hselD2);

cstr22_s_precedes: Ccsl_R_s_precedes
  port map (c_a=>c_penable, c_b=>c_hselD2, v=>v2);
```

Note that the constraint 8.2 is a loose constraint which only checks for the flooding of data caused by the AHB master. This constraint is not effective to check if the buffer size implementation is less than two. This is the case with Gaisler's Leon II implementation which has a smaller internal buffer size but successfully passes this constraint verification. The next constraint gives a more strict version of this property.

Constraint 3

Considering the previous two constraints 8.1 and 8.2 together, they specify the property that

$$\forall i \in \mathbb{N}^\star, hsel[i] \boxed{\prec} penable[i] \boxed{\prec} hsel[i+2] \tag{8.5}$$

This is a typical producer-consumer relationship with a bounded buffer of capacity 2. As discussed before this property is a loosely bound property and a more relevant property is the saturation of the buffer. We would like a signal $full$ to be emitted whenever the buffet gets saturated. This can be expressed as $full = \big(\#hsel = \#penable + 2\big)$, where $\#s$ stands for the number of occurrences of s since the origin of the execution. Unfortunately, there is no direct specification of this behavior in CCSL. We have to use a combination of CCSL expressions and relations. Figure 8.17 represents a possible evolution of the system when only HSEL and PENABLE are observed. All the allowed trajectories (*i.e.*,respecting constraint 8.5) are confined to a strip limited by two parallel lines. Whenever the trajectory touches the lower line, the buffer is full. We have distinguished two points `a` and `b` on a possible trajectory:

Point	`#HSEL`	`#PENABLE`	full
a	4	3	*false*
b	5	3	*true*

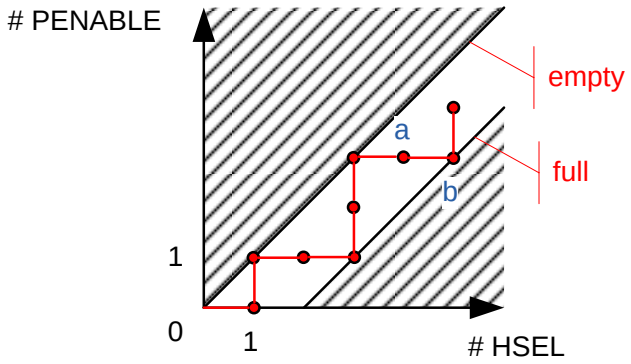


Figure 8.17: A possible evolution.

On this figure, we may see that the buffer gets full whenever $hsel[i+1] \boxed{\prec} penable[i]$. This is the case at point b where the index of `HSEL` is 5, and the 'still to occur' index of `PENABLE` is (or more exactly, will be) 4. Hence, we can reformulate the saturation of the buffer in terms of logical clocks as:

$$\forall j \in \mathbb{N}^\star, \exists k \in \mathbb{N}^\star, \big(hsel[j+1] \prec penable[j]\big) \Rightarrow \big(full[k] \equiv hsel[j+1]\big) \tag{8.6}$$

We create new local `c-clocks`: `hselD` (`hsel` delayed for 1 instant), and `first` (the first to occur between `hselD` and `penable` at the same index). These `c-clocks` are characterized by the following mathematical expressions:

$$\forall k \in \mathbb{N}^\star,\ hselD[k] \equiv hsel[k+1] \tag{8.7}$$

$$\forall k \in \mathbb{N}^\star,\ first[k] \equiv \begin{cases} hselD[k] \text{ if } hselD[k] \preccurlyeq penable[k], \\ penable[k] \text{ otherwise.} \end{cases} \tag{8.8}$$

The CCSL transcription in terms of clock constraints are

$$hselD \ \boxed{=}\ (hsel \blacktriangledown 0b0(1)) \tag{8.9}$$

$$first \ \boxed{=}\ (hselD \wedge penable) \tag{8.10}$$

$$full \ \boxed{=}\ (first - penable) \tag{8.11}$$

Clock constraint 8.11 keeps only the instants of `first` that are not coincident with an instant of `penable`. This reflects the fact that whenever `first` and `penable` tick in coincidence, the difference $\#hsel - \#penable$ is unchanged, and therefore `full` cannot tick.

The equations 8.9, 8.10, and 8.11 can also be termed as the verification of the *internal buffer size* property checking of the APB bridge. Based on the status `full` of the buffer, we can check the HREADY signal of the APB bridge to see if it reflects the internal status properly (*i.e.*,busy only when buffer is full), just as in equation 8.12. For this purpose, we introduce the inverse of `hready` signal (falling edge event of HREADY) in CCSL represented by clock `invhready`. We can then compare this clock `invhready` with the clock `full`. The adaptor for HREADY signal is given before in the sub-section related to adaptors.

$$invhready \ \boxed{=}\ full \tag{8.12}$$

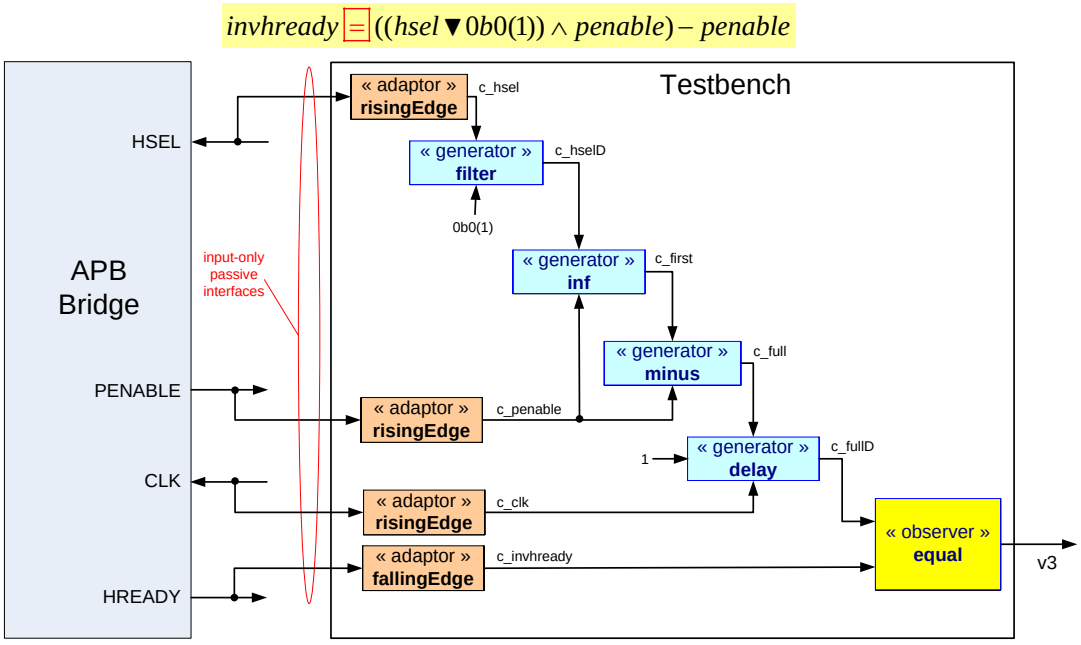


Figure 8.18: Functioning of VHDL Observer based Testbench.

All this working of the testbench based on the observers created in VHDL is shown in figure 8.18. One thing to note regarding all these constraints is that all the data/control clocks are the sub-clock of the system master clock `clk`. This information is omitted in the testbench figure but has to be kept in mind. In the figure 8.18, the filter pattern determines the size of the buffer to be checked. Here the buffer size is considered to be two (filter pattern 0b0(1)). The VHDL implementation of these set of constraints is given next.

```
cstr31_filter: Ccsl_E_filter
  generic map (init=>"0", period=>"1")
  port map (c_sup=>c_hsel,  c_sub=>c_hselD);

cstr32_inf: Ccsl_E_inf
  port map (c_a=>c_hselD, c_b=>c_penable, c_c=>c_first);

cstr33_minus: Ccsl_E_minus
  port map (c_a=>c_first, c_b=>c_penable, c_c=>c_full);

cstr34_delay: Ccsl_E_delay
  generic map (n=>1)
  port map (c_a=>c_full, c_b=>c_clk, c_c=>c_fullD);

cstr35_equal: Ccsl_R_equal
  port map (c_a=>c_fullD, c_b=>c_invhready, v=>v3);
```

Before testing our constraints in the VHDL environment, we simulate it in the Eclipse based CCSL simulator (TimeSquare). One possible result of the simulation of our third constraint is shown in figure 8.19. In this figure, we test for the buffer being of size two. Whenever the clock `full` event occurs, the next event of `hready` is not present. In this way, through simulation, when once we have some confidence in our constraints and applied logic, we move to the VHDL implementation. Such a practice also saves time and effort for us.

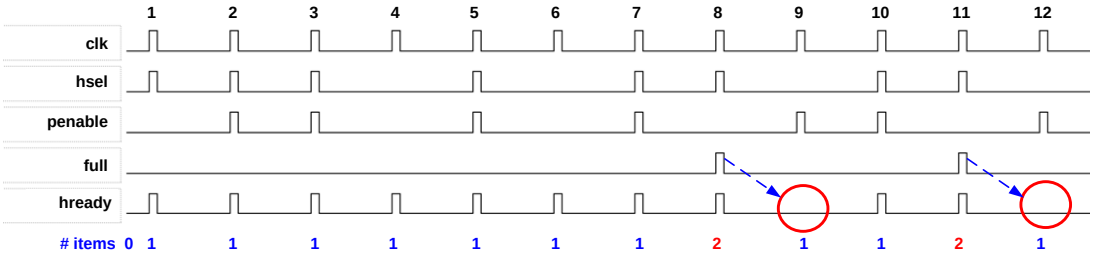


Figure 8.19: Sample Execution of Constraint 3 on CCSL Simulator

For the case of buffer size of one, HSEL has not to be delayed, so equations 8.9 to 8.12 reduces to equation 8.13.

$$invhready \boxed{=} (hsel \wedge penable) - penable \tag{8.13}$$

While working with the Gaisler's Leon II architecture platform, we noticed that the test for equation 8.12 failed for his APB bridge implementation whereas it worked well for the equation 8.13. This showed that he did not complied fully with the AMBA specification regarding the bridge implementation. This IP may function well in their system, but for other systems it can have different results. This anomalous behavior, against the norms specified in the AMBA specification, is shown as 'v3' in the figure 8.20. In this figure, the `invhready` clock shows that

the buffer is full whereas our calculation of buffer (clock `fullD`) does not show any such signs and hence causes a violation (`v3`). When the buffer size '1' is considered, then our calculations match the actual results.

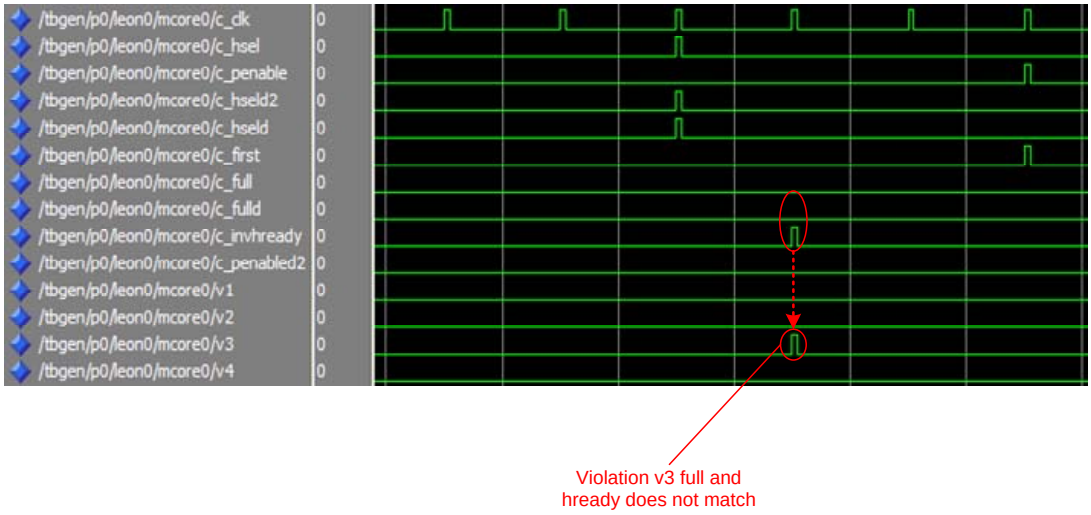


Figure 8.20: Simulation showing Specification Violation of Bridge IP Behavior.

Constraint 4

The first three constraints were interrelated and starting from a basic constraint, we built a complex relation to check the buffer size of APB bridge. Here our last CCSL constraint is relative to PENABLE signal. In APB bus protocol, each single write operation on the APB bus takes at least two clock cycles. The PSEL signal is high for both the write clock cycles, while PENABLE is high for one cycle only. Hence, we can verify this feature that the PENABLE signal can never go high in less than two clock cycles. This is to ensure that the minimum delay of two cycles between the consecutive occurrences of PENABLE is present, as stated in equation 8.14.

$$penable \boxed{\sim} (penable\ (\ 2\ ) \rightsquigarrow clk) \tag{8.14}$$

Here the signal of our interest are CLK and PENABLE. For this property verification, initially a delayed version of PENABLE signal called `c_penableD2` is created. Later on this local signal is tested for alternate occurrences with the PENABLE signal. Hence our preceding constraint can be divided into two sub-constraints of equations 8.15 and 8.16.

$$penableD2 \boxed{=} \left(penable\ (\ 2\ ) \rightsquigarrow clk\right) \tag{8.15}$$

$$penable \boxed{\sim} penableD2 \tag{8.16}$$

Finally the VHDL implementation of these equations is quite straight-forward.

```
cstr41_delay: Ccsl_E_delay
  generic map (n=>2)
  port map (c_a=>c_penable, c_b=>c_clk, c_c=>c_penableD2);

cstr42_s_alternates: Ccsl_R_s_alternates
  port map (c_a=>c_penable, c_b=>c_penableD2, v=>v4);
```

Summary

In this section, we viewed the practical implementation of CCSL constraints through observers created in VHDL. We also visualized the use of these observer constructs to create complex testbenches. Moreover, we also analyzed our results obtained from simulation in TimeSquare CCSL compiler and the ModelSim VHDL. This section provided concrete examples to support our concepts established earlier.

8.5 Conclusion

Early validation of systems built by assembling components requires to associate abstract behavioral and timing models with IPs. In this chapter, CCSL is proposed as an alternative for the representation of time properties at a high abstraction level. The CCSL description acts as a specification of timing requirements for IP-Xact components as waveforms and timing diagrams do in the paper datasheets. We also show how this specification can be used to generate testbenches of valid scenarios that must be satisfied by the component implementations whatever their level of abstraction and the language used is. Generating testbenches for models at various abstraction levels from the same formal specification is an important step towards establishing the equivalence, or at least the consistency of RTL and TLM implementations.

Chapter 9

Conclusion

9.1 Contributions

The objective of this thesis was to understand how the new technology driven engineering models could provide solutions for increased productivity, design integration, and interoperability of system-on-chips, from both structural and functional view points. The first part of this manuscript summarizes the reasoning that motivated our work. Discussions in the chapters 2 and 3 related to ESL design and MDE highlight the need for some *glue approach* between industrial IP implementations and UML-based abstract design approaches. In the later chapters, our contribution focuses on a few main points:

- Definition of the UML profile for IP-Xact that provides a junction between the MARTE profile and the IP-Xact standard for IP reuse, integration and interoperability.
- Application of this profile through MDE techniques, enabling automation of UML models and making possible the integration of a UML design flow with the industrial standard for the structural assimilation of IPs *i.e.,*IP-Xact. Later we show the use of this profile over the practical application of Leon II architecture and demonstrate design automation (model to model transformation) techniques to generate IP-Xact files.
- Proposition of an approach to express and verify time-related functionalities of IPs using CCSL observers.

The benefits of the concepts of carrying IP-Xact in UML are many. Firstly, the UML is more and more used in the industry both for general-purpose and for domain-specific modeling (through the use of dedicated profiles). Thus the integration of UML models with IP-Xact specifications enables the utilization of a wide variety of UML tools, encouraging its exploitation without considering to invest into specialized tools for IP-Xact. The export of UML-based IP components to the IP-Xact design promotes the exchange of components with third party vendors and their better integration into the industrial CAD tools.

UML models are much more adequate for human processing than IP-Xact XML files. Here, UML with our profile defined in chapter 6 allows the use of UML models as input for building IP-Xact specifications. The application of UML in the process of identification and interconnection of components guarantees the compatibility of various components used.

From our case study of Leon II based system, we saw a huge difference between the levels of complexity of the IP-Xact description of those components and that of UML. The information present in the UML models is well organized and graphical. Moreover, this model information

is not scattered as is the case with IP-Xact. The elevation of the level of abstraction for the system design (from IP-Xact to UML) allows the system designers to focus more on the analysis phase of the design-flow. Also, it provides a bridge between the high-level modeling (focusing on concepts) and the low-level modeling (mostly focusing on the implementation issues).

In the framework of a collaboration of the AOSTE team within the CIMPACA *design platform*[1], we had the opportunity to test our work with the industrial tools bought by CIMPACA. Firstly, we used the *coreTools* from *Synopsys* for our experimentation. We imported our Leon II IP-Xact files generated from UML with the *coreAssembler* tool from Synopsys. This tool is used for the integration of IPs and works with both the IPs created with the *coreBuilder* tool and any other third party IP. It works only with the IPs at the RTL level. Our IPs were well imported and integrated in the coreAssembler with other components without any issue. For the TLM level IP implementations, we successfully tested our generated IPs with another Synopsys tool, *Innovator*. With the successful integration with these Synopsys tools, it is easier and much attractive to take advantage of Synopsys *Design-ware System Level Library* (DW-SLL) for modeling systems at abstract levels.

In chapters 7 and 8, we have compared various formalisms to define an abstract timed model for IPs. We started by an enumeration of alternative implementations of our running example at different abstraction levels. The expressivity of Esterel and the use of logical clocks appeared as the best match for the abstract behavior representation. CCSL time model brings polychronous logical clocks into UML (via MARTE) and we then proposed to complement the structural description with a CCSL specification of IP time properties. The CCSL specification is then used to verify that the time properties hold on candidate implementations of IP-Xact components. For this purpose, we have built a library of VHDL observers for each and every CCSL constraint. The use of those observers is illustrated on a VHDL implementation of a Leon II-based architecture. In the same way that paper datasheets come with waveforms or timing diagrams to describe the expected time behavior of IPs, we think that IP-Xact specifications should come with an equivalent representation. Using UML MARTE as a front-end for IP specification allows the use of CCSL for specifying classical physical-time properties (thus replacing waveforms) but also more logical time properties, thus widening the field of properties that can be expressed. CCSL specification being part of the model opens possibilities for various analysis that must be further explored in future works.

9.2 Future works

Using the MARTE profile we have presented our approach to represent IP-Xact models in UML. These models are transformed into IP-Xact using the ATL language. IP-Xact as a whole is a huge standard covering almost every aspect of modeling an ESL design. There is need for constant efforts to improve the transformation process as well as the representation mechanisms in UML. Recently, similar model transformation efforts have been done in the framework of ID-TLM project including the model transformation from IP-Xact to UML models.

For the functional aspects of IP integration, we introduced logical clocks and clock constraints expressed in CCSL. A technique of verification of temporal properties using *observers* was proposed. This work has been done at the RTL level using VHDL. Now for the future, we need RTL and TLM or any other candidate implementation to verify the same set of time constraints expressed by CCSL description. So, implementing observers for the TLM level, preferably in SystemC, may be a short term development task.

Another aspect to explore regarding the CCSL observers is of the role of transactors and their effect on observers. As we discussed before in section 8.2.1 on page 138, for the equivalence

[1] https://ssl.arcsis.org/cimpaca.html

checking of IP implementations, the CCSL observers need the same interfaces. At different abstraction levels (like RTL, TLM) we have different types of interfaces and hence transactors prove to be of vital role. However, it will be necessary to clarify the role of transactors on the use of CCSL observers and their effect on the time properties of an IP.

Finally, in sub-section 8.4.2 we have seen the creation of some testbenches based on properties extracted from the data sheet specification of an IP. Some of the properties are simple while others are constituted of a complex network of generators and observers. For future, it can be a great effort to create a library of such properties like maximum jitter, delay on output, buffer size to verify on the IPs. These properties are quite universal in their nature and with little or no modification, should work for most IPs. In fact this will be an important contribution as CCSL and its observers will be more effective with such a library.

Appendices

Appendix A

Acronyms, abbreviations and definitions

A.1 Electronic systems

The following list gives a short description of the acronyms, abbreviations and definitions used throughout this report, together with their explanation.

Term	Description
ARM	Acorn RISC Machines (Company)
AMBA	Advanced Microprocessor Bus Architecture
AHB	Advanced High-speed Bus
ASB	Advanced System Bus
APB	Advanced Peripheral Bus
SPARC	Scalable Processor ARChitecture
DSP	Digital Signal Processor
DMA	Direct Memory Access
UART	Universal Asynchronous Receiver-Transmitter
DUT	Design under Test
SSM	Safe State Machine
FSM	Finite State Machine
FPGA	Field Programmable Gate Array
ASIC	Application-Specific Integrated Circuit
VHSIC	Very High-Speed Integrated Circuit
VHDL	VHSIC Hardware Description Language
SPRINT	Open SoC Design Platform for Reuse and Integration of IPs
MOCC	Model of Computation and Communication
DE	Design Environment
EDA	Electronic Design Automation
ESL	Electronic System Level
ESD	Electronic System Design
MDE	Model-Driven Engineering
MDA	Model-Driven Architecture
MDD	Model-Driven Development
SLD	System Level Design

Term	Description
CAD	Computer Aided Design
TTM	Time to Market
SPT	Schedulability, Performance and Time (Profile)
ADL	Architecture Description Language
HDL	Hardware Description Language
API	Application Program Interface
NFP	Non-Functional Properties
LAU	Least Addressable Unit (of memory)
OSCI	Open SystemC Initiative
PV	Programmers View
PVT	Programmers View with Timing
RTL	Register Transfer Level
SoC	System-on-a-Chip
IC	Integrated Circuit
CIMPACA	Centre Intégré de Microélectronique Provence-Alpes-Côte d'Azur
SPIRIT	Structure for Packaging, Integrating and Re-using IP within Tool flows
IP-XACT	Standard proposed by SPIRIT Consortium
TLM	Transaction-Level Modeling
UTF	Untimed Functional Models
CP	Communicating Processes
CPT	Communicating Processes with Time
BCA	Bus Cycle Accurate
CA	Cycle Accurate
CC	Cycle-Callable
VLNV	Vendor Library Name Version
ATL	ATLAS Transformation Language
XSLT	XSL Transform
XML	eXtensible Markup Language

A.2 CCSL symbols

Mathematical symbols

Symbol	Meaning
$\mathbb{N}$	the set of the natural numbers: $0, 1, 2, \cdots$
$\mathbb{N}^\star$	the set of the natural numbers without 0: $1, 2, \cdots$

Discrete clocks

Notation	Meaning
$c[k]$for $k \in \mathbb{N}^\star$	k^{th} instant of clock c

Instant relations

Notation	Meaning
$i \equiv j$	Instant i coincides with instant j
$i \prec j$	Instant i strictly precedes instant j
$i \preccurlyeq j$	Instant i (non strictly) precedes instant j
$i \# j$	Instant i and instant j are not coincident

Clock relations

Notation	Meaning
$a \boxed{=} b$	clock a and b are synchronous
$a \boxed{\#} b$	clock a and b are exclusive
$a \boxed{\subset} b$	clock a is a sub-clock of clock b
$a \boxed{\prec} b$	clock a strictly precedes clock b
$a \boxed{\preccurlyeq} b$	clock a (non strictly) precedes clock b
$a \boxed{\sim} b$	clock a (strictly) alternates with clock b

Appendix B

Acquisition System Codes

B.1 Esterel code

Interfaces

```
data ApplTypes:
    type Data_t = unsigned <[8]>;
    type Addr_t = unsigned <[16]>;
end data


interface AcqT:
    extends ApplTypes;
    input Value: Data_t;
    output Sample;
end interface


interface CommonT:
    extends ApplTypes;
    output Data: Data_t;
    output Addr: Addr_t;
end interface

interface MSaveT:
    extends CommonT;
    input Grant, Done;
    output Breq;
end interface

interface SSaveT:
    extends mirror CommonT;
    input Sel;
end interface
```

Sensor

```
module Sensor:
  extends ApplTypes;
  port S_A: mirror AcqT;
  input Val: value Data_t; // ``physical'' value

  loop
```

```
      await S_A.Sample;
      await 3 tick; // simulates an acquisition
      emit ?S_A.Value <= ?Val
    end loop
  end module
```

Processor

```
  module Processor:
    extends ApplTypes;
    function DetermineAddr(Data_t):Addr_t;
    input StartAcq, StartW;
    port M_A: AcqT;
    port M_B: MSaveT;

    // behavior
    loop
      await
      case StartAcq do
        emit M_A.Sample; // trigger a new acq
        await M_A.Value; // returned value
      case StartW do
        abort
          sustain M_B.Breq
        when M_B.Grant;
        // now the bus is granted: store data
        emit {
          ?M_B.Data <= ?M_A.Value,
          ?M_B.Addr<= DetermineAddr(?M_A.Value)
        };
        await M_B.Done
      end await
    end loop
  end module
```

Memory

```
  module Memory:
      extends ApplTypes;
      port S_B: SSaveT;

      output JustWritten: Data_t;// for simul

      signal m: Data_t init 0 in
          loop
              await S_B.Sel;
              // ignore Addr
              emit {
                  ?m <= ?S_B.Data,
                  ?JustWritten <= ?m
              }
          end loop
      end signal
end module
```

Bus

```
  module Bus:
      extends ApplTypes;
      port MM_1: mirror MSaveT;
      port MM_2: mirror MSaveT;
      port MS_1: mirror SSaveT;
```

```
    port MS_2: mirror SSaveT;
    port MS_3: mirror SSaveT;

    // behavior

    signal
        LData:Data_t, LAddr:Addr_t, IllegalWhere
    in
        loop
            await
                case MM_1.Breq do
                    pause;
                    emit MM_1.Grant;
                    pause;
                    emit {
                        ?LAddr <= ?MM_1.Addr,
                        ?LData <= ?MM_1.Data
                    }
                case MM_2.Breq do
                    pause;
                    emit MM_2.Grant;
                    pause;
                    emit {
                        ?LAddr <= ?MM_2.Addr,
                        ?LData <= ?MM_2.Data
                    }
            end await;
            if  // emit to which is concerned
                case ?LAddr = 0 do
                    emit {
                        MS_1.Sel,
                        ?MS_1.Addr <= ?LAddr,
                        ?MS_1.Data <= ?LData
                    }
                case ?LAddr = 1 do
                    emit {
                        MS_2.Sel,
                        ?MS_2.Addr <= ?LAddr,
                        ?MS_2.Data <= ?LData
                    }
                case ?LAddr = 2 do
                    emit {
                        MS_3.Sel,
                        ?MS_3.Addr <= ?LAddr,
                        ?MS_3.Data <= ?LData
                    }
                default do
                    emit IllegalWhere
            end if

        end loop
    end signal
end module
```

Application

```
main module Application:
  extends ApplTypes;

  input StartAcq[2], StartW[2]; // for simul
  input EndAcq[2], Val[2]:Data_t; // for simul
  output JustWritten[3]:Data_t;
```

```
signal
        port SAP[2]: AcqT,
        port MMP[2]: MSaveT,
        port MSP[3]: SSaveT
in
        run S1/Sensor
            [
                SAP[0]/S_A,
                Val[0]/Val,
                EndAcq[0]/EndAcq
            ]
||
        run S2/Sensor
            [
                SAP[1]/S_A,
                Val[1]/Val,
                EndAcq[1]/EndAcq
            ]
||
        run P1/Processor
            [
                SAP[0]/M_A,
                MMP[0]/M_B,
                StartAcq[0]/StartAcq,
                StartW[0]/StartW
            ]
||
        run P2/Processor
            [
                SAP[1]/M_A,
                MMP[1]/M_B,
                StartAcq[1]/StartAcq,
                StartW[1]/StartW
            ]
||
        run M1/Memory
            [
                MSP[0]/S_B,
                JustWritten[0]/JustWritten
            ]
||
        run M2/Memory
            [
                MSP[1]/S_B,
                JustWritten[1]/JustWritten
            ]
||
        run M3/Memory
            [
                MSP[2]/S_B,
                JustWritten[2]/JustWritten
            ]
||
        run Bus
            [
                MMP[0]/MM_1,
                MMP[1]/MM_2,
                MSP[0]/MS_1,
                MSP[1]/MS_2,
                MSP[2]/MS_3
            ]
```

```
    end signal

  end module
```

B.2 SystemC code

Sensor

sensor.h

```
  #ifndef _SENSOR_H_
  #define _SENSOR_H_

  #include <systemc.h>
  #include "tlm.h"
  #include "pv_slave_base.h"
  #include "pv_target_port.h"
  #include "types.h"

  class sensor :sc_module,
    public pv_slave_base< Addr_t,Data_t >
  {
  public:
    SC_HAS_PROCESS(sensor);
    sensor( sc_module_name module_name );
    ~sensor();
    // bus side interface
    pv_target_port< Addr_t,Data_t > S_A;

    // bus functions
    tlm::tlm_status write(
        const Addr_t &addr , const Data_t &data,
        const unsigned int byte_enable = tlm::NO_BE,
        const tlm::tlm_mode mode = tlm::REGULAR,
        const unsigned int export_id = 0 );
    tlm::tlm_status read(
        const Addr_t &addr , Data_t &data,
        const unsigned int byte_enable = tlm::NO_BE,
        const tlm::tlm_mode mode = tlm::REGULAR,
        const unsigned int export_id = 0 );

  private:
    Data_t lAcq;
  };

  #endif
```

sensor.cpp

```
  #include "sensor.h"

  sensor::sensor (sc_module_name module_name):
    sc_module(module_name),
    pv_slave_base< Addr_t,Data_t >(name()),
    S_A("S_A")
```

```
{
  S_A( *this );
}

sensor::~sensor() {}

tlm::tlm_status sensor::read(
            const Addr_t &addr , Data_t &data,
            const unsigned int byte_enable,
            const tlm::tlm_mode mode,
            const unsigned int export_id)
{
  tlm::tlm_status status;
  cout << sensor::name() << "::bevSample()" << endl;
  cout << sensor::name() << ": Sample value = ";
  cin >> lAcq;
  cout << sensor::name() << ": Data sent to Processor = "
                         << lAcq << endl;
  data = lAcq;
  status.set_ok();
  return status;
}

tlm::tlm_status sensor::write(
            const Addr_t &addr , const Data_t &data,
            const unsigned int byte_enable,
            const tlm::tlm_mode mode,
            const unsigned int export_id)
{
  tlm::tlm_status status;
  status.set_ok();
  return status;
}
```

Processor

processor.h

```
#ifndef _PROCESSOR_H_
#define _PROCESSOR_H_

#include <systemc.h>
#include "tlm.h"
#include "pv_initiator_port.h"
#include "types.h"

class processor :
  public sc_module
{
public:
  SC_HAS_PROCESS(processor);
  processor( sc_module_name module_name );
  ~processor();
  // bus side interface
  pv_initiator_port< Addr_t,Data_t > M_A;
```

```
    pv_initiator_port< Addr_t,Data_t > M_B;
    sc_in<Command_t> Start;
    sc_event eStart;

  private:
    void Compute();
    Addr_t calcAddr(Data_t);
    Data_t lData;
    Addr_t lAddr;
  };

  #endif
```

processor.cpp

```
  #include "processor.h"

  processor::processor(sc_module_name module_name):
    sc_module(module_name),
    M_A("M_A"),
    M_B("M_B"),
    Start("Start")
  {
    SC_THREAD(Compute);
  }

  processor::~processor() {}

  void processor::Compute() {
    tlm::tlm_status status;
    Command_t lCmd;
    while(1) {
      wait(eStart);
      eStart.cancel();
      wait(SC_ZERO_TIME);
      lCmd = Start.read();
      switch(lCmd)
      {
      case ACQ: // read sensor
        status = M_A.read(0, lData); // only 1 address
        if (status.is_ok()){
          cout << processor::name() << ": Data Acquired"
            << endl;
        } else {
          cout << processor::name() << ": Data Acquire Failed"
            << endl;
        }
        break;
      case SAVE: // save the value contained in lData
        lAddr = calcAddr(lData);
        status = M_B.write(lAddr, lData);
        if (status.is_ok()){
          cout << processor::name() << ": Data Sent" << endl;
        } else {
          cout << processor::name() << ": Data Sending Failed"
```

```
            << endl;
        }
        break;
      }
    }
  }

  Addr_t processor::calcAddr(Data_t temp){
    if(temp < (0x2FFF*5)) // 61435
      return (temp / 5);
    else
      return 0x2FFF;
  }
```

Bus

bus.h

```
  #ifndef _BUS_H_
  #define _BUS_H_

  #include <systemc.h>
  #include "pv_router.h"
  #include "types.h"

  // this class has 2 target ports and 3 initiator ports
  typedef pv_router< Addr_t , Data_t > basic_router;

  class bus:  public basic_router
  {
  public:
    bus (sc_module_name module_name, const char* mapFile):
        basic_router (module_name, mapFile) {}
    void end_of_elaboration() {
      basic_router::end_of_elaboration();
      cout << name() << " constructed." << endl;
    }
  };

  #endif
```

Memory

memory.h

```
  #ifndef _MEMORY_H_
  #define _MEMORY_H_

  #include <systemc.h>
  #include "tlm.h"
  #include "pv_slave_base.h"
  #include "pv_target_port.h"
  #include "types.h"

  class memory:  public sc_module,
```

```
    public pv_slave_base< Addr_t,Data_t >
{
public:
  SC_HAS_PROCESS(memory);
  memory( sc_module_name module_name );
  ~memory();
// bus side interface
  pv_target_port< Addr_t,Data_t > S_B;

// bus functions
  tlm::tlm_status write(
      const Addr_t &addr , const Data_t &data,
      const unsigned int byte_enable = tlm::NO_BE,
      const tlm::tlm_mode mode = tlm::REGULAR,
      const unsigned int export_id = 0 );
  tlm::tlm_status read(
      const Addr_t &addr , Data_t &data,
      const unsigned int byte_enable = tlm::NO_BE,
      const tlm::tlm_mode mode = tlm::REGULAR,
      const unsigned int export_id = 0 );
private:
  Data_t iMemory[0x1000];
};

#endif
```

memory.cpp

```
#include "memory.h"

memory::memory(sc_module_name module_name):
  sc_module(module_name),
  pv_slave_base< Addr_t,Data_t >(name()),
  S_B("S_B")
{
  S_B( *this );
}

memory::~memory() {
    for (int i=0; i < 0x1000; i++) iMemory[i]=0;
}

tlm::tlm_status memory::write(
            const Addr_t &addr , const Data_t &data,
            const unsigned int byte_enable,
            const tlm::tlm_mode mode,
            const unsigned int export_id)
{
  tlm::tlm_status status;
  printf ("\%s : Writing DATA = \%x at ADDRESS = \%x \n",
          memory::name(),data,addr);
  iMemory[addr] = data;
  status.set_ok();
  return status;
}
```

```
tlm::tlm_status memory::read(
            const Addr_t &addr , Data_t &data,
            const unsigned int byte_enable,
            const tlm::tlm_mode mode,
            const unsigned int export_id)
{
  tlm::tlm_status status;
  data = iMemory[addr];
  printf ("\%s : Reading DATA = \%x at ADDRESS = \%x \n",
          memory::name(),data,addr);
  status.set_ok();
  return status;
}
```

Top and Other Modules

types.h

```
#ifndef _TYPES_H_
#define _TYPES_H_

  enum Command_t {ACQ, SAVE};
  typedef unsigned int Addr_t;
  typedef unsigned int Data_t;

#endif
```

bus.map

```
mytop.umem0.S_B    0000 1000
mytop.umem1.S_B    1000 1000
mytop.umem2.S_B    2000 1000
```

top.h

```
#ifndef _TOP_H_
#define _TOP_H_

#include <systemc.h>
#include "sensor.h"
#include "processor.h"
#include "bus.h"
#include "memory.h"
#include "types.h"

SC_MODULE(top)
{
public:
  sc_signal<Command_t>  myStart0;
  sc_signal<Command_t>  myStart1;
// component instances
  sensor*    i_sensor_0;
  sensor*    i_sensor_1;
  processor*  i_proc_0;
```

```
    processor*  i_proc_1;
    bus*     i_bus;
    memory*    i_mem_0;
    memory*    i_mem_1;
    memory*    i_mem_2;

  SC_CTOR(top) {
    // Thread Declaration
    SC_THREAD(Simulation);
    // instantiation
    i_sensor_0  = new sensor("usensor0");
    i_sensor_1  = new sensor("usensor1");
    i_proc_0  = new processor("uproc0");
    i_proc_1  = new processor("uproc1");
    i_bus    = new bus("ubus","bus.map");
    i_mem_0   = new memory("umem0");
    i_mem_1   = new memory("umem1");
    i_mem_2   = new memory("umem2");

    cout<< "Instantiated..." << endl;

    // Interface Binding
    i_proc_0->Start(myStart0);
    i_proc_1->Start(myStart1);

    i_proc_0->M_A(i_sensor_0->S_A);
    i_proc_1->M_A(i_sensor_1->S_A);

    i_proc_0->M_B(i_bus->target_port);
    i_proc_1->M_B(i_bus->target_port);

    i_bus->initiator_port(i_mem_0->S_B);
    i_bus->initiator_port(i_mem_1->S_B);
    i_bus->initiator_port(i_mem_2->S_B);

    cout<< "Ports Binding done!" << endl;
  }

  ~top() {
    delete i_sensor_0;
    delete i_sensor_1;
    delete i_proc_0;
    delete i_proc_1;
    delete i_bus;
    delete i_mem_0;
    delete i_mem_1;
    delete i_mem_2;
  }

  void Simulation() {
    // Start Simulation
    wait(50,SC_NS);

    // System Reset
```

```
    cout << endl << "@Top: Resetting System at time "
         << sc_time_stamp() << endl << endl;
    // System Initialize
    cout<< "  start0 = ACQ  " << endl;
    myStart0.write(ACQ); i_proc_0->eStart.notify();
    cout<< "  start1 = ACQ  " << endl;
    myStart1.write(ACQ); i_proc_1->eStart.notify();
    wait(50,SC_NS);

    cout<< "  start0 = SAVE " << endl;
    myStart0.write(SAVE); i_proc_0->eStart.notify();
    cout<< "  start1 = SAVE " << endl;
    myStart1.write(SAVE); i_proc_1->eStart.notify();
    wait(50,SC_NS);
  }

  };

  #endif
```

top.cpp

```
  #include "top.h"

  int sc_main( int argc , char **argv )
  {
    top mytop("mytop");

    sc_start(10000,SC_NS);
    cout << "@Top: Simulation ended at ... "
         << sc_time_stamp() << endl;
    system("pause");

    return 0;
  }
```

Appendix C

CCSL and VHDL Observers

```
Package Ccsl is
  constant EPSILON: Time;
end  Ccsl;

Package body Ccsl is
  -- the smallest duration as possible
  constant EPSILON: Time:= 1 fs;
end Ccsl;
```

C.1 Adaptors

Rising edge adaptor

```
entity Ccsl_A_risingEdge is
  port (
    a: in std_logic;
    c_a: out bit:= '0');
end entity Ccsl_A_risingEdge;

architecture Ccsl_A_risingEdge_arch of Ccsl_A_risingEdge is
begin
  process
  begin
        wait until rising_edge(a);
        c_a <= '1', '0' after EPSILON;
  end process;
end Ccsl_A_risingEdge_arch;
```

Falling edge adaptor

```
entity Ccsl_A_fallingEdge is
  port (
    a: in std_logic;
    c_a: out bit:= '0');
end entity Ccsl_A_risingEdge;

architecture Ccsl_A_fallingEdge_arch of Ccsl_A_fallingEdge is
```

```
begin
  process
  begin
        wait until falling_edge(a);
        c_a <= '1', '0' after EPSILON;
  end process;
end Ccsl_A_fallingEdge_arch;
```

C.2 Observers

Equality

```
entity Ccsl_R_equal is
  port (
    c_a,c_b: in bit;
    v: out bit:= '0');
end entity Ccsl_R_equal;

architecture Ccsl_R_equal_arch of Ccsl_R_equal is
begin
  postponed v <= not(c_a xnor c_b);
end architecture Ccsl_R_equal_arch;
```

Exclusion

```
entity Ccsl_R_exclusive is
  port (
    c_a,c_b: in bit;
    v: out bit:= '0');
end entity Ccsl_R_equal;

architecture Ccsl_R_exclusive_arch of Ccsl_R_exclusive is
begin
  postponed v <= c_a and c_b;
end architecture Ccsl_R_exclusive_arch;
```

Subclocking

```
entity Ccsl_R_subclock is
  port (
    c_a,c_b: in bit;
    v: out bit:= '0');
end entity Ccsl_R_subclock;

architecture Ccsl_R_subclock_arch of Ccsl_R_subclock is
begin
  postponed v <= c_a and not(c_b);
end architecture Ccsl_R_subclock_arch;
```

(non strict) precedence

```
entity Ccsl_R_precedes is
  port (
    c_a,c_b: in bit;
```

```
    v: out bit:= '0');
end entity Ccsl_R_precedes;

architecture Ccsl_R_precedes_arch of Ccsl_R_precedes is
  signal d:integer := 0;
begin
  process(c_a,c_b)
    variable c:integer := 0;
  begin
    -- ignore now = 0
    if now >= EPSILON then
      if c_a'event and c_a = '1' then
        c := c + 1;
      end if;
      if c_b'event and c_b = '1' then
        c := c - 1;
      end if;
      d <= c;
    end if;
  end process;

  postponed v <= '1' when d<0 else '0';
end architecture Ccsl_R_precedes_arch;
```

strict precedence

```
entity Ccsl_R_s_precedes is
  port (
    c_a,c_b: in bit;
    v: out bit:= '0');
end entity Ccsl_R_s_precedes;

architecture Ccsl_R_s_precedes_arch of Ccsl_R_s_precedes is
  type State_t is (Sz,Sp,Sv,Tp,Tap,Tbp,Tbz,Tabp);
  signal State: State_t := Sz;
begin
  process (c_a,c_b)
      variable d: integer := 0;
  begin
    -- ignore now = 0
    if now >= EPSILON then
      case State is
        when Sz =>
          if c_b'event and c_b = '1' then
            -- ^B
            State <= Sv;
          elsif c_a'event and c_a = '1' then
            -- ^A
            State <= Tp;
          end if;
        when Tp =>
          if c_b'event and c_b = '1' then
            State <= Sv;
          elsif c_a'event and c_a = '0' then
```

```
      d := 1;
      State <= Sp;
    end if;
  when Sp =>
    if c_a'event and c_a = '1' then
      if c_b'event and c_b = '1' then
        -- both ^A and ^B
        State <= Tabp;
      else
        -- ^A only
        d := d + 1;
        State <= Tap;
      end if;
    elsif c_b'event and c_b = '1' then
      -- ^B only
      if d > 1 then
        d := d - 1;
        State <= Tbp;
      else -- d = 1
        d := 0;
        State <= Tbz;
      end if;
    end if;
  when Tap =>
    if c_b'event and c_b = '1' then
      d := d - 1;
      State <= Tabp;
    elsif c_a'event and c_a = '0' then
      -- next simulation time
      State <= Sp;
    end if;
  when Tbp =>
    if c_a'event and c_a = '1' then
      d := d + 1;
      State <= Tabp;
    elsif c_b'event and c_b = '0' then
      -- next simulation time
      State <= Sp;
    end if;
  when Tbz =>
    if c_a'event and c_a = '1' then
      d := 1;
      State <= Tabp;
    elsif c_b'event and c_b = '0' then
      -- next simulation time
      State <= Sz;
    end if;
  when Tabp =>
    if c_a'event and c_a = '0' then
      -- next simulation time
      State <= Sp;
    end if;
  when Sv =>
    null; -- this state is a sink
```

```
        end case;
      end if;
    end process;

    postponed v <= '1' when State = Sv else '0';
  end architecture Ccsl_R_s_precedes_arch;
```

strict alternation

```
  entity Ccsl_R_s_alternates is
    port (
      c_a,c_b: in bit;
      v: out bit:= '0');
  end entity Ccsl_R_s_alternates;

  architecture Ccsl_R_s_alternates_arch of Ccsl_R_s_alternates is
    type State_t is (S0,S1,Sv,T01,T10);
    signal State: State_t := S0;
  begin
    process (c_a,c_b)
    begin
      -- ignore now = 0
      if now >= EPSILON then
        case State is
          when S0 =>
            if c_b'event and c_b = '1' then
              State <= Sv;
            elsif c_a'event and c_a = '1' then
              State <= T01;
            end if;
          when S1 =>
            if c_a'event and c_a = '1' then
              State <= Sv;
            elsif c_b'event and c_b = '1' then
              State <= T10;
            end if;
          when Sv =>
            null; -- this state is a sink
          when T01 =>
            if c_b'event and c_b = '1' then
              State <= Sv;
            elsif c_a'event and c_a = '0' then
              -- next simulation time
              State <= S1;
            end if;
          when T10 =>
            if c_a'event and c_a = '1' then
              State <= Sv;
            elsif c_b'event and c_b = '0' then
              -- next simulation time
              State <= S0;
            end if;
        end case;
      end if;
```

```
    end process;

    postponed v <= '1' when State = Sv else '0';
  end architecture Ccsl_R_s_alternates_arch;
```

C.3 Generators

inf

```
  entity Ccsl_E_inf is
    port (
      c_a,c_b: in bit;
      c_c: out bit:= '0');
  end entity Ccsl_E_inf;

  architecture Ccsl_E_inf_arch of Ccsl_E_inf is
    signal cnta: integer := 0;
    signal cntb: integer := 0;
    signal cntc: integer := 0;
    signal do_c: bit := '0';
  begin
    process (c_a,c_b,do_c)
    begin
      -- ignore now = 0
      if now >= EPSILON then
        if c_a'event and c_a = '1' then
          cnta <= cnta +1;
          if cnta = cntc then
            do_c <= '1';
          end if;
        end if;
        if c_b'event and c_b = '1' then
          cntb <= cntb + 1;
          if cntb = cntc then
            do_c <= '1';
          end if;
        end if;
        if do_c'event and do_c = '1' then
          c_c <= '1', '0' after EPSILON;
          cntc <= cntc + 1;
          do_c <= '0' after EPSILON;
        end if;
      end if;
    end process;
  end architecture Ccsl_E_inf_arch;
```

delayedFor

```
  entity Ccsl_E_delay is
    generic (n: positive:= 3);
    port (
      c_a,c_b: in bit;
      c_c: out bit:= '0');
  end Ccsl_E_delay;
```

```
architecture Ccsl_E_delay_arch of Ccsl_E_delay is
  type State_t is (S0,Sa,Sb,Sab);
  signal State: State_t := S0;
  signal buff:bit_vector((n-1) downto 0):=(others=>'0');
begin
  process(c_a,c_b)
  begin
    -- ignore now = 0
    if now >= EPSILON then
      case State is
        when S0 =>
          if c_b'event and c_b = '1' then
            if buff(0) = '1' then
              c_c <= '1', '0' after EPSILON;
            end if;
            if c_a'event and c_a = '1' then
              State <= Sab;
            else
              State <= Sb;
            end if;
          elsif c_a'event and c_a = '1' then
            State <= Sa;
          end if;
        when Sa =>
          if c_b'event and c_b = '1' then
            if buff(0) = '1' then
              c_c <= '1', '0' after EPSILON;
            end if;
            State <= Sab;
          elsif c_a'event and c_a = '0' then
            -- next instant
            State <= S0;
            -- memorize trigger
            buff(n-1)<= '1';
          end if;
        when Sb =>
          if c_a'event and c_a = '1' then
            State <= Sab;
          elsif c_b'event and c_b = '0' then
            -- next instant
            State <= S0;
            -- shift without memorizing
            if n > 1 then
              buff((n-2) downto 0)<= buff((n-1) downto 1);
            end if;
            buff(n-1)<= '0';
          end if;
        when Sab =>
          if c_a'event and c_a = '0' then
            -- next instant
            State <= S0;
            -- shift, then memorize
            buff((n-2) downto 0)<= buff((n-1) downto 1);
```

```
            buff(n-1)<= '1';
          end if;
        end case;
      end if;
    end process;
  end Ccsl_E_delay_arch;
```

filteredBy

```
  entity Ccsl_E_filter is
    generic (
      init: bit_vector:= "";
      period: bit_vector:= "1");
    port (
      c_sup: in  bit;
      c_sub: out bit:= '0');
  end Ccsl_E_filter;

  architecture Ccsl_E_filter_arch of Ccsl_E_filter is
    signal nInit,nPeriod: integer := 0;
  begin
    process(c_sup)
      constant init_size: Natural := init'length;
      constant period_size: Natural := period'length;
    begin
      -- ignore first instant
      if now >= EPSILON then
        if c_sup'event and c_sup = '1' then
          if (nInit < init_size) then
            -- in initial phase
            if init(nInit) = '1' then
              c_sub <= '1', '0' after EPSILON;
            end if;
            nInit <= nInit + 1;
          else  -- in periodic phase
            if period(nPeriod) = '1' then
              c_sub <= '1', '0' after EPSILON;
            end if;
            nPeriod <= (nPeriod + 1) mod period_size;
          end if;
        end if;
      end if;
    end process;
  end Ccsl_E_filter_arch;
```

minus

```
  entity Ccsl_E_minus is
    port (
      c_a,c_b: in bit;
      c_c: out bit:= '0');
  end entity Ccsl_E_minus;

  architecture Ccsl_E_minus_arch of Ccsl_E_minus is
```

```
  type State_t is (S0,Sa,Sp);
  signal State: State_t := S0;
begin
  process (c_a,c_b)
  begin
    case State is
      when S0 =>
        if c_b'event and c_b = '1' then
          State <= Sa; -- absent
        elsif c_a'event and c_a = '1' then
          -- possibly present
          State <= Sp;
          c_c <= '1', '0' after EPSILON;
        end if;

      when Sa =>
        if c_b'event and c_b = '0' then
          -- next simulation instant
          State <= S0;
        end if;
      when Sp =>
        if c_b'event and c_b = '1' then
          State <= Sa;
          c_c <= '0';
        elsif c_a'event and c_a = '0' then
          -- next simulation instant
          State <= S0;
        end if;

    end case;
  end process;

end architecture Ccsl_E_minus_arch;
```

Bibliography

[Aer09] Aeroflex Gaisler Research. *GRLIB IP Core User's Manual*, August 2009. http://www.gaisler.com/products/grlib/grip.pdf.

[AM08] Charles André and Frédéric Mallet. Clock constraint specification language in UML/MARTE CCSL. Research Report 6540, INRIA, 05 2008.

[AM09a] Charles André and Frédéric Mallet. Modèles de contraintes temporelles pour systèmes polychrones. *JESA*, 43(7–8–9):725–739, 2009.

[AM09b] Charles André and Frédéric Mallet. Specification and verification of time requirements with CCSL and Esterel. In Christoph M. Kirsch and Mahmut T. Kandemir, editors, *LCTES*, pages 167–176. ACM, 2009.

[AMdS07] Charles André, Frédéric Mallet, and Robert de Simone. Modeling time(s). In Gregor Engels, Bill Opdyke, Douglas C. Schmidt, and Frank Weil, editors, *MoDELS*, volume 4735 of *Lecture Notes in Computer Science*, pages 559–573. Springer, 2007.

[AMdS08] Charles André, Frédéric Mallet, and Robert de Simone. *Embedded Systems Specification and Design Languages*, volume 10 of *LNEE*, chapter Modeling of AADL data-communications with UML Marte, pages 150–170. Springer, May 2008.

[AMMdS08] Charles André, Frédéric Mallet, Aamir Mehmood Khan, and Robert de Simone. Modeling Spirit IP-XACT in UML Marte. In *Conf. on Design, Automation and Test in Europe (DATE), MARTE Workshop*, pages 35–40, March 2008.

[AMPF07] C. André, F. Mallet, and M.-A. Peraldi-Frati. A multiform time approach to real-time system modeling: Application to an automotive system. In Universidade Nova de Lisboa, editor, *Int. Symp. on Industrial Embedded Systems*, pages 234–241, Lisboa, Portugal, July 2007. IEEE.

[And96] C. André. Representation and analysis of reactive behaviors: A synchronous approach. In *Computational Engineering in Systems Applications (CESA)*, pages 19–29, Lille (F), July 1996. IEEE-SMC.

[And09a] Charles André. Library of CCSL observers for CCSL. Technical Report 6925, INRIA, 12 2009.

[And09b] Charles André. Syntax and semantics of the clock constraint specification language (CCSL). Research Report 6925, INRIA, 05 2009.

[ARM99] ARM Limited. *AMBA Specification Rev 2.0*, May 1999. http://www.arm.com/products/solutions/AMBA_Spec.html.

[ASHH09] Tero Arpinen, Erno Salminen, Timo D. Hämäläinen, and Marko Hännikäinen. Evaluating UML2 Modeling of IP-XACT Objects for Automatic MP-SoC Integration onto FPGA. In *Conf. on Design, Automation and Test in Europe (DATE)*, April 2009.

[Ayn09] Aynsley, John. *OSCI TLM-2.0 Language Reference Manual*. Open SystemC Initiative, 2009.

[BB91] A. Benveniste and G. Berry. The synchronous approach to reactive and real-time systems. *Proceeding of the IEEE*, 79(9):1270–1282, September 1991.

[BDF08] Nicola Bombieri, Nicola Deganello, and Franco Fummi. Integrating rtl ips into tlm designs through automatic transactor generation. In *DATE*, pages 15–20. IEEE, 2008.

[BdS91] Frédéric Boussinot and Robert de Simone. The esterel language. another look at real time programming. *Proceedings of the IEEE*, 79(9):1293–1304, September 1991.

[Ber00a] Gérard Berry. *The Esterel Language Primer, version v5_91*. Ecole des Mines de Paris, CMA, INRIA, July 2000.

[Ber00b] Gérard Berry. The foundations of Esterel. In C. Stirling G. Plotkin and M. Tofte, editors, *Proof, Language and Interaction: Essays in Honour of Robin Milner*. MIT Press, 2000.

[Ber02] Janick Bergeron. *Writing Testbenches, Functional Verification of HDL Models*. Kluwer Academic Publishers, 2002.

[Ber07] Gérard Berry. SCADE: Synchronous design and validation of embedded control software. In S. Ramesh and Prahladavaradan Sampath, editors, *Next Generation Design and Verification Methodologies for Distributed Embedded Control Systems*, pages 19–33. Springer-Verlag, 2007.

[BK08] V.S. Bagad and S.P. Kawachale. *VLSI Design*. Technical Publications Pune, 2008.

[BKS03] Gérard Berry, Michael Kishinevsky, and Satnam Singh. System level design and verification using a synchronous language. In *ICCAD*, pages 433–440. IEEE Computer Society / ACM, 2003.

[BMP07] Brian Bailey, Grant Martin, and Andrew Piziali. *ESL Design and Verification, A Prescription for Electronic System-Level Methodology*. Morgan Kaufmann Academic Press, 2007.

[Boe84] B. W. Boehm. Verifying and validating software requirements and design specifications. *IEEE Softw.*, 1(1):75–88, 1984.

[CG03] Lukai Cai and Daniel Gajski. Transaction level modeling: an overview. In Rajesh Gupta, Yukihiro Nakamura, Alex Orailoglu, and Pai H. Chou, editors, *CODES+ISSS*, pages 19–24. ACM, 2003.

[Chu06] Pong P. Chu. *RTL Hardware Design Using VHDL: Coding for Efficiency, Portability, and Scalability.* Wiley-IEEE Press, 2006.

[CSL+03] Rong Chen, Marco Sgroi, Luciano Lavagno, Grant Martin, Alberto Sangiovanni-Vincentelli, and Jan Rabaey. UML and platform-based design. In *UML for real: design of embedded real-time systems*, pages 107–126, Norwell, MA, USA, 2003. Kluwer Academic Publishers.

[DGG02] Rainer Dømer, Andreas Gerstlauer, and Daniel Gajski. *SpecC Language Reference Manual, Version 2.0.* SpecC Technology Open Consortium, 2002.

[Dou98] Bruce P. Douglass. *Real-Time UML. Developing efficient objects for embedded systems.* Object technology. Addison Wesley Longman, Inc., 1998.

[DTA+08] Sébastien Demathieu, Frédéric Thomas, Charles André, Sébastien Gérard, and François Terrier. First experiments using the UML profile for Marte. In *11th IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC 2008)*, pages 50–57. IEEE Computer Society, 2008.

[Erb06] Cagkan Erbas. *System-Level Modeling and Design Space Exploration for Multiprocessor Embedded System-on-Chip Architectures.* Amsterdam University Press, 2006.

[Est05] Esterel Technologies. *The Esterel v7 Reference Manual*, November 2005.

[Fou06] The Eclipse Foundation. *Atlas Transformation Language, User Manual, version 0.7.* LINA and INRIA, ATLAS Group, February 2006.

[Gaj07] Daniel D. Gajski. New strategies for system-level design. In Patrick Girard, Andrzej Krasniewski, Elena Gramatová, Adam Pawlak, and Tomasz Garbolino, editors, *DDECS*, page 15. IEEE Computer Society, 2007.

[GHC09] Jiri Gaisler, Sandi Habinc, and Edvin Catovic. *GRLIB IP Library User's Manual*, 2009. http://www.gaisler.com/products/grlib/grlib.pdf.

[Ghe05] Frank Ghenassia. *Transaction-level modeling with SystemC.* Springer, 2005.

[GL06] Michaela Guiney and Eric Leavitt. An introduction to OpenAccess: an open source data model and API for IC design. In Fumiyasu Hirose, editor, *ASP-DAC*, pages 434–436. IEEE, 2006.

[GOO06] Susanne Graf, Ileana Ober, and Iulian Ober. A real-time profile for UML. *STTT*, 8(2):113–127, 2006.

[Gro02] Thorsten Grotker. *System Design with SystemC.* Kluwer Academic Publishers, Norwell, MA, USA, 2002.

[Hal92] Nicolas Halbwachs. *Synchronous Programming of Reactive Systems.* Kluwer Academic Publishers, Norwell, MA, USA, 1992.

[Har87] David Harel. Statecharts: A visual formalism for complex systems. *Sci. Comput. Program.*, 8(3):231–274, 1987.

[HCRP91] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous dataflow programming language lustre. *Proceedings of the IEEE*, 79(9):1305–1320, September 1991.

[HLR94] Nicolas Halbwachs, Fabienne Lagnier, and Pascal Raymond. Synchronous observers and the verification of reactive systems. In *AMAST '93: Proceedings of the Third International Conference on Methodology and Software Technology*, pages 83–96, London, UK, 1994. Springer-Verlag.

[HN96] David Harel and Amnon Naamad. The statemate semantics of statecharts. *ACM Trans. Softw. Eng. Methodol.*, 5(4):293–333, 1996.

[HSGP06] Brian Henderson-Sellers and Cesar Gonzalez-Perez. Uses and abuses of the stereotype mechanism in uml 1.x and 2.0. In Oscar Nierstrasz, Jon Whittle, David Harel, and Gianna Reggio, editors, *MoDELS*, volume 4199 of *Lecture Notes in Computer Science*, pages 16–26. Springer, 2006.

[IEE00] IEEE. *IEEE Standard VHDL Language Reference Manual*. IEEE Standard, 2000. IEEE Std 1076a-2000.

[IEE05] IEEE Standards Association. *Open SystemC Language Reference Manual*. Open SystemC Initiative, 2005. IEEE Std. 1666–2005.

[IEE09] IEEE. *12th IEEE Int. Symp. on Object-Oriented Real-Time Distributed Computing (ISORC 2009)*. IEEE Computer Society, March 2009.

[JAB+06] Frédéric Jouault, Freddy Allilaire, Jean Bézivin, Ivan Kurtev, and Patrick Valduriez. Atl: a qvt-like transformation language. In Peri L. Tarr and William R. Cook, editors, *OOPSLA Companion*, pages 719–720. ACM, 2006.

[JABK08] Frédéric Jouault, Freddy Allilaire, Jean Bézivin, and Ivan Kurtev. ATL: a model transformation tool. *Sci. Comput. Program.*, 72(1-2):31–39, 2008.

[Jan03] Axel Jantsch. *Modeling Embedded Systems and SoC's: Concurrency and Time in Models of Computation*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2003.

[JCG08] JCGM. *Vocabulaire international de métrologie (VIM) – Concepts fondamentaux et généraux et termes associés*. Joint Committee for Guides in Metrology, 2008. JCGM 200.

[KWB03] Anneke G. Kleppe, Jos Warmer, and Wim Bast. *MDA Explained: The Model Driven Architecture: Practice and Promise*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2003.

[Lam78] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, 1978.

[LGLBLM91] Paul Le Guernic, Michel Le Borgne, and Claude Le Maire. Programming real-time applications with Signal. *Proceedings of the IEEE*, 79(9):1321–1336, September 1991.

[MA09] Frédéric Mallet and Charles André. On the semantics of UML/marte clock constraints. In *ISORC* [IEE09], pages 305–312.

[MAD09] Frédéric Mallet, Charles André, and Julien Deantoni. Executing AADL models with UML/marte. In *ICECCS*, pages 371–376. IEEE Computer Society, June 2009.

[Mal08] Frédéric Mallet. Clock Constraint Specification Language: Specifying clock constraints with UML/MARTE. *Innovations in Systems and Software Engineering*, 4(3):309–314, October 2008.

[MdS09] Frédéric Mallet and Robert de Simone. *MARTE vs. AADL for Discrete-Event and Discrete-Time Domains*, volume 36 of *LNEE*, chapter 2, pages 27–41. Springer, April 2009.

[MG06] Tom Mens and Pieter Van Gorp. A taxonomy of model transformation. *Electronic Notes in Theoretical Computer Science*, 152:125–142, March 2006.

[MM05] Grant Martin and Wolfgang Mueller. *UML for SoC Design*. Springer, 2005.

[MPA09] Frédéric Mallet, Marie-Agnès Peraldi-Frati, and Charles André. Marte CCSL to execute East-ADL timing requirements. In *ISORC* [IEE09], pages 249–253.

[MT00] Nenad Medvidovic and Richard N. Taylor. A classification and comparison framework for software architecture description languages. *IEEE TRANSACTIONS ON SOFTWARE ENGINEERING*, 26(1):70–93, 2000.

[OMG03] OMG. *UML 2.0 OCL Specification*. Object Management Group, October 2003. ptc/03-10-14.

[OMG05] OMG. *UML Profile for Schedulability, Performance, and Time Specification*. Object Management Group, Object Management Group, Inc., 492 Old Connecticut Path, Framing-ham, MA 01701., January 2005. OMG document number: formal/05-01-02 (v1.1).

[OMG06] OMG. *UML profile for System on a Chip v1.0.1*. Object Management Group, 2006. OMG document number: formal/06-08-01.

[OMG07] OMG. *Unified Modeling Language, Superstructure*. Object Management Group, November 2007. Version 2.1.2 formal/2007-11-02.

[OMG08a] OMG. *Meta Object Facility (MOF) 2.0 Query/View/Transformation Specification, v1.0*. Object Management Group., April 2008. http://www.omg.org/spec/QVT/1.0/.

[OMG08b] OMG. *Systems Modeling Language (SysML) Specification 1.1*. Object Management Group, May 2008. OMG document number: ptc/08-05-17.

[OMG08c] OMG. *UML Profile for MARTE, beta 2*. Object Management Group, June 2008. OMG document number: ptc/08-06-08.

[OMG09a] OMG. *OMG Unified Modeling Language, Infrastructure, V2.1.2*. Object Management Group, February 2009. formal/2007-11-04.

[OMG09b] OMG. *UML Superstructure, v2.2*. Object Management Group, February 2009. formal/2009-02-02.

[PBEB07] Dumitru Potop-Butucaru, S. Edwards, and G. Berry. *Compiling Esterel*. Springer, 2007.

[Rev08] Sébastien Revol. *Profil UML pour TLM: contribution à la formalisation et à l'automatisation du flot de conception et vérification des systèmes sur puce*. PhD thesis, INPG, Grenoble, France, June 2008.

[RSRB05] E. Riccobene, P. Scandurra, A. Rosti, and S. Bocchio. A Soc design methodology involving a UML 2.0 profile for SystemC. In *Conf. on Design, Automation and Test in Europe (DATE)*. IEEE Computer Society, March 2005.

[RSRS99] Bernhard Rumpe, M. Schoenmakers, Ansgar Radermacher, and Andy Schürr. UML + ROOM as a standard ADL? In *ICECCS*, pages 43–53. IEEE Computer Society, 1999.

[RTT+08] S. Revol, S. Taha, F. Terrier, A. Clouard, S. Gérard, A. Radermacher, and J-L. Dekeyser. Unifying HW analysis and SoC design flows by bridging two key standards: UML and IP-XACT. In Bernd Kleinjohann, Lisa Kleinjohann, and Wayne Wolf, editors, *Distributed Embedded Systems: Design, Middleware and Resources*, volume 271 of *IFIP*, pages 69–78. Springer Verlag, 2008.

[Sch03] Klaus-Dieter Schubert. Improvements in functional simulation addressing challenges in large, distributed industry projects. In *DAC '03: Proceedings of the 40th annual Design Automation Conference*, pages 11–14, New York, NY, USA, 2003. ACM.

[Sch05] Tim Schattkowsky. UML 2.0 - overview and perspectives in SoC design. In *DATE*, pages 832–833. IEEE Computer Society, 2005.

[Sel96] Bran Selic. Real-time object-oriented modeling (room). *Real-Time and Embedded Technology and Applications Symposium, IEEE*, 0:214, 1996.

[Sel98] Bran Selic. Using uml for modeling complex real-time systems. In Frank Mueller and Azer Bestavros, editors, *LCTES*, volume 1474 of *Lecture Notes in Computer Science*, pages 250–260. Springer, 1998.

[Sel05] Bran Selic. On software platforms, their modeling with uml 2, and platform-independent design. *Object-Oriented Real-Time Distributed Computing, IEEE International Symposium on*, 0:15–21, 2005.

[Sel06] Bran Selic. Model-driven development: Its essence and opportunities. In *ISORC*, pages 313–319. IEEE Computer Society, 2006.

[Sel07] Bran Selic. A systematic approach to domain-specific language design using UML. In *ISORC*, pages 2–9, Los Alamitos, CA, USA, 2007. IEEE Computer Society.

[SPI08] SPIRIT. *IP-XACT v1.4: A specification for XML meta-data and tool interfaces*. Spirit Consortium, March 2008. http://www.spiritconsortium.org.

[SSS+03] Heinz-Joseph Schlebusch, Gary Smith, Donatella Sciuto, Daniel Gajski, Carsten Mielenz, Christopher K. Lennard, Frank Ghenassia, Stuart Swan, and Joachim Kunkel. Transaction based design: Another buzzword or the solution to a design problem? In *DATE*, pages 10876–10879. IEEE Computer Society, 2003.

[SVCDBS04] Alberto Sangiovanni-Vincentelli, Luca Carloni, Fernando De Bernardinis, and Marco Sgroi. Benefits and challenges for platform-based design. In *DAC '04: Proceedings of the 41st annual Design Automation Conference*, pages 409–414, New York, NY, USA, 2004. ACM.

[SX08] Tim Schattkowsky and Tao Xie. UML and IP-XACT for Integrated SPRINT IP Management. In *Design, Automation Conference (DAC), UML for SoC workshop*, June 2008.

[SXM09] Tim Schattkowsky, Tao Xie, and Wolfgang Mueller. A UML front-end for IP-XACT-based IP management. In *Conf. on Design, Automation and Test in Europe (DATE)*, April 2009.

[vdB94] M. von der Beek. A comparison of statechart variants. In L. de Roever and J. Vytopil, editors, *Formal Techniques in Real-Time and Fault-Tolerant Systems*, volume 863 of *Lecture Notes in Computer Science*, pages 128–148. Springer-Verlag, 1994.

[VMD08] Yves Vanderperren, Wolfgang Mueller, and Wim Dehaene. UML for Electronic Systems Design: A Comprehensive Overview. *Design Automation for Embedded Systems*, 12(4):261–292, December 2008.

[VSBR06] Alexander Viehl, Timo Schönwald, Oliver Bringmann, and Wolfgang Rosenstiel. Formal performance analysis and simulation of UML/SysML models for ESL design. In Georges G. E. Gielen, editor, *DATE*, pages 242–247. European Design and Automation Association, Leuven, Belgium, 2006.

[Wei08] Tim Weilkiens. *Systems Engineering with SysML/UML: Modeling, Analysis, Design.* The MK/OMG Press, Burlington, MA, USA., 2008.

[WGK+00] Thomas Weigert, David Garlan, John Knapman, Birger Møller-Pedersen, and Bran Selic. Modeling of architectures with uml (panel). In Andy Evans, Stuart Kent, and Bran Selic, editors, *UML*, volume 1939 of *Lecture Notes in Computer Science*, pages 556–569. Springer, 2000.

[ZBG+08] J. Zimmermann, O. Bringmann, J. Gerlach, F. Schaefer, and U. Nageldinger. Holistic system modeling and refinement of interconnected microelectronics systems. In *Conf. on Design, Automation and Test in Europe (DATE), MARTE Workshop*, March 2008.